08장 서브루틴과 제어 추상화

【8.6.3】 반복자의 구현

예 8.75
동시 실행 루틴 기반 반복자 호출

예 6.68의 다음 for 루프를 살펴보자.

```
for i in from_to_by(first, last, step) do
    ...
end
```

컴파일러는 이를 다음과 같이 해석한다.

```
iter := new from_to_by(first, last, step, i, done, current coroutine)
while not done do
    ...
    transfer(iter)
destroy(iter)
```

루프가 완료된 후 구현은 iter가 소비한 공간을 해제한다.

예 8.76
동시 실행 루틴 기반 반복자 구현

from_to_by의 정의 자체는 매우 직관적이다.

```
coroutine from_to_by(from val, to val, by amt : int;
                     ref i : int; ref done : bool; caller : coroutine)
    i := from_val
    if by_amt > 0 then
       done := from_val ≤ to_val
       detach
       loop
          i +:= by_amt
```

```
            done := i ≤ to_val
            transfer(caller)          -- i를 양도함
      else
         done := from_val ≥ to_val
         detach
         loop
            i +:= by_amt
            done := i ≥ to_val
            transfer(caller)          -- i를 양도함
```

반복자가 호출자 문맥에서 변경할 수 있게 매개변수 i와 done을 참조로 전달한다. 반복자가 다음 루프 인덱스를 계산할 때 어떤 동시실행루틴을 재개해야 하는지를 말해주기 위해 호출자의 식별자를 마지막 인자로 전달한다. 호출자에 명시적으로 이름을 붙이기 때문에 그림 6.5처럼 반복자가 중첩되는 것은 쉽다.

단일 스택 구현

동시실행루틴이 반복자의 구현에 충분하지만 필연적인 것은 아니다. 더 간단한 단일 스택 구현도 가능하다. 주어진 반복자(예를 들어 from_to_by의 인스턴스)를 코드 내의 동일한 장소에서 항상 재개하기 때문에 반복자를 실행할 때마다 서브루틴 호출 스택은 항상 동일한 프레임을 가지고 있을 것이다. 게다가 yield문은 반복자의 주된 몸체에서만 나타날 수 있기(중첩된 루틴에서는 결코 그렇지 않기) 때문에 반복자가 호출자로 전이할 때마다 스택은 항상 동일한 프레임을 가지고 있을 것임을 확신할 수 있다. 이런 두 사실은 단일 중추 스택에 호출자의 프레임 꼭대기에 반복자의 프레임을 직접적으로 위치시킬 수 있음을 의미한다.

반복자를 생성할 때 그것의 프레임을 스택에 푸쉬한다. 그것은 한 값을 양도할 때 제어는 for 루프에 복귀하지만 반복자의 프레임을 그 스택에 남겨둔다. 루프의 몸체가 서브루틴 호출을 하면 그 호출에 대한 프레임을 반복자의 프레임 위쪽으로 할당할 것이다. 반복자가 재개되기 전에 제어가 루프에 복귀해야 하기 때문에 그 반복자가 그것을 다시 볼 기회를 가지기 전 그런 프레임을 다시 떠나야 할 것임을 안다. 그것이 서브루틴 자체를 호출할 필요가 있다면 그것 위의 스택은 없을 것이다. 마찬가지로 반복자가 어떤 서브루틴은 호출한다면 for 루프를 다시 실행하기 전에 복귀할(스택에서 그 프레임을 팝할) 것이다. 중첩된 반복자는 특별한 문제를 제시하지 않는다(©심화학습에 있는 연습문제 8.42를 보자).

자료 구조 구현

예 8.77
C#에서의 반복자 사용법

C# 2.0을 위한 컴파일러는 반복자의 다른 구현을 사용한다. 자바와 마찬가지로 C# 1.1은 반복자 객체를 제공했다. 그런 각 객체는 MoveNext와 Current 메소드를 제공하는 IEnumerator 인터페이스를 구현한다. 일반적으로 IEnumerable 인터페이스를 호출하는 객체(컨테이너)의 GetEnumerator 메소드를 호출해 반복자를 얻는다.

```
for (IEnumerator i = myTree.GetEnumerator(); i.MoveNext();) {
    object o = i.Current;
    Console.WriteLine(o.ToString());
}
```

예 8.78
C# 반복자의 구현

C# 2.0은 반복자 객체의 확장으로서 참 반복자를 제공한다. 프로그래머는 단지 한 개 이상의 yield return문을 포함하고 IEnumerator나 IEnumerable 유형의 반환형을 가진 메소드를 선언한다. 여기 후자의 예가 있다.

```
static IEnumerable FromToBy (int fromVal, int toVal, int byAmt)
{
    if (byAmt >= 0) {
       for (int i = fromVal; i <= toVal; i += byAmt) {
          yield return i;
       }
    } else {
       for (int i = fromVal; i >= toVal; i += byAmt) {
          yield return i;
       }
    }
}
```

컴파일러는 알아서 이 코드를 ◎(심화학습에 있는) 그림 8.15처럼 GetEnumerator 메소드를 가진 숨겨진 클래스로 변환한다. 이 코드 안에서 명시적 상태 변수는 마지막 yield문의 "프로그램 카운터"를 추적한다. 게다가 참 반복자의 지역 변수 i는 FromToByImpl 클래스의 자료 멤버를 가지고 루프의 반복에 걸쳐서 스택 프레임의 필요가 없는 반복자를 남는다. 아주 솔직히 말해서 컴파일러는 각 참 반복자를 반복자 객체로 변환한다.

재귀적 반복자도 특별한 어려움이 없다. 외부 반복자가 foreach 루프에 진입할 때 중첩된 반복자를 요구가 있는 즉시 할당한다. 자세한 내용에 대해 연습문제 8.43로 미룬다. 반복자 객체를 힙에서 할당하기 때문에 참 반복자의 C# 구현은 이전 하위 절의 스택 기반 구현보다 다소 느리다.

```
static IEnumerable FromToBy(int fromVal, int toVal, int byAmt) {
    return new FromToByImpl(fromVal, toVal, byAmt);
}
class FromToByImpl : IEnumerator, IEnumerable {
    enum State {starting, goingUp, goingDown, done}
    int i, tv, ba;
    State s;

    public FromToByImpl(int fromVal, int toVal, int byAmt) {
       fv = fromVal; tv = toVal; ba = byAmt; s = State.starting;
    }
    public IEnumerator GetEnumerator() {
       return this;
    }
    public object Current {
       get { return i; }
    }
    public bool MoveNext() {
       switch (s) {
          case State.starting :
             if (ba >= 0) {
                if (i <= tv) { s = State.goingUp; return true; }
                else { s = State.done; return false; }
             } else {
                if (i >= tv) { s = State.goingDown; return true; }
                else { s = State.done; return false; }
             }
          case State.goingUp :
             i += ba;
             if (i <= tv) return true;
             else { s = State.done; return false; }
          case State.goingDown :
             i += ba;
             if (i >= tv) return true;
             else { s = State.done; return false; }
```

그림 8.15 | C#에서의 참 반복자와 동일한 반복자 객체. 이 손으로 작성한 코드는 예 8.78에 일치한다. 그것은 소스 레벨에서 컴파일러가 중간 코드(intermediate code) 수준에서 생성하는 것인 시작 상태, 종료 상태와 각 yield문에 대한 상태와 함께 원 반복자의 프로그램 카운터를 추적하는 상태 기계를 의미한다. switch문이 가지는 한 상태에서 다음 상태로 이동하는 원 반복자에 코드 경로를 저장한다.(이어짐)

```
            default:                // 완전함을 위해
            case State.done : return false;
        }
    }
    public void Reset() {
        s = State.starting;
    }
}
```

그림 8.15 | C#에서의 참 반복자와 동일한 반복자 객체

✓ 확인문제

74. 동시 실행 루틴을 사용한 반복자의 "명백한" 구현을 설명하라.

75. 하나의 스택에서 다중 활성 반복자의 상태를 어떻게 유지할 수 있는지를 설명하라.

76. 참 반복자를 반복자 객체로 바꾸기 위해 C# 컴파일러가 사용하는 변환을 설명하라.