프로그래밍 언어론

Scope(영역)

컴퓨터공학과
이만호

충남대학교
CHUNGNAM NATIONAL UNIVERSITY

Scope(영역)

학습목표

프로그램에서 변수의 선언과 사용 사이에 관계를 맺어주는 방법에 대해서 학습한다.

학습내용

- 변수의 Scope(영역)
- Referencing Environment(참조환경)

목 차

알고가기 1

오른쪽에 주어진 프로그램 골격에서, block D의 DDD 지점을 실행할 때, 사용 가능한 변수들을 모두 나열하고 있는 것은?

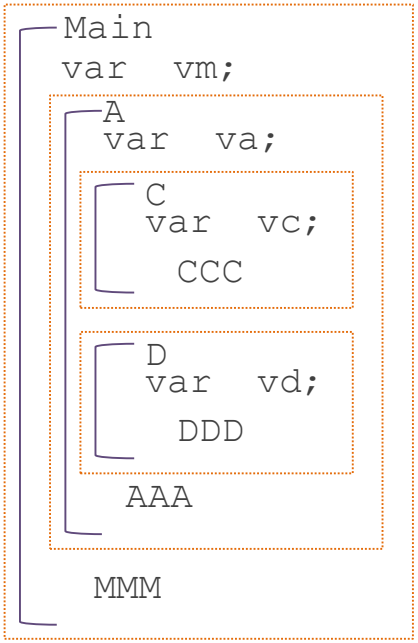
* 힌트: 모든 box는 외부에서는 내부가 보이지 않지만, 내부에서는 외부가 보인다고 생각해서, 각 block에서 보이는 변수를 선택하면 된다.

(a) vm, va, vc

(b) vd

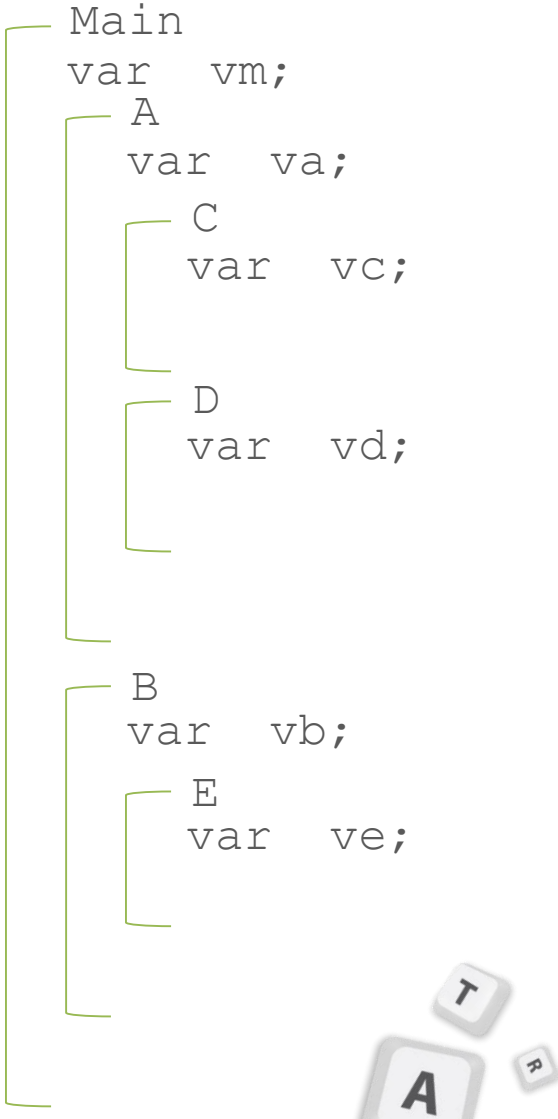
(c) va, vc, vd

(d) vm, va, vd



변수의 Scope(영역)

- 프로그램에서 선언된 변수를 사용할 수 있는 영역
- 프로그램의 어떤 지점에서 사용할 수 있는 변수가 있으면, 그 변수는 그 지점에서 가시적(visible)이라고 한다.



변수의 Scope와 변수의 종류

지역변수 (Local Variable)

- Subprogram/block에서 선언된 변수
  - Subprogram/block 내부에서는 가시적(visible)이다.
  - Subprogram/block 외부에서는 가시적이지 않다.

예 오른쪽 그림에서 각 block의 지역변수

Main:{vm}, A:{va}, B:{vb},
C:{vc}, D:{vd}, E:{ve}

비지역변수 (Nonlocal Variable)

- Subprogram/block에서 가시적이나, 그 block에 선언되어 있지 않다.

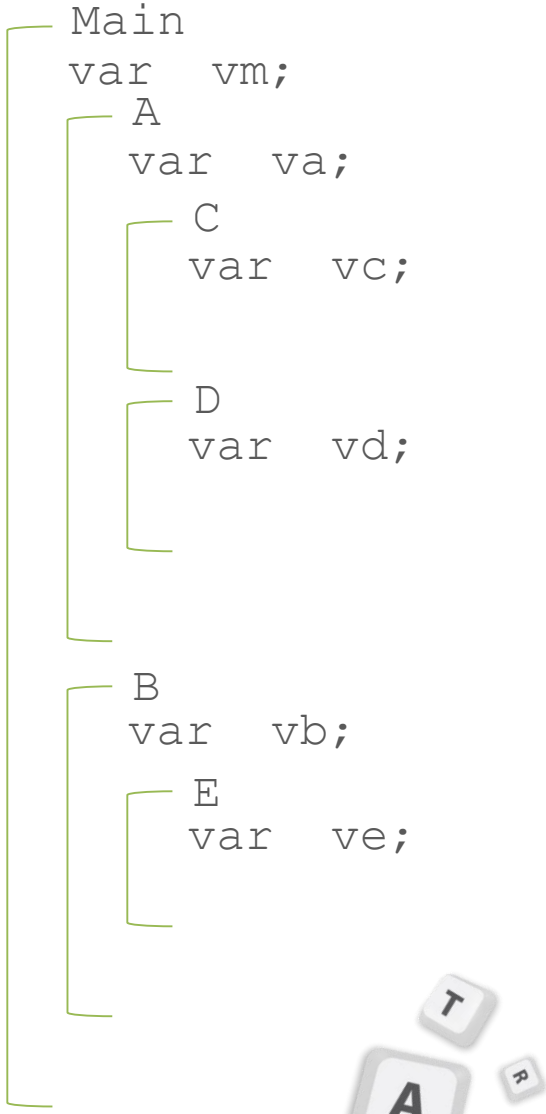
예 오른쪽 그림에서 각 block의 비지역변수

A:{vm}, C:{vm,va}, D:{vm,va}
B:{vm}, E:{vm,vb}

전역변수(Global variable)

- 프로그램 전체 영역에서 가시적인 변수

예 오른쪽 그림에서 전역변수: {vm}



Scope Rule(영역 규칙)

프로그램의 어떤 지점에서 사용된 변수 이름을 어디에서 선언된 변수와 대응시켜줄 것인지를 결정하는 규칙

두 가지 규칙

- Static Scope Rule (정적 영역규칙, SSR로 표기하기로 함)
- Dynamic Scope Rule (동적 영역규칙, DSR로 표기하기로 함)

중첩된 구조 (nested structure)

- Subprogram이 내부에 중첩된(nested) subprogram을 정의할 수 있는 언어
  - **Ada**
- 중첩된 subprogram을 정의할 수 없는 언어
  - **C**
- Block으로 중첩될 수 있다.

```
int foo()
{ ...
  while (...) { int k; ... }
  ... }
```

```
Main
var  vm; x;
  A
  var  va; x;
    C
    var  vc;
     x (of ?)
    D
    var  vd; x;
     x (of ?)
   x (of ?)
  B
  var  vb;
    E
    var  ve; x;
    x (of ?)
   x (of ?)
 x (of ?)
```

Static Scope Rule (SSR, 정적 영역규칙)

Subprogram들의 공간적 배치 구조(textual layout)에 근거하고 있다.
즉 공간적(spatial)이다.

- Static parent(정적 부모) — 자신을 직접 둘러싸고 있는 Subprogram
  - `Main: A와 B의 static parent,  A: C와 D의 static parent`
- Static ancestors(정적 조상) — 자신을 둘러싸고 있는 모든 Subprogram들
  - `Main, A: C와 D의 static ancestors,  Main,B: E의 static ancestors`

사용된 변수 이름에 대한 해당 변수의 선언문을 아래와 같은 순서로 찾는다.

1. 변수 이름이 사용된 Subprogram 내에서 찾는다.
   (찾으면 지역변수)
2. 변수 이름이 사용된 Subprogram의 static parent 내에서 찾는다.
   (찾으면 비지역변수)
3. 선언문을 찾을 때까지 2번 과정을 반복하여,
   모든 static ancestors 내에서 순서대로 찾는다.
   (마지막에 찾으면 전역변수)

※ 위 과정에서 선언문을 찾지 못하면, 오류 처리함.

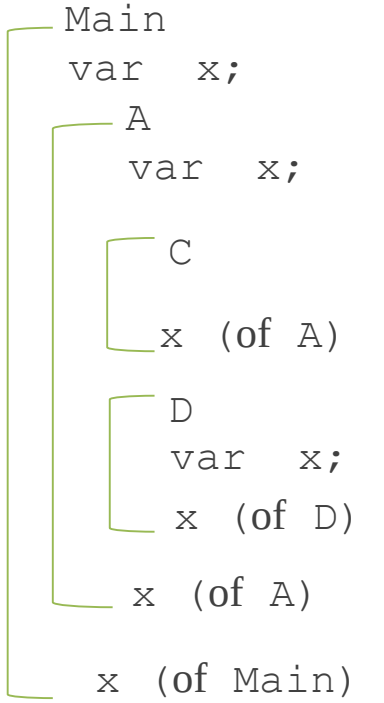
은폐된 변수 (Hidden Variable)

- 변수 이름이 사용된 지점에서 가까운 곳에 선언문이 있으면, 그 곳의 static ancestor에 선언된 동일한 이름의 변수는 변수 사용 지점에서는 보이지 않는다.
  - C 내부에서 사용된 변수 x는 A에서 선언된 변수이고, Main에서 선언된 x는 은폐된다.
  - D 내부에서 사용된 변수 x는 D에서 선언된 변수이고, A와 Main에서 선언된 x는 은폐된다.
  - A 내부에서 사용된 변수 x는 A에서 선언된 변수이고, Main에서 선언된 x는 은폐된다.
- Ada에서, D 내부에서 은폐되어 있는 A에서 선언된 변수 x 사용하기

```
A.x
```

- C++에서, foo에서 은폐되어 있는 전역변수로 선언된 변수 x 사용하기

```
::x
```

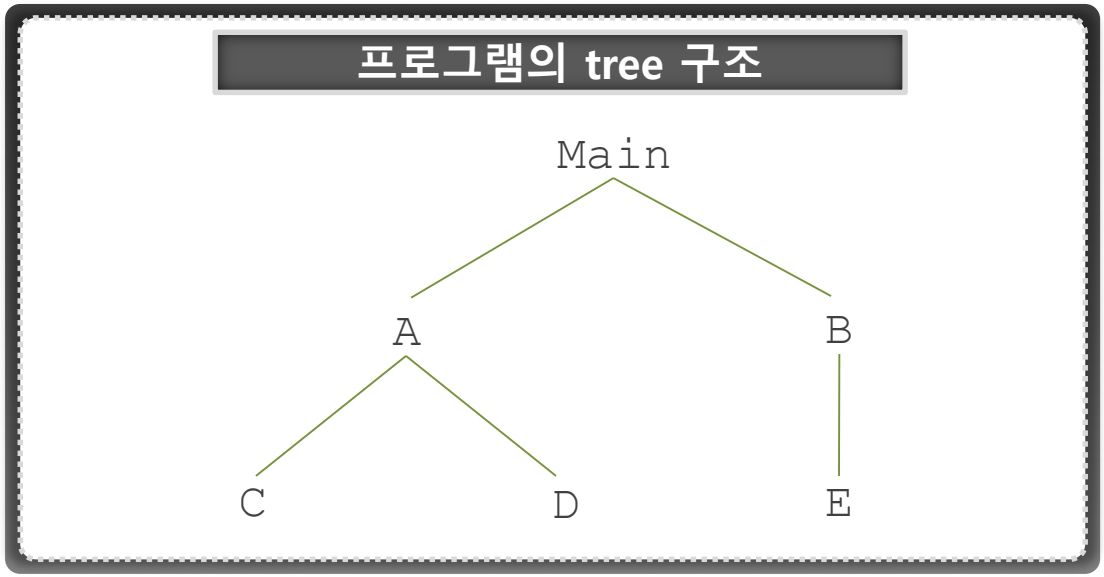



```
int x;
...
int foo()
{ int x;
  ...
  x (of foo)
  ::x
  ...
}
```

Subprogram의 선언

예제 프로그램의 골격

```
Main
    A
        C
        D
        call C;
        call D;
    B
        E
        call A;
        call E;
    call A;
    call B;
```



- Subprogram 안에 변수를 선언하듯이, 중첩된 subprogram도 subprogram 안에 선언되어 있다고 생각할 수 있다.
  - Main 안에 A와 B가 선언되어 있다.
  - A 안에 C와 D가 선언되어 있다.
  - B 안에 E가 선언되어 있다.

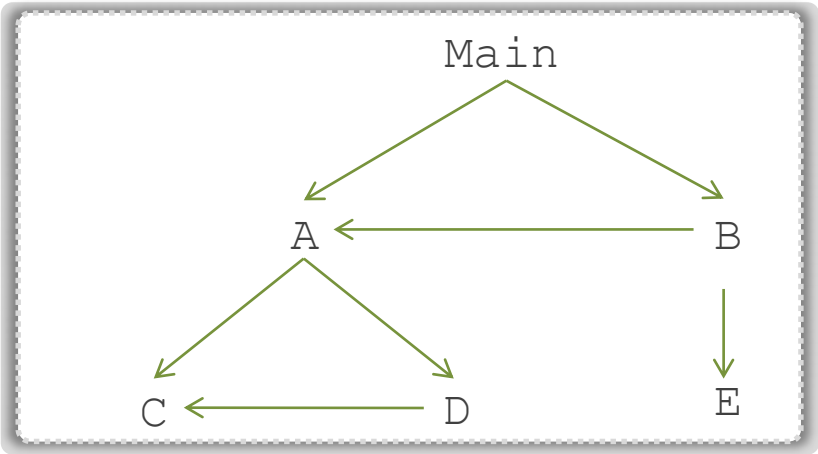
Subprogram의 호출

예제 프로그램의 골격

```
Main
    A
        C
        D
        call C;
        call D;
    B
        E
        call A;
        call E;
    call A;
    call B;
```

Desirable Call Graph

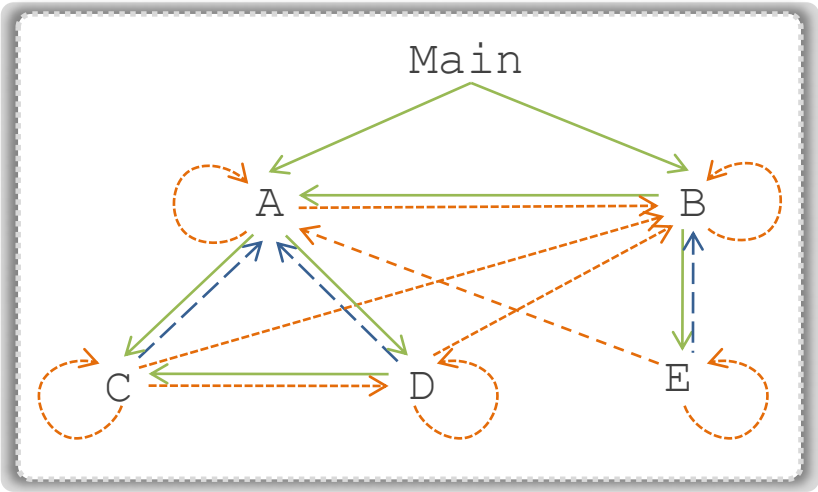
- 자신이 선언한 subprogram을 호출
- 자신보다 앞에 선언된 subprogram을 호출



Possible Call Graph

- Desirable call(→)
- Static ancestor를 호출 (-->)
- Static ancestor 안에 선언되어 있고, forward call이 허용되는 경우 (⇢)

※ Desirable call 이외의 호출은 주의해서 사용해야 한다.



생각할 문제

- C에서 C 자신을 호출할 수 있는 이유
- C에서 A를 호출할 수 있는 이유
- C에서 B를 호출할 수 있는 이유

지역변수 선언 지점

함수의 지역변수 선언 지점

- 일반적으로 함수의 맨 위에서 선언된다.
- 함수 내 임의의 장소에서 선언될 수도 있다

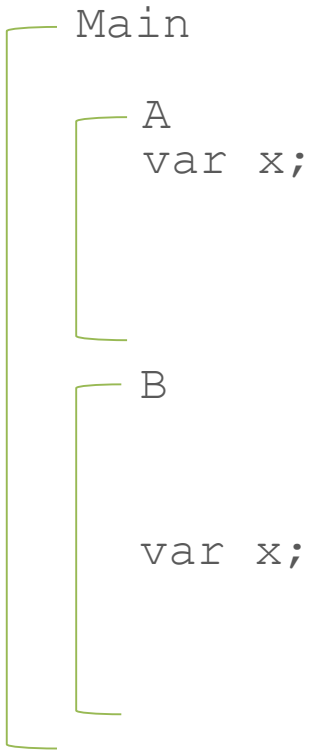
함수의 중간에 선언된 변수의 영역(Scope)

- 선언된 지점부터 함수의 끝

  예 C99, C++, Java

- 함수 전체

  예 C#



전역변수 - C

선언(declaration) vs. 정의(definition)

- type과 일부 속성을 지정함
- 선언: 기억장소 할당 안 함. 값의 초기화 못 함. 변수의 정보만 표시
- 정의: 기억장소 할당함. 값의 초기화 가능

C에서 전역변수 사용하기

- 정의 지점 이후부터 모든 함수에서 사용 가능
- 정의 지점 이전에 있는 함수 안에서 사용하려면, 함수 안에 "extern type-name 전역변수-이름"과 같이 선언해야 함

예

```
int ga;
int foo() {
    extern int gb;
    ga = 3;
    gb = 5;
    ...
}
int gb;
```

전역변수 - PHP

PHP에서 전역변수 사용하기

- 함수 외부에서 배정문(assignment statement)의 LHS에 나타난 변수는 모두 전역변수임
- 전역변수의 Scope
  - 선언 지점부터 프로그램 끝까지
- 함수 내부에서 전역변수 사용하기
  - 전역변수와 동일한 이름의 지역변수가 선언되어 있는 경우
    `$GLOBAL['전역변수-이름']`
  - 전역변수와 동일한 이름의 지역변수가 선언되어 있지 않은 경우
    `global $전역변수-이름; ... $전역변수-이름 ...`

예

```
$day = "Monday";    $month = "May";     // 전역변수
function calendar() {
   $day = "Tuesday";          // day:지역변수
   global $month;
   print "$day <br />";          // 출력 지역변수: Tuesday
   $gday = $GLOBAL['day'];   // day:전역변수
   print "$gday <br />"          // 출력: Monday
   print "$month <br />";        // 출력: May
}
```

전역변수 - Python

Python에서 전역변수 사용하기

- 함수 외부에서 배정문(assignment statement)의 LHS에 나타난 변수는 모두 전역변수임
- 전역변수의 Scope
  - 선언 지점부터 프로그램 끝까지
- 함수 내부에서 전역변수 사용하기
  - 함수 안에 "global 전역변수-이름"과 같이 선언해야 함

예-1

```
day = "Monday"
def foo();
    print day          // 출력: Monday
```

예-2

```
day = "Monday"
def foo();
    global day
    print day          // 출력: Monday
    day = "Tuesday"
    print day          // 출력: Tuesday
```

Static Scope Rule(SSR)에 대한 평가

장점

- Reliability 향상
  - 비지역 변수에 대한 접근이 방법이 결정적(deterministic)이고, 효과적이다.
- Readability 향상
  - 비지역 변수의 속성을 정적으로 알 수 있다.

단점

- 전역변수(global variable)는 모든 subprogram에 가시적이다.
- 바람직하지 않은 호출이 가능하여, 프로그래머의 의도와 다르게 호출해도 오류로 검출되지 않는다.→ 유지관리 비용 증가
- 프로그램이 개발된 후 요구 사항이 변경되었을 경우, 재 설계 어려움이 크다.
  → 프로그램 구조가 낮은 중첩 수준을 갖는 subprogram들의 나열이 되기 쉽다.
  (중첩을 허용하지 않는 C와 같은 형태)
- 프로그래밍언어가 subprogram의 중첩을 허용하지 않는 방향으로 발전하고 있다.

Dynamic Scope Rule(DSR)

단위 프로그램들의 호출순서(calling sequence)에 근거하고 있다.
즉 시간적(temporal)이다.

➔ Dynamic Scope은 실행 시간에 결정된다.

사용된 변수 이름에 대한 해당 변수의 선언문을 아래와 같은 순서로 찾는다.

1. 변수 이름이 사용된 subprogram 내에서 찾는다.
   (찾으면 지역변수)
2. 변수 이름이 사용된 subprogram을 호출한 subprogram 내에서 찾는다.
3. 선언문을 찾을 때까지 2번 과정을 반복하여,
   호출된 subprogram을 역순으로 찾는다.

※ 위 과정에서 선언문을 찾지 못하면, 오류 처리함.

Dynamic Scope Rule(DSR)의 예

예제 프로그램의 골격

```
Main
 var   x;

  Sub1
    var   x;

    ...
    call  Sub2;
    ...

  Sub2
    ...
    … x …;
    ...

...
if (…) call Sub1
else   call Sub2;
...
```

문제

Sub2에서 사용된 변수 x는 어떤 선언문에서 선언된 변수 x인가?

1. 아래와 같은 호출 순서를 가정하자.

   Main calls Sub1,
   Sub1 calls Sub2,
   Sub2에서 변수 x 사용.

   - SSR을 적용하는 경우
     - Sub2에는 x가 선언되어 있지 않음.
     - Sub2의 static parent인 Main에 x가 선언되어 있음.

       답: Main 안에 선언된 x
   - DSR을 적용하는 경우
     - Sub2에는 x가 선언되어 있지 않음.
     - Sub2를 호출한 Sub1에 x가 선언되어 있음.

       답: Sub1 안에 선언된 x

2. 아래와 같은 호출 순서를 가정하자.

   Main calls Sub2,
   Sub2에서 변수 x 사용.

   - SSR을 적용하는 경우: Main 안에 선언된 x
   - DSR을 적용하는 경우: Main 안에 선언된 x

Dynamic Scope Rule(DSR)에 대한 평가

단점

- **Reliability 저하**
  - > Subprogram을 호출한 subprogram의 지역 변수는 호출된 subprogram에서 가시적이기 때문에, 다른 subprogram에 의한 지역 변수의 수정을 방지할 방법이 없다.
- **비지역 변수에 대한 dynamic type checking 때문에 실행 시간 증가**
  - > 비지역 변수에 대한 type check를 정적으로 할 수 없다.
- **Readability 저하**
  - > 사용된 변수의 속성이 실행 시 호출 순서에 따라 다르다.
- **비지역 변수에 대한 접근 시간이 SSR에 비해 더 소요된다.**
  - > SSR: 중첩된 깊이에 좌우됨
  - > DSR: 일련의 호출에서 호출 단계 수에 좌우됨

장점

- **Subprogram을 호출할 때, parameter(매개 변수)를 전달할 필요가 없다.**
  - > Parameter(인자)로 전달할만한 변수는 대부분 호출하는 subprogram의 지역 변수이고, 이들 지역 변수는 호출된 subprogram에 가시적이다.

Static Scope Rule(SSR) vs. Dynamic Scope Rule(DSR)

- SSR을 적용하는 언어로 작성된 프로그램은 DSR을 적용하는 언어로 작성된 프로그램에 비해서
  Readability, Reliability, 실행 비용 면에서 우수하다.

- SSR을 적용하는 언어가 DSR을 적용하는 언어보다 많다.

- DSR을 적용하는 언어는 SSR도 지원한다.
  - Perl, Common Lisp

- DSR을 적용하던 언어에서 SSR을 적용하는 언어로 발전하는 경우도 있다.
  - Scheme : Lisp과 같은 유형의 언어로서, Lisp의 dialect(방언)라고 한다.

- 요즈음 개발되는 대부분의 언어는 SSR을 적용한다.

Scope와 존속기간(Lifetime)

Scope(영역)와 존속기간(lifetime)은 밀접하게 관련된 것처럼 보인다.

- **Subprogram의 지역변수(Stack-dynamic 변수)**
  - **Scope**: 변수 선언문부터 subprogram의 끝 부분까지
  - **존속기간**: Subprogram이 호출되어 실행을 시작한 시점부터 subprogram의 실행이 종료되어 **return**할 때까지

Scope와 존속기간은 완전히 별개의 개념이다.

- **Scope은 공간적(spatial) 개념이고, 존속기간은 시간적(temporal) 개념이다.**
- **Subprogram에서 `static`으로 선언된 지역변수(static 변수)**
  - **Scope**: 변수 선언문부터 subprogram의 끝 부분까지
  - **존속기간**: 프로그램의 시작 시점부터 프로그램의 종료 시점까지

Scope와 Lifetime(존속기간)–예

c언어

```
void goo()
{ int  r;
  ...
}
     void foo(...)
     { int          p;
       static int   q;
       ... goo(); ...
     }
```

	Scope	존속기간(Lifetime)
p	foo 내부	foo가 호출됨 ~ goo 실행 ~ foo에서 return
q	foo 내부	프로그램 시작 ~ 프로그램 종료
r	goo 내부	goo가 호출됨 ~ goo에서 return

Referencing Environment(참조환경)

프로그램의 어느 한 문장(statement)의 참조환경(referencing environment)

- 문장에 가시적인 모든 이름(name)들의 집합
- SSR이 적용되는 언어의 경우
  - 지역 변수(local variable)
  - 정적 조상(static ancestors)의 지역변수 (전역 변수 포함)
  - 은폐된 변수(hidden variable)는 제외된다.
- DSR이 적용되는 언어의 경우
  - 지역 변수(local variable)
  - 문장이 실행될 때까지 active subprogram들의 지역변수
  - 은폐된 변수(hidden variable)는 제외된다.

※ **active subprogram**이란 subprogram이 호출되어 실행이 시작되었으나, 아직 종료되지 않은 상태에 있는 subprogram을 말한다.

참조환경의 예

SSR 적용

```
Main
  var  A;
  Sub1
    var  B, X;
    begin
      ... ②{B,X(of Sub1),A,Sub1,Sub2}
    end;
  Sub2
    var  C, X;
    Sub3
      var  D, X;
      begin
        ... ④
          {D,X(of Sub3),C,Sub3,A,Sub1,Sub2}
      end;
    begin
      ... ③
        {C,X(of Sub2),Sub3,A,Sub1,Sub2}
    end;
  begin
    ... ①{A,Sub1,Sub2}
end.
```

DSR 적용

```
void sub1() {
  int  C,X;
  ... ③{C,X(of sub1),B,A}
}
void sub2() {
  int  B,X;
  ... ②{B,X(of sub2),A}
  sub1();
}
void main() {
  int  A,X;
  ... ①{A,X(of main)}
  sub2();
}
```

평가하기

마지막으로 내가 얼마나 이해했는지를 한번 확인해 볼까요?

총 2문제가 있습니다.

START

평가하기 1

1. **SSR과 DSR에 관하여 올바르게 설명하고 있는 것은? ("SSR을 적용하는 언어"를 S라고 표현하고, "DSR을 적용하는 언어"를 D라고 표현하기로 한다.)**

   (a) S로 작성된 프로그램은 실행해 보지 않아도 사용된 각 변수에 대한 선언문을 알 수 있다.

   (b) D로 작성된 프로그램을 실행할 때, 특정 장소에서 사용된 프로그램에 대한 선언문은 항상 동일하다.

   (c) D로 작성된 프로그램은 S로 작성된 프로그램보다 readability가 우수하다.

   (d) S의 수는 D의 수보다 적다.

평가하기 2

```
Main;
  var  m;
  Asub;
    var  a;
    Csub;
      var  c;
      begin
        ...   --- $$
      end;
    begin
      call Csub;
    end;
  Bsub;
    var  b;
    begin
      call Asub;
    end;
  begin
    call Bsub;
  end;
```

2. 왼쪽에 주어진 프로그램 골격에서, SSR과 DSR 각각에 대해서, $$로 표현된 지점에서 가시적인 변수를 모두 나열하고 있는 것은?

(a) SSR: m, a, c DSR: m, a, b, c

(b) SSR: m, a, c DSR: m, a, c

(c) SSR: m, a, b, c DSR: m, a, c

(d) SSR: m, a, b, c DSR: m, a, b, c

정리하기

- **변수의 Scope(영역)**
  - 프로그램에서 변수가 가시적(visible)인 영역
  - Scope에 따른 변수의 분류
    - 지역변수 (Local Variable)
    - 비지역변수 (Nonlocal Variable)
    - 전역변수(Global variable)
- **Scope Rule(영역규칙)**
  - Static Scope Rule
  - Dynamic Scope Rule
- **Scope(영역)와 존속기간(Lifetime)**
  - 변수의 scope은 공간적 개념이고, 존속기간은 시간적 개념으로, 둘은 완전히 별개의 개념이다.
- **문장(statement)의 참조환경(Referencing Environment)**
  - 문장에 가시적인 모든 이름(name)들의 집합