03장 이름, 유효 범위, 바인딩

3.4 유효 범위의 구현

정적 유효 범위든지 동적 유효 범위든지 언어 구현은 사실상 프로그램의 각 지점에서 이름-객체 바인딩을 유지해야 한다. 주요 차이점은 시점/시간이다. 정적 유효 범위를 사용하는 경우 컴파일러는 심볼 테이블을 사용해서 컴파일 시점에 바인딩을 추적한다. 동적 유효 범위를 사용하는 경우 해석기나 실행 시간 시스템은 관계 리스트나 중앙 참조 테이블을 사용해서 실행 시간에 바인딩을 추적한다.

【3.4.1】 심볼 테이블

정적 유효 범위를 사용하는 언어에서 컴파일러는 새로운 선언이 나올 때마다 이름-객체 바인딩을 심볼 테이블에 위치시키기 위해 `insert` 연산을 사용한다. 이미 선언되었어야 하는 이름의 사용이 나오면 컴파일러는 현존하는 바인딩을 검색하기 위해 `lookup` 연산을 사용한다. 유효 범위의 끝부분에 심볼 테이블로부터 이름을 삭제하기 위해 `remove` 연산을 수행함으로써 정적 유효 범위 지정의 가시성 규칙을 지원하려는 것은 매력적으로 보인다. 불행히도 몇 가지 요소로 인해 이러한 직관적인 접근 방법은 비현실적이다.

- 중첩된 유효 범위를 가지는 대부분의 언어에서 바깥쪽의 선언을 숨기기 위해 안쪽의 선언을 사용할 수 있는 것은 심볼 테이블이 주어진 이름에 대해 임의의 수의 매핑을 포함할 수 있어야 함을 의미한다. `lookup` 연산은 가장 안쪽의 매핑을 반환해야 하며 이 유효 범위가 끝나면 바깥쪽 매핑이 볼 수 있게 되어야 한다.

- 알골 계열어에서 레코드(구조체)와 클래스는 유효 범위 특성의 일부를 가지지만 잘 중첩된 구조는 공유하지 않는다. 의미 분석기는 레코드 선언이 나오면 레코드의 필드 이름을 (레코드가 중첩된 경우 재귀적으로) 기억해야 한다. 선언의 끝에서 필드명은 볼 수 없게 되어야 한다. 그러나 이후에 (`my_rec.field_name`에서와 같이) 프로그램 텍스트에 레코드형의 변수가 나오면 참조의 마침표 다음 부분은 갑자기 레코드 항목을 다시 볼 수 있어야 한다. 파스칼과 `with`문을 사용하는 기타 언어(7.3.3절)에서 항목 이름은 다중문 문맥에서 볼 수 있어야 한다.

- 3.3.3절에서 살펴봤듯이 이름은 때때로 선언되기 전에 사용되며, 이는 알골 계열의 언어에서도 그렇다. 예를 들어 알골과 C는 레이블에 대한 전방 참조를 허용한다. 파스칼은 포인터 선언에서 전방 참조를 허용한다. 모듈라 3는 모든 종류의 전방 참조를 허용한다.

- 3.3.3절에서 살펴봤듯이 C, C++, 에이다는 객체의 선언과 정의를 구별한다. 파스칼은 상호 재귀적인 서브루틴에 대해서도 유사한 기법을 사용한다. 선언이 나오면 컴파일러는 일관성을 위해 최종 정의를 검사할 수 있게 보이지 않는 상세를 기억해야 한다. 이러한 연산은 레코드의 항목 이름을 기억하는 것과 유사하다.

- 유효 범위의 끝에서 이름을 삭제하고 이 이름이 심볼 테이블에서 차지하는 공간을 회수하는 것이 바람직할 수 있는 반면 이에 대한 정보는 심볼릭 디버거가 사용할 수 있게 저장할 필요가 있을 수 있다. 디버거는 사용자가 실행 중인 프로그램을 조작(시작하거나 정지시키거나 자료를 읽거나 쓰는 것)할 수 있게 해주는 도구다. 사용자의 고수준 명령어(예를 들어 `my_firm^.revenues[1999]` 값의 출력)를 구문 분석하기 위해 디버거는 컴파일러의 심볼 테이블에 접근할 수 있어야 한다. 심볼 테이블을 실행 시간에 사용할 수 있게 하기 위해 컴파일러는 주로 최종적인 기계어 프로그램의 숨겨진 부분에 심볼 테이블을 저장한다.

예 3.34

르블랑–쿡 심볼 테이블

이러한 요소들을 충당하기 위해 대부분의 컴파일러는 심볼 테이블에서 어떠한 것도 절대로 삭제하지 않는다. 대신 `enter_scope`와 `leave_scope` 연산을 사용해서 가시성을 관리한다. 구현은 컴파일러에 따라 다양하며 여기서는 르블랑과 쿡[CL83]이 제안한 접근 방법을 설명한다.

유효 범위는 나타날 때 일련번호를 할당 받는다. 가장 바깥쪽의 유효 범위(미리 정의된 식별자를 포함하는 유효 범위)는 번호 0을 부여 받는다. 프로그래머가 선언한 전역 이름을 포함하는 유효 범위는 번호 1을 부여 받는다. 추가적인 유효 범위는 나올 때마다 순서대로 다음 번호를 부여 받는다. 모든 일련번호는 유일하다. 중첩된 서브루틴이 자연적으로 자신보다 큰 유효 범위의 번호들보다 큰 번호로 끝나는 경우를 제외하면 일련번호는 어휘적 중첩의 수준을 나타내지 않는다.

유효 범위와 무관하게 모든 이름은 하나의 큰 해시 테이블에 입력되며 이름으로 키가 주어진다. 그 다음 테이블의 각 필드는 기호 이름, 분류(변수, 상수, 유형, 프로시저, 필드명,

매개변수 등), 유효 범위 번호, 유형(다른 심볼 테이블 항목에 대한 포인터), 부가적인 항목, 특화 항목을 포함한다.

해시 테이블과 더불어 심볼 테이블은 현재 참조 환경을 구성하는 유효 범위를 순서대로 나타내는 유효 범위 스택을 가진다. 의미 분석기는 프로그램을 분석할 때 유효 범위에 진입하거나 유효 범위에서 나올 때마다 각기 이 스택을 푸쉬하고 팝한다. 유효 범위 스택의 항목들은 유효 범위 번호, 유효 범위가 닫혔는지에 대한 정보, 어떤 경우에는 그 이상의 정보를 포함한다.

테이블에서 이름을 검색하기 위해서는 찾고자 하는 이름과 일치되는 항목을 검색하면서 적절한 해시 체인을 살펴봐야 한다. 일치하는 각 항목에 대해 이 항목의 유효 범위가 볼 수 있는 것인지를 알아보기 위해 유효 범위 스택을 살펴본다. 들여오기와 내보내기는 테이블에 추가적인 항목을 생성함으로써 보통의 유효 범위 밖에서도 볼 수 있게 된다. 이러한 추가적인 항목들은 실제 항목에 대한 포인터를 포함한다. 유효 범위 번호 0을 가지는 항목에 대해서는 유효 범위 스택을 조사할 필요가 전혀 없다. 이러한 항목은 어디서나 볼 수 있다. 이 lookup 알고리즘의 의사코드는 그림 3.19와 같다.

```
procedure lookup(name)
    pervasive := best := nil
    적절한 체인을 찾기 위해 이름에 해시 함수를 적용
    foreach entry e on chain
       if e.name = name        -- 동일한 해시 값을 가지는 것만 고려
          if e.scope = 0
             pervasive := e
          else
             foreach scope s on scope stack, top first
                if s.scope = e.scope
                   best := e            -- 좀 더 가까운 인스턴스
                elsif best ≠ nil and then s.scope = best.scope
                   exit inner loop      -- 더 나은 것을 찾지 못함
                if s.closed
                   exit inner loop      -- 더 멀리 볼 수 없음
    if best ≠ nil
       while best is an import or export entry
          best := best.real entry
       return best
    elsif pervasive ≠ nil
       return pervasive
    else
       return nil        -- 이름을 찾지 못함
```

그림 3.19 | 르블랑–쿡 심볼 테이블의 lookup 연산

예 3.35

예제 프로그램의 심볼 테이블

◎(심화학습에 있는) 그림 3.20의 우측 하단 부분에는 모듈라 2와 비슷한 프로그램의 골격이 나와 있다. 이 그림의 나머지 부분은 프로시저 P2에 있는 with문의 참조 환경을 위한 심볼 테이블 설정을 보여준다. 유효 범위 스택은 각기 with문, 프로시저 P2, 모듈 M, 전역 유효 범위를 나타내는 네 개의 항목을 포함한다. with문에 대한 유효 범위는 이 유효 범위의 이름(필드)이 속하는 특정 레코드 변수를 나타낸다. 가장 바깥쪽의 어디서나 볼 수 있는 유효 범위는 명시적으로 표현하지 않는다.

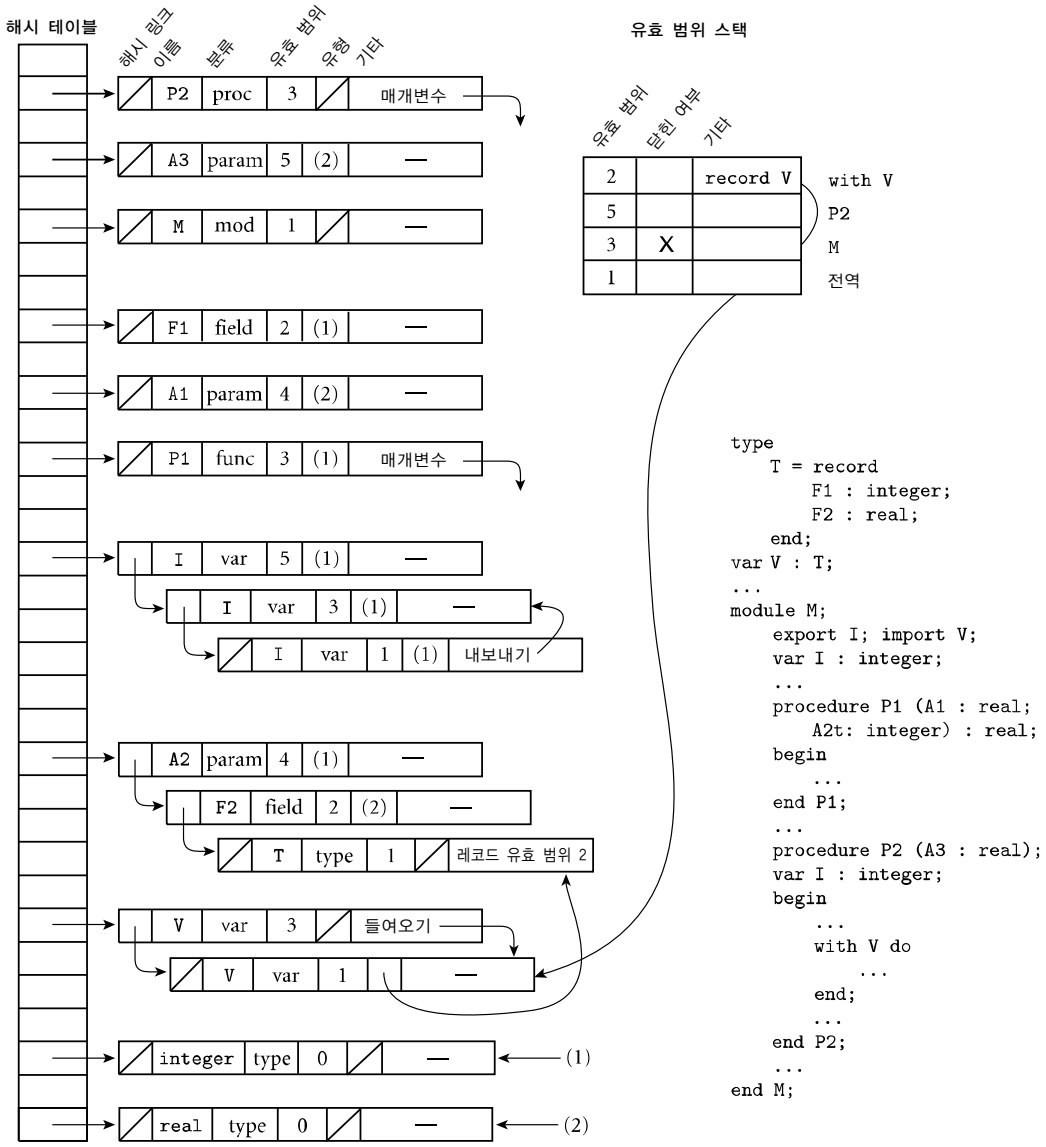

그림 3.20 | 모듈라 2와 비슷한 언어로 작성한 예제 프로그램의 르블랑–쿡 심볼 테이블. 유효 범위 스택은 프로시저 P2에 있는 with문의 참조 환경을 나타낸다. 명료성을 위해 유형 필드에서 심볼 테이블 항목으로의 integer와 real을 위한 포인터 중 다수는 화살표 대신 괄호로 묶은 (1)과 (2)로 나타냈다.

심볼 테이블의 키가 이름으로 주어지기 때문에 주어진 이름에 대한 모든 항목은 동일한 해시 체인에 나온다. 이 예에서 A2, F2, T는 해시 충돌 때문에 역시 동일한 체인에서 끝난다. 변수 V와 I(M의 I)는 닫힌 유효 범위 M의 경계 밖에서도 보일 수 있게 하기 위해 추가적인 항목을 가진다. P2 내부에서 I에 대한 lookup 연산은 P2의 I를 찾을 것이다. M의 I에 대한 항목들은 모두 보이지 않는다. 유형 T에 대한 항목은 with문 동안 유효 범위 스택에 푸쉬될 유효 범위 번호를 나타낸다. 각 서브루틴에 대한 항목은 호출 분석에서의 사용을 위해 서브루틴의 매개변수들을 연결하는 리스트의 헤드 포인터를 포함한다(이러한 체인의 추가적인 연결은 보이지 않았다). 코드 생성 동안 많은 심볼 테이블 항목은 크기나 실행 시간 주소와 같은 정보에 대한 부가적인 항목을 포함하게 된다.

【3.4.2】 관계 리스트와 중앙 참조 테이블

(심화학습에 있는) 그림 3.21은 동적 유효 범위의 두 가지 주요 구현을 그림으로 나타낸 것이다. 관계 리스트는 간단하고 훌륭하지만 매우 비효율적이다. 중앙 참조 테이블은 별도의 유효 범위 스택이 없는 단순화된 르블랑–쿡 심볼 테이블과 유사하다. 중앙 참조 테이블은 유효 범위 진입점과 출구에서 관계 리스트보다 더 많은 작업을 필요로 하지만 lookup 연산을 더 빠르게 수행한다.

예 3.36
리스프의 A 리스트 검색

리스프에서 A 리스트는 사전 추상화를 위해 널리 쓰이며 대부분의 리스프 계열 언어에서 풍부한 고유 함수 집합이 A 리스트를 지원한다. 그러므로 리스프 해석기가 이름–값 바인딩을 유지하기 위해 A 리스트를 사용하는 것과 실행 중인 프로그램이 이 리스트를 볼 수 있게 명시적으로 나타내는 것은 당연하다. 바인딩은 유효 범위에 진입할 때 생성되며 유효 범위를 떠나거나 유효 범위로부터 복귀할 때 파괴되므로 A 리스트는 스택으로 동작한다. 실행 시간에 실행이 유효 범위에 진입하면 해석기는 이 유효 범위에서 선언된 이름들에 대한 바인딩을 A 리스트의 전면에 푸쉬한다. 실행이 최종적으로 유효 범위를 벗어날 때 이러한 바인딩이 제거된다. 수식의 이름이 가지는 의미를 찾기 위해 해석기는 적절한 바인딩을 찾을 때까지(또는 A 리스트의 끝에 도달할 때까지(이 경우 오류가 발생한다)) A 리스트의 앞부분부터 검색한다. A 리스트의 각 항목은 의미 검사(예를 들어 7.2절에서 살펴볼 유형 검사)를 수행하고 메모리 위치를 점유하는 변수와 다른 객체를 찾는 데 필요한 모든 정보를 포함한다. (심화학습에 있는) 그림 3.21의 윗 부분에서 A 리스트의 첫 번째(최상단) 항목은 가장 최근에 나온 선언(프로시저 P의 I)을 나타낸다. 두 번째 항목은 프로시저 Q의 J를 나타낸다. 그 아래로 전역 기호 Q, P, J, I와 (명시적으로 나타내지는 않았지만) 리스프 해석기가 제공하는 미리 정의된 이름들이 나온다.

프로그램의 참조 환경을 나타내기 위해 관계 리스트를 사용함으로써 발생하는 문제는 리스트에서 특정 항목을 찾는 데 긴 시간이 걸릴 수 있다는 점이다. 이는 특히 찾고자

하는 항목이 프로그램 실행 초기에 나온 유효 범위에서 선언된 객체로 지금은 리스트의 하단에 묻혀있는 경우 더욱 그렇다.

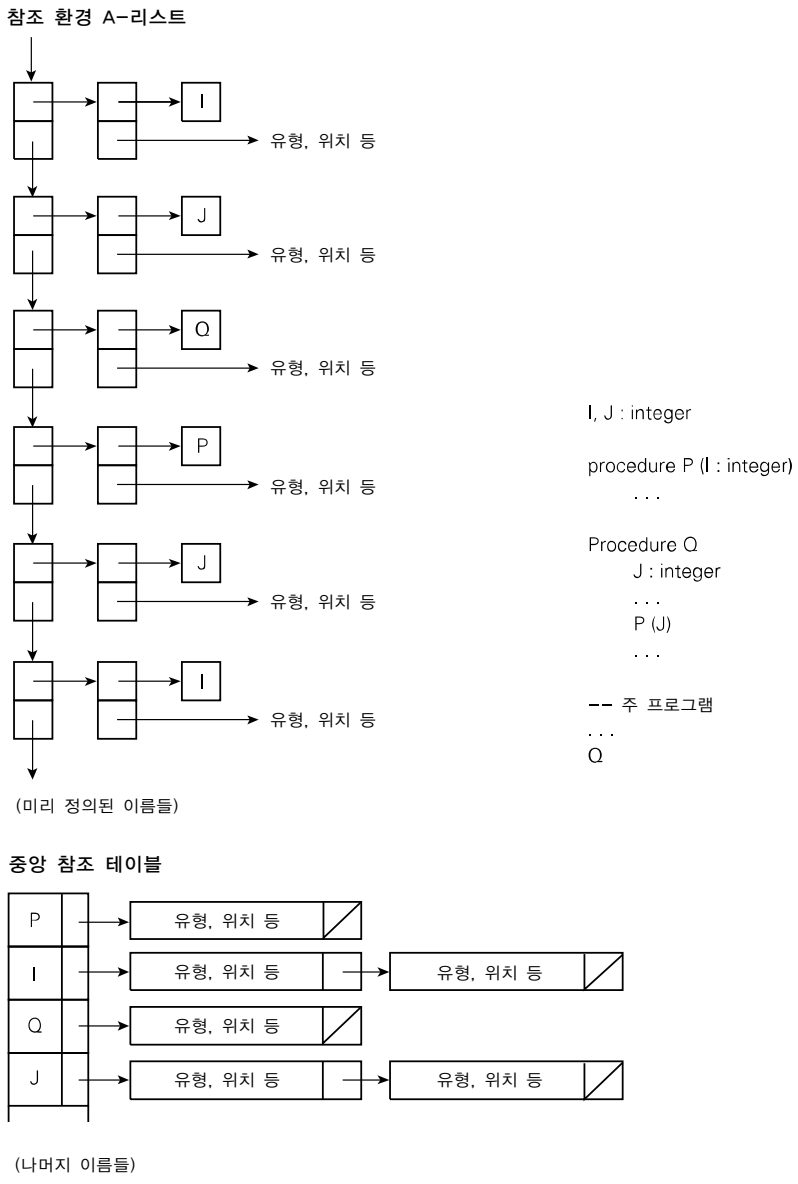

그림 3.21 | 관계 리스트(좌측 상단)나 중앙 참조 테이블(하단)을 이용한 동적 유효 범위 지정의 구현. 두 구조 모두 사실상 오른쪽의 코드에서 주 프로그램이 Q를 호출하고 차례로 Q가 P를 호출한 직후의 참조 환경을 나타낸다.

예 3.37
중앙 참조 테이블

중앙 참조 테이블은 좀 더 빠른 접근을 위해 설계되었으며, 프로그램에서 구별되는 모든 이름마다 공간(slot)을 하나씩 둔다. 테이블 공간은 실행 시간에 나오는 선언들의

리스트(스택)를 포함하는데, 가장 최근에 나온 것이 리스트의 처음에 위치한다. 이제 이름 검색은 쉽다. 테이블의 적절한 공간에 있는 리스트의 첫 항목이 현재 의미를 나타낸다. 그림 3.21의 아래 부분에서 `I` 리스트의 첫 번째 항목은 프로시저 `P`의 `I`며 두 번째 항목은 전역 `I`다. 프로그램이 컴파일되고 이름들의 집합을 컴파일 시점에 안다면 각 이름은 테이블에 정적으로 할당된 공간을 가질 수 있다. 이렇게 할당된 공간은 컴파일된 코드가 바로 참조할 수 있다. 프로그램이 컴파일되지 않거나 이름들의 집합을 정적으로 알 수 없다면 실행 시간에 적절한 공간을 찾기 위해 해시 함수가 필요하다.

실행 시간에 제어가 새로운 유효 범위로 진입하면 항목들은 이 유효 범위에서 이름이 (재)선언된 중앙 참조 테이블의 모든 리스트의 처음 부분에 푸쉬되어야 한다. 제어가 최종적으로 유효 범위를 벗어날 때 이러한 항목들은 팝되어야 한다. 수반되는 작업의 비용은 A 리스트의 푸쉬나 팝보다 다소 크지만 매우 큰 것은 아니며 `lookup` 연산은 훨씬 빨라진다. 전적 유효 범위 지정을 사용하는 언어를 위한 컴파일러의 심볼 테이블과 달리 주어진 유효 범위를 위한 중앙 참조 테이블의 항목은 유효 범위가 실행을 완료할 때 저장할 필요가 없으며 실행 완료 후 해당 공간은 회수된다.

리스프 커뮤니티에서는 관계 리스트를 통한 동적 유효 범위 지정의 구현을 때때로 **깊은 바인딩**이라고 하는데 이는 `lookup` 연산이 A 리스트를 임의의 깊이만큼 검색할 수도 있기 때문이다. 중앙 참조 테이블을 통한 구현은 때때로 **얕은 바인딩**이라고 하는데 이는 현재 연결을 주어진 참조 체인의 첫 부분에서 찾기 때문이다. 불행히도 "깊은 바인딩과 얕은 바인딩"이라는 용어는 3.5절에서 살펴본 것과 같이 전혀 다른 의미로 보다 널리 사용된다. 잠재적인 혼동을 피하기 위해 일부 저자는 동적 유효 범위 지정의 구현에 대해 "깊은 접근과 얕은 접근"[Seb04]이나 "깊은 검색과 얕은 검색"[Fin96]을 사용한다.

✓ 확인문제

40. 심볼 테이블이 제공하는 기본 연산을 나열하라.

41. 르블랑-쿡 방식 심볼 테이블의 구현을 간단히 설명하라.

42. 일반적으로 어떤 이름의 유효 범위가 끝나는 지점에서 컴파일러가 심볼 테이블로부터 이 이름을 제거하지 않는 이유는 무엇인가?

43. 동적 유효 범위 지정을 구현하기 위해 사용하는 관계 리스트와 중앙 참조 테이블 자료 구조를 설명하라. 이들 간의 트레이드오프를 요약하라.