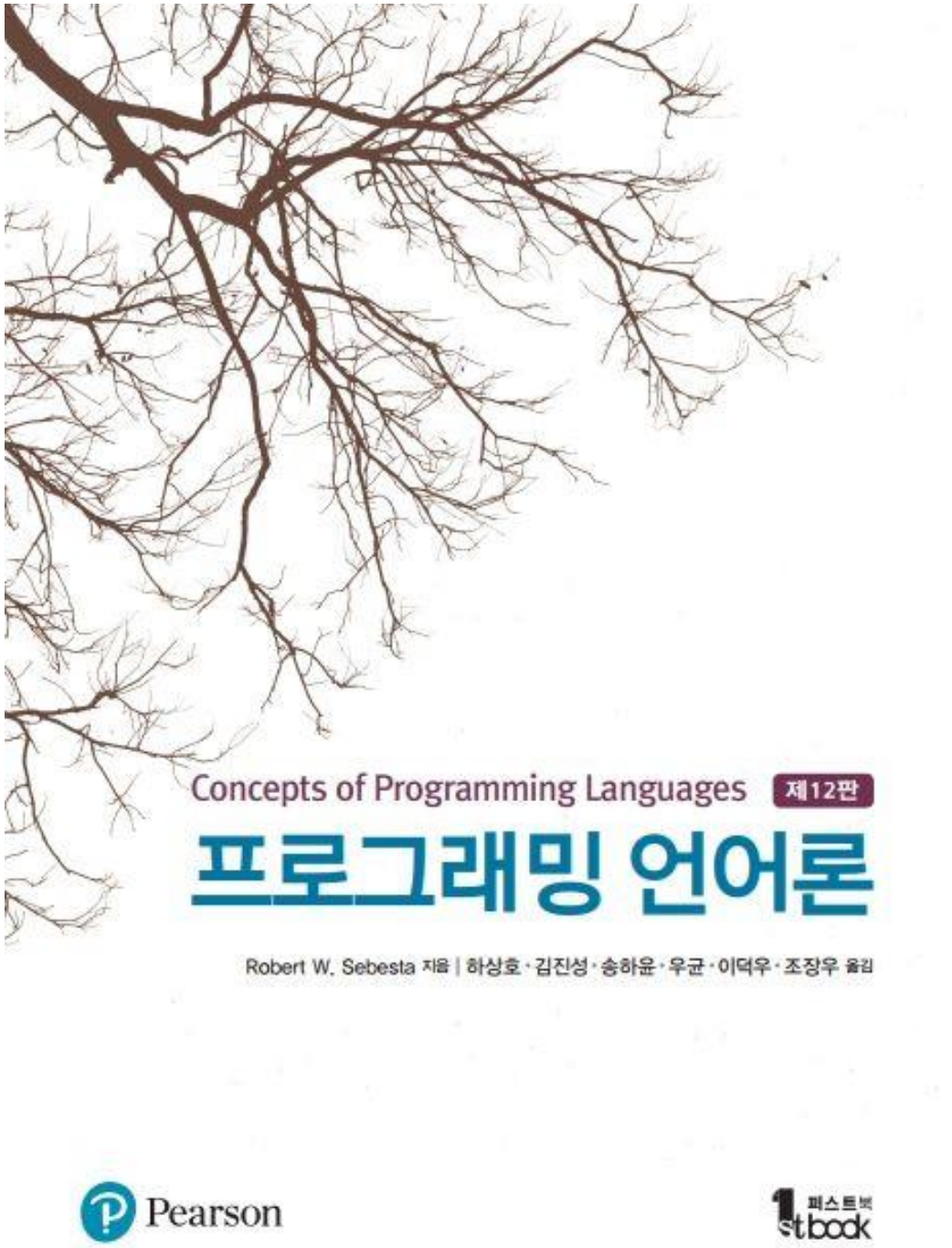
5장

이름, 바인딩, 영역


Concepts of Programming Languages
제12판
프로그래밍 언어론
Robert W. Sebesta 지음 | 하상호 · 김진성 · 송하윤 · 우균 · 이덕우 · 조장우 옮김
Pearson
1st book 퍼스트북

주제

- 서론
- 이름
- 변수
- 바인딩 개념
- 영역
- 영역과 존속기간
- 참조 환경
- 이름 상수

서론

- 명령형 언어는 폰노이만 구조의 추상화
  - 메모리: 명령어와 데이터의 저장
  - 프로세서: 메모리의 내용을 변경하기 위한 연산을 제공
- 변수
  - 명령형언어: 메모리 셀에 대한 추상화
  - 함수형언어?

이름

- 식별자(identifier) = 이름(name)
  - 프로그램의 개체들을 식별하기 위해 사용되는 문자열.
  - 프로그램의 개체: 변수, 함수, 형식 매개변수, 클래스, 등
- 설계 고려사항
  - 이름 형식
  - 특수어

이름 형식

- 길이
  - 너무 짧으면 그 의미를 포함시킬 수 없다
  - Ex:
    - FORTRAN 95: 최대 31
    - C99: 제한이 없으나 처음 63개 문자만 의미, 외부 이름은 최대 31자에 제한
    - C#, Ada, Java: 제한 없고, 이름에 포함된 모든 문자는 의미가 있음
    - C++: 제한 없음, 흔히 언어 구현자가 제한
- 형식:
  - 문자 다음에 문자, 숫자, 밑줄문자로 구성된 문자열
  - 낙타표기법, myStack

이름 형식(계속)

- 특수 문자
  - PHP: 모든 변수 이름은 '$'로 시작
  - Perl: 모든 변수 이름은 특수 문자($, @, %)로 시작되어 변수의 타입을 명세: scalar, array, hash
  - Ruby: @, @@ 는 사례변수와 클래스변수

이름 형식(계속)

- 대소문자 구분(case sensitivity)
  - Ex. Rose, ROSE, Rose
  - 판독성 저하:
  - 언어 예제
    - 주로 C 기반 언어
    - C에서는 변수 이름에 대문자를 포함하지 않는 관례
      - student_rec_type, … studentRecType
    - Java, C#에서는 미리 정의된 이름에 대소문자 구분
      - Ex: IndexOutOfBoundsException
      - 작성력 저하 (철자 기억)

이름 형식(계속)

- 특수어 (special words)는 판독성 향상
  - 의미가 미리 정의되어 있음
  - 특수어는 키워드이거나 예약어

- 키워드(keywords)
  - 어떤 문맥에서만 특별한 의미를 갖는 단어

    e.g. Fortran에서,

    real valrName // real은 키워드

    real = 3.4 // real은 변수 이름

- 예약어(reserved words)
  - 이름으로 사용될 수 없는 특수어
  - 대부분의 언어에서 특수어는 예약어임

변수

- 변수는 메모리 셀의 추상화
- 다음 6개의 속성들로 특징 지워짐
  - 이름(name)
  - 주소(address)
  - 값(value)
  - 타입(type)
  - 존속기간(lifetime)
  - 영역(scope)

주소

- 변수와 연관된 메모리 주소
- 변수의 주소는 때때로 l-value로 불림
  Ex. a = b + 1; //load, store 명령어

- 고려사항
  - 변수는 실행 중 다른 시기에 다른 주소를 가질 수 있는가?
  - 두 개의 변수 이름이 동일한 메모리 위치를 접근할 수 있는가?

- 용어:
  - 별칭(alias): 2개 이상의 변수 이름이 동일한 메모리 위치를 접근하는데 사용될 때, 그 변수들을 별칭이라 한다.
    - c 언어에서 두 개의 포인터 변수가 동일한 메모리를 가리키는경우

타입

- 변수의 값 범위와 그 타입의 값들에 대해서 정의된 연산들의 집합을 결정

  Ex. in C, int

값

- 변수와 연관된 위치에 저장된 내용
- 때때로 r-value라 불림

  Ex. a = b + 1; //load, store 명령어

- 추상 메모리 셀은 한 변수에 할당된 메모리 셀이나 셀들의 집합을 의미함
  - float 변수에는 4개의 바이트가 할당될 수 있지만 한 개의 추상 메모리 셀에 할당된 것으로 생각
  - 메모리 셀 용어는 추상 메모리 셀을 의미

바인딩의 개념

- 바인딩(binding)
  - 개체와 속성 간의 연관(association)

  Ex.
    - 변수와 타입
    - 변수와 값
    - 기호와 연산: +

- 바인딩 시기(binding time)
  - 바인딩이 일어나는 시기

바인딩 시기

- 언어 설계 시간
  - 연산자 기호와 연산 간의 바인딩
- 언어 구현 시간 (컴파일러 설계 시간)
  - 정수 타입과 그 크기 간의 바인딩 in C
- 컴파일 시간
  - 변수와 타입 간의 바인딩
- 링킹(linking) 시간
  - 라이브러리 함수 호출과 함수 코드와의 바인딩
- 적재(loading) 시간
  - static 변수와 메모리 셀의 바인딩
- 실행 시간
  - 비 static 지역 변수와 메모리 셀의 바인딩

예제

- 다음 코드에서 바인딩 시기는?

```
// In C

int count;
…
count = count + 5;
```

  - count의 타입
  - count의 가능한 값들의 집합
  - 연산자 기호 '+'의 의미
  - 리터럴 5의 내부 표현
  - count의 값

바인딩 유형

- 바인딩이 실행 전에 일어나고, 프로그램 실행 중에 변경되지 않으면, 그 바인딩은 정적(static)이다.

- 바인딩이 실행 중에 일어나고, 프로그램 실행 중에 변경될 수 있으면, 그 바인딩은 동적(dynamic)이다.

타입 바인딩

- 변수는 프로그램에서 참조될 수 있기 전에 어떤 데이터 타입에 바인딩되어야 한다.

- 타입 바인딩의 2가지 고려사항
  - 타입이 어떻게 명세되는가?
  - 바인딩이 언제 일어나는가?

- 타입 바인딩이 정적이면, 타입 명세는 명시적(explicit)이거나 묵시적(implicit)일 수 있다.

정적 타입 바인딩

- 명시적 선언(explicit declaration)은 선언문을 통해서 타입 명세
- 묵시적 선언(implicit declaration)은 선언문이 아닌 디폴트 관례를 통해서 타입 명세
  - 명칭 관례
    - Fortran에서 I, J, K, L, M, N으로 시작되는지의 여부에 따라서 타입이 묵시적으로 결정
    - 작성력 향상, 신뢰성 저해
  - 특정 타입의 이름을 특정 문자로 시작되게
    - Perl($: 스칼라 변수, @: 배열, %: 해시 구조체)

정적 타입 바인딩 (계속)

- 문맥을 통해서 묵시적으로 선언 => 타입 추론(type inference)
  - 변수는 선언문 초기화 시 할당된 값의 타입으로 명세

    Ex. In C#,

```
var sum = 0; // 반드시 초기값을 포함해야 함
var total = 0.0;
var name = “Fred”;
```

  - 문맥으로 타입 추론

    Ex. In ML, fun circumf(r) = 3.14159 * r * r;

    // 매개변수 r과 함수 값의 타입?

동적 타입 바인딩

- 변수 타입이 배정문에서 변수에 값이 할당될 때
  - 좌측 변수는 우측 식의 값의 타입으로 바인딩
  - 변수는 임의 타입의 값에 할당 가능
  - 변수는 배정문 실행으로 주소나 메모리 셀에 바인딩
  - 변수의 타입은 실행 중에 여러 번 변경 가능

    Ex. JavaScript, Python, PHP 등

Ex. // in JavaScript

```
list=[10.2, 4.33, 6, 8];
...
list = 17.3;
```

```
// in C#,
dynamic any; // any에 임의 타입
             // 값 할당 가능
any = 100;
any = "Hello";
```

동적 타입 바인딩 (계속)

- 장점:
  - 유연성(flexibility)
  - 포괄적 프로그램(generic program) 작성가능
    - 임의의 수치 타입을 처리 가능
    - 반면, C 언어는?
- 단점:
  - 실행 비용–동적 타입 검사, 실행 시 타입정보를 유지해야 함
  - 신뢰성 – 오류 탐지 능력 저하
- 동적 타입 바인딩 언어는 보통 순수 해석으로 구현

기억장소 바인딩과 존속기간

- 존속기간(lifetime)
  - 변수가 특정 메모리 셀에 바인딩되어 있는 기간

- 용어:
  - 기억장소 할당(allocation): 가용 메모리로부터 변수에 바인딩되는 메모리 셀 할당

  - 기억장소 회수(deallocation): 변수로부터 바인딩이 해제된 메모리 셀을 가용 메모리에 반환

변수의 유형

- 존속기간 기준 스칼라 변수의 4가지 유형
  - 정적
  - 스택-동적
  - 명시적 힙-동적
  - 묵시적 힙-동적

정적 변수

- 정적 변수(static variable)는 프로그램 실행 시작 전에 메모리 셀에 바인딩되고, 그 바인딩이 프로그램 실행이 종료될 때까지 유지
  - 장점:
    - 효율성(efficiency): 직접 주소지정
    - 과거 민감(history-sensitive) 부프로그램 지원
  - 단점
    - 유연성(flexibility) 감소: 재귀 함수 불가
    - 기억공간이 변수들 간에 공유 불가

Ex. C의 ?

스택-동적 변수 (stack-dynamic variable)

- 선언문이 세련화(elaboration)될 때 기억 공간에 바인딩되는 변수
  - 선언문 세련화: 선언문이 지시한 기억 공간 할당과 바인딩 과정
  - 다른 모든 속성은 정적으로 바인딩
  - 메모리 셀은 실행-시간 스택 상에 할당

  - 장점
    - 재귀적 부프로그램 지원: 재귀적 부프로그램의 활성화될 때마다, 지역변수의 자신의 버전을 가짐
    - 부프로그램간 메모리 공간 공유

  - 단점
    - 할당/회수에 따른 실행 부담
    - 간접 주소지정으로 dangling 포인터 발생가능
    - 과거 민감 부프로그램 지원 불가

명시적 힙-동적 변수
(explicit heap-dynamic variable)

- 프로그래머가 명세하는 실행시간 명령어에 의해서 할당되고 회수되는 이름 없는 메모리 셀
  - C 의 malloc(), free(), C++, Java의 new
  - 메모리 셀은 힙 상에 할당 및 회수
  - 포인터나 참조 변수를 통해서 참조

    Ex. Java의 객체, C++에서 new로 할당되는 객체
  - 컴파일 언어에서 변수의 타입 바인딩은 정적, 변수의 기억공간 바인딩은 동적
  - 장점: 동적 자료구조 생성
  - 단점: 비효율적, 비 신뢰적

묵시적 힙-동적 변수

- 묵시적 힙-동적 변수(implicit heap-dynamic variable)는 할당문에 의해서 할당과 반환이 이루어지는 변수
  - 메모리 셀은 힙 상에 할당
  - 변수의 모든 속성이 값이 할당될 때마다 바인딩

    Ex. // in JavaScript,
    list = [74, 84, 86, 90, 71];
    list = 3.5;

  - 장점: 유연성
  - 단점:
    - 비효율적
    - 컴파일러에 의한 오류 탐지 능력 상실

영역

- 변수의 영역(scope)은 변수가 가시적인 문장들의 범위
- 변수가 문장에서 참조될 수 있으면, 변수는 그 문장에서 가시적(visible)이라고 함
- 영역 규칙은 변수의 참조가 변수의 선언과 어떻게 연관되는지 결정
  - 정적 영역
  - 동적 영역
- 용어
  - 지역 변수(local variables)는 프로그램 단위 또는 블록 내부에 선언된 변수
  - 비 지역변수(nonlocal variables)는 프로그램 단위에서 가시적이나 그곳에 선언되지 않은 변수
  - 전역 변수(global variables)는 비 지역 변수이며, 부 프로그램 외부에 선언된 변수

정적 영역

- 정적 영역(static scope)은 프로그램 텍스트에 기반
  - 실행 전에 변수 영역이 결정
  - 판독자나 컴파일러는 변수의 타입 식별 가능

- 이름 참조를 변수와 연관하는 과정:
  - 먼저, 지역적으로 선언문을 찾고, 다음에, 그 이름에 대한 선언문이 찾아질 때까지, 현재 영역을 포함한 점차 더 큰 영역 순서로 선언문 탐색

- 용어:
  - 정적 조상(static ancestors) 현재 영역을 포함하는 더 큰 모든 영역
  - 정적 부모(static parents) 가장 가까운 정적 선조

정적 영역 (계속)

- 중첩된 부프로그램을 허용한 언어는 중첩 정적 영역 생성
  - Ex. Python, JavaScript, Pascal, Ada, Fortran 2003+ 등
- 그렇지 않은 언어에서 중첩된 클래스나 블록에 의해서 중첩된 영역 생성
  - Ex. C 기반 언어

정적 영역 (계속)

- 변수는 동일한 이름을 갖는 정적 선조 영역에 속한 변수로 인해서 프로그램 단위로부터 은폐 가능
- 은폐된 변수는 정적 선조 영역의 이름을 포함한 선택적 참조(selective reference)로 접근 가능

```
function big() {
    function sub1() {
        var x = 7;
        sub2();
    } // end of sub1()
    function sub2() {
        var y = x;
    } end of sub2()
```

```
    var x = 3;
    sub1();
} //end of big()
```

블록

- 블록(blocks)은 프로그램 단위 내부에서 정적 영역을 생성하는 방법 제공
  - ALGOL 60에서 도입
  - 블록-구조 언어(block-structured language)의 기원
  - 블록에 선언된 변수는 스택-동적
  - 영역이 최소화된 지역 변수 허용

```
If (list[i] < list[j]) {
  int tmp;
  tmp = list[i];
  list[i] = list[j];
  list[j] = tmp;
}
```

블록 (계속)

- 블록에서 더 큰 영역의 이름 재사용 여부?

```
void sub() {
 int count;
 while (...) {
      int count;
      count++;
      ...
 }
  …
}
```

  - sub의 count는 while 내부에서 은폐
  - C/C++에서 적법하나 Java/C#에서는 불가

LET 구조

- 대부분 함수형 언어는 블록과 관련된 let 구조 지원
- let 구조는 2개 부분으로 구성
  - 첫번째 부분은 이름들을 값에 바인딩
  - 두번째 부분은 첫번째 부분에서 정의된 이름들을 사용
  - 이름들의 영역은 LET에 지역적

Ex: in Scheme,

```
(LET (
   (name1 expression1)
          …
   (namen expressionn))
   expression
)
```

```
(LET (
   (top (+ a b))
   (bottom (- c d)))
   (/ top bottom) // (a+b)/(c-d)
)
```

선언 순서

- C99, C++, Java, C#는 변수 선언은 문장이 올 수 있는 곳이면 어디든지 올 수 있음
  - C99, C++, Java에서, 지역 변수의 영역은 선언 지점부터 블록 끝까지
  - C#에서, 블록에 선언된 변수 영역은 블록 전체(선언문의 위치에 관계없이 블록 전체)

```
// in C#
{ int x;
    …
    { int x; // illegal
          …
    }
    …
}
```

```
// in C#
{
    { int x; // illegal
          …
    }
    int x;
    …
}
```

선언 순서 (계속)

- C++, Java, C#에서, 변수는 for 문에서 선언 가능
  - 변수 영역은 for 문 내부에 제한

```
void fun() {
    …
    for (int count = 0; count < 10; count++) {
        …
    }
    …
}
```

전역 영역

- C, C++, PHP, Python은 한 파일에 여러 함수 정의들을 포함하는 프로그램 구조
  - 이러한 언어는 함수 외부에 전역 변수 선언

- C, C++는 전역 데이터에 대해서 선언(속성만)과 정의(속성과 기억공간)를 구분
  - 함수 외부의 변수 선언은 그 변수가 다른 파일에 정의되어 있음을 명세

```
extern int sum;
```

전역 영역 (계속)

- PHP의 전역변수는 함수에서 묵시적으로 비 가시적
  - 전역 변수는 global 선언을 통해서 가시적
  - 은폐된 전역 변수는 $GLOBAL 배열을 통해서 접근 가능

```
$day = “Monday”;
$month = “January”;

function calendar(){
  $day = “Tuesday”;
  global $month;

  print “local day is $day <br />”;
  $gday = $GLOBALS[‘day’];
  print “global day is $gday <br />”;
  print “global month is $month <br />”;
}
```

전역 영역 (계속)

- Python의 전역 변수는 함수에서 참조될 수 있으나, global로 선언된 경우에만 할당 가능

```
day = “Monday”
def tester():
    print (“The global day is:”, day)
```

```
day = “Monday”
def tester():
    print (“The global day is:”, day)
    day = “Tuesday”
    print (“The new value of day is:”, day)
```

UnbounfLocalError

```
day = “Monday”
def tester():
    global day
    print (“The global day is:”, day)
    day = “Tuesday”
    print (“The new value of day is:”, day)
```

정적 영역 평가

- 많은 상황에서 잘 동작한다.

- 문제점
  - 대부분의 경우에, 변수와 함수에 대해서 필요 이상의 너무 많은 접근을 허용
  - 프로그램이 수정되면서, 변수와 부프로그램 접근을 제한하는 그 초기 구조는 사라지고, 지역변수와 부프로그램들이 전역화되어 가는 경향

동적 영역 (5/1)

- 동적 영역(dynamic scope)은 부프로그램의 호출 시퀀스에 기반하여 실행 시간에 결정

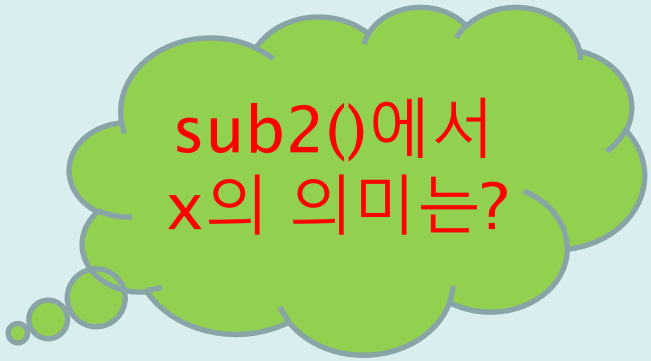
  Ex. APL, SNOBOL4, LISP 초기 버전

  - Perl, Common LISP은 디폴트가 정적이나 동적 영역도 허용

- 변수에 대한 참조는 현 지점의 실행에 이르게 한 부프로그램의 호출 체인의 역순으로 선언문 탐색

예제: 동적 영역

```
function big() {
    function sub1() {
        var x = 7;
            … sub2()
    }
    function sub2() {
        var y = x;
        var z = 3
    }
    var x = 3;
    … sub1()
    … sub2()
}
```



- 다음 호출 시퀀스를 고려하라.
  - big => sub1 => sub2
  - big => sub2

동적 영역 평가

- 장점: 편의성(convenience)

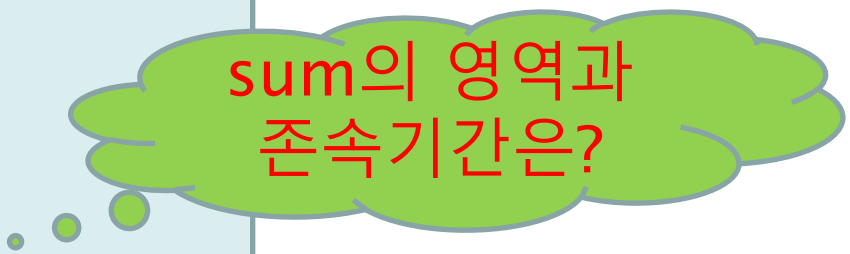
- 단점
  - 비지역변수의 속성을 정적으로 결정할 수 없음
  - 변수들은 모든 호출된 부프로그램에 가시적
  - 비 지역변수에 대한 접근 비 효율적

영역과 존속기간

- 공간적 개념 vs. 시간적 개념

```
void printheader() {
  …
}

void compute() {
  int sum;
  …
   printheader();
}
```



  - C/의 함수에 선언된 static 변수의 영역과 존속기간은?

참조 환경

- 어떤 문장의 참조 환경(referencing environment)은 그 문장에서 가시적인 모든 이름들의 집합
- 정적 영역에서 참조 환경은 지역 영역에 선언된 변수와 조상 영역에 속한 가시적인 모든 변수들로 구성
- 동적 영역에서 참조 환경은 지역 변수와 현재 활성화된 모든 부 프로그램들에게 속한 모든 변수들로 구성
- 용어:
  - 활성화된 부 프로그램 (active subprogram)은 실행이 시작되었으나 아직 종료되지 않은 상태의 부 프로그램

```
g = 3; # 전역 변수
def sub1():
    a = 5; # 지역 변수를 생성한다
    b = 7; # 다른 지역 변수를 생성한다
    ... <---------------------- 1
def sub2():
    global g; # 전역 변수 g는 이제 여기서 배정 가능하다
    c = 9; # 새로운 지역 변수를 생성한다
    ... <---------------------- 2
    def sub3():
        nonlocal c: # 비지역 변수 c를 여기서 가시적이게 한다
        g = 11; # 새로운 지역 변수를 생성한다
•         ... <-------------------- 3
```

지점	참조환경
1	sub1의 a와 b, 참조를 위한 전역 변수 g, 배정은 안됨
2	sub2의 c, 참조와 배정을 위한 전역변수 g
3	sub2의 c, sub3의 g

예제

- 다음 Python 코드 실행 결과는?

```
g=10
def myfunc1():
  x = "John"
  print(g)

  def myfunc2():
    g = 20
    x = "hello"
    print(x)
    print(g)

  myfunc2()
  print(x)
  print(g)

myfunc1()
```

```
g=10
def myfunc1():
  x = "John"
  print(g)

  def myfunc2():
    nonlocal x
    global g
    g = 20
    x = "hello"
    print(x)
    print(g)

  myfunc2()
  print(x)
  print(g)

myfunc1()
```

```
void sub1(void) { // 동적 영역 고려
  int a, b;
   …  ←---------- 1
}

void sub2(void) {
  int b, c;
  … ←---------- 2
  sub1();
}

void main()
{
  int c, d;
  … ←-----------3
  sub2();
}
```

지점	참조환경
1	sub1의 a와b, sub2의 c, main의 d (main의 c와 sub2의 b는 은폐된다.)
2	sub2의 b와c, main의 d (main의 c는 은폐된다.)
3	main의 c와 d

이름 상수

- 이름 상수(named constant)는 단지 한번만 값에 바인딩 되는 변수
  - 판독성, 신뢰성 향상
  - 프로그램 매개변수화 가능

```
pi = 3.14159265
```

```
void example() {
  int[] intList = new int[100];
  string[] strList = new String[100]
  …
  for (I =0; i<100; i++) {
          …
  }
…
```

```
void example() {
  final int len = 100;
  int[] intList = new int[len];
  string[] strList = new String[len]
  …
  for (I =0; i<len; i++) {
          …
  }
…
```

이름 상수 (계속)

- 이름 상수에 바인딩되는 값은 정적이거나 동적일 수 있다.

```
const int result = 2 * width + 1; // in C++
                                  // 동적 바인딩 허용
final ...          // in Java
                   // 동적 바인딩 허용

const ...          // in C#: 정적 바인딩
readonly...        // in C#: 동적 바인딩
```