INF 212
ANALYSIS OF PROG. LANGS
ELEMENTS OF IMPERATIVE PROGRAMMING STYLE

Instructors: Crista Lopes

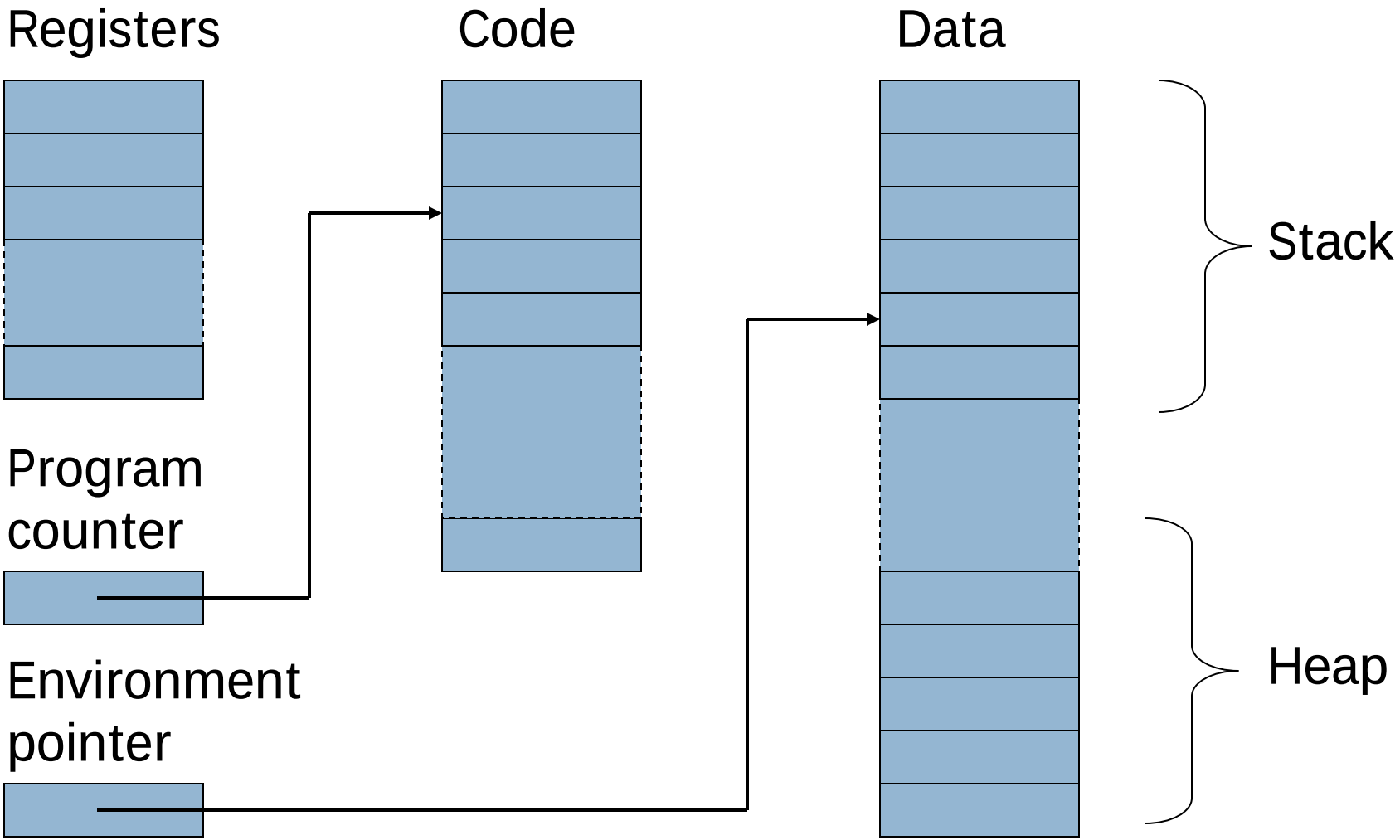
Objectives

- Level up on things that you may already know…
  - Machine model of imperative programs
  - Structured vs. unstructured control flow
  - Assignment
  - Variables and names
  - Lexical scope and blocks
  - Expressions and statements
- …so to understand existing languages better

Imperative Programming

- Oldest and most popular paradigm
  - Fortran, Algol, C, Java …
- Mirrors computer architecture
  - In a von Neumann machine, memory holds instructions and data
- Control-flow statements
  - Conditional and unconditional (GO TO) branches, loops
- Key operation: assignment
  - Side effect: updating state (i.e., memory) of the machine

Simplified Machine Model



Memory Management

- Registers, Code segment, Program counter
  - Ignore registers (for our purposes) and details of instruction set
- Data segment
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - Environment pointer points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record

Control Flow

- Control flow in imperative languages is most often designed to be sequential
  - Instructions executed in order they are written
  - Some also support concurrent execution (Java)

- But…

Goto in C

```
# include <stdio.h>
int main(){
   float num,average,sum;
   int i,n;
   printf("Maximum no. of inputs: ");
   scanf("%d",&n);
   for(i=1;i<=n;++i){
       printf("Enter n%d: ",i);
       scanf("%f",&num);
       if(num<0.0)
         goto jump;
       sum=sum+num;
   }
jump:
  average=sum/(i-1);
  printf("Average: %.2f",average);
  return 0;
}
```

Before C: Goto in Fortran

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT, ONE BLANK CARD FOR END-OF-DAT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPAY ERROR MESSAGE ON OUTPUT
  501 FORMAT(3I5)
  601 FORMAT(4H A= ,I5,5H  B= ,I5,5H C= ,I5,8H AREA= ,F10.2,12HSQUARE UNIT
  602 FORMAT(10HNORMAL END)
  603 FORMAT(23HINPUT ERROR, ZERO VALUE)
      INTEGER A,B,C
   10 READ(5,501) A,B,C
      IF(A.EQ.0 .AND. B.EQ.0 .AND. C.EQ.0) GO TO 50
      IF(A.EQ.0 .OR.  B.EQ.0 .OR.  C.EQ.0) GO TO 90
      S = (A + B + C) / 2.0
      AREA = SQRT( S * (S - A) * (S - B) * (S - C))
      WRITE(6,601) A,B,C,AREA
      GO TO 10
   50 WRITE(6,602)
      STOP
   90 WRITE(6,603)
      STOP
      END
```

Structured Control Flow

- Program is structured if control flow is evident from syntactic (static) structure of program text
  - Hope: programmers can reason about dynamic execution of a program by just analysing program text
  - Eliminate complexity by creating language constructs for common control-flow patterns
    - Iteration, selection, procedures/functions

Historical Debate

- Dijkstra, "GO TO Statement Considered Harmful"
  - Letter to Editor, Comm. ACM, March 1968
  - Linked from the course website
- Knuth, "Structured Prog. with Go To Statements"
  - You can use goto, but do so in structured way …
- Continued discussion
  - Welch, "GOTO (Considered Harmful)^n, n is Odd"
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Structured Programming

- Standard constructs that structure jumps

  if … then … else … end

  while … do … end

  for … { … }

  case …

- Group code in logical blocks
- Avoid explicit jumps (except function return)
- Cannot jump into the middle of a block or function body

Cyclomatic Complexity

- A metric to measure the amount of control flow paths in a block of code

$$CC = E - N + 2P$$

where
E = number of edges
N = number of nodes
P = number of exit nodes

Less is better

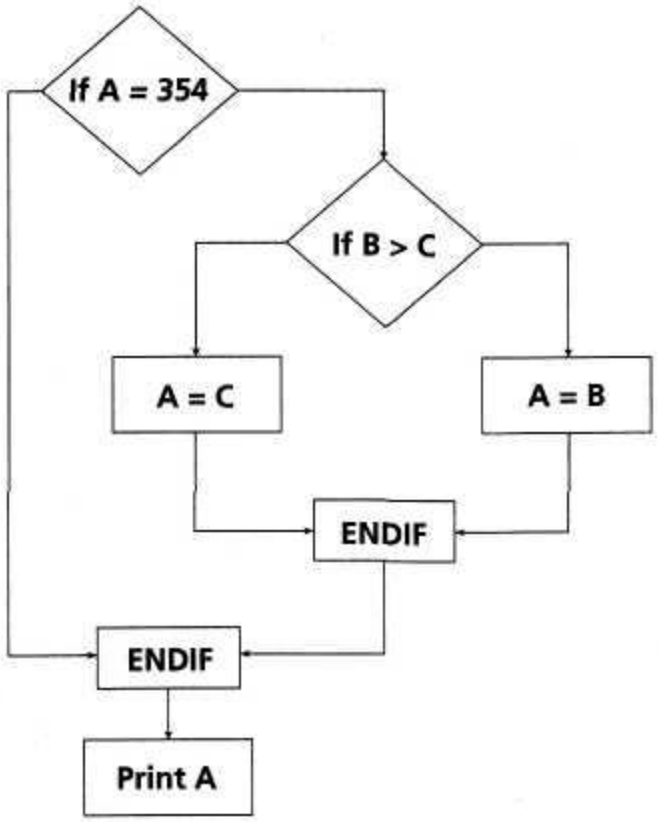
Cyclomatic Complexity

- Rule of thumb:
  - CC < 10 : ok
  - 10 < CC < 20 : moderate risk
  - 20 < CC < 50 : high risk
  - CC > 50 : extremely high risk

Less is better

CC example

```
IF A = 354 THEN
  IF B > C THEN
    A = B
  ELSE
    A= C
  ENDIF
ENDIF
Print A
```



CC = 8 – 7 + 2*1 = 3

Another example

```
insertion_procedure (int a[], int p [], int N)
{
    int i,j,k;
    for (i=0; i<=N; i++)
        p[i] = i;
    for (i=2; i<=N; i++) {
        k = p[i];
        j = 1;
        while (a[p[j-1]] > a[k]) {
            p[j] = p[j-1];
            j--;
        }
        p[j] = k;
    }
}
```

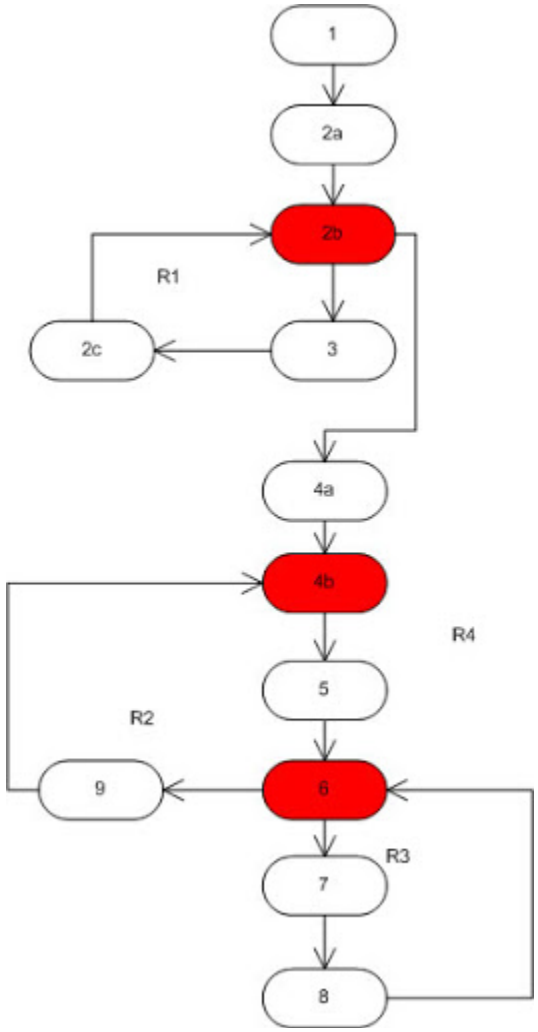
Source: stackoverflow

Another example

```
insertion_procedure (int a[], int p [], int N)
 {
(1)     int i,j,k;
(2)     for ((2a)i=0; (2b)i<=N; (2c)i++)
(3)         p[i] = i;
(4)     for ((4a)i=2; (4b)i<=N; (4c)i++)
        {
(5)         k=p[i];j=1;
(6)         while (a[p[j-1]] > a[k]) {
(7)             p[j] = p[j-1];
(8)             j--
            }
(9)         p[j] = k;
        }
 }
```

Source: stackoverflow

Another example



CC = 4

Source: stackoverflow

Assignment (you thought you knew)

```
x = 3
x = y+1
x = x+1
```

Informal:

"Set x to 3"
"Set x to the value of y plus 1"
"Add 1 to x"

Let's look at some other examples

Assignment (you thought you knew)

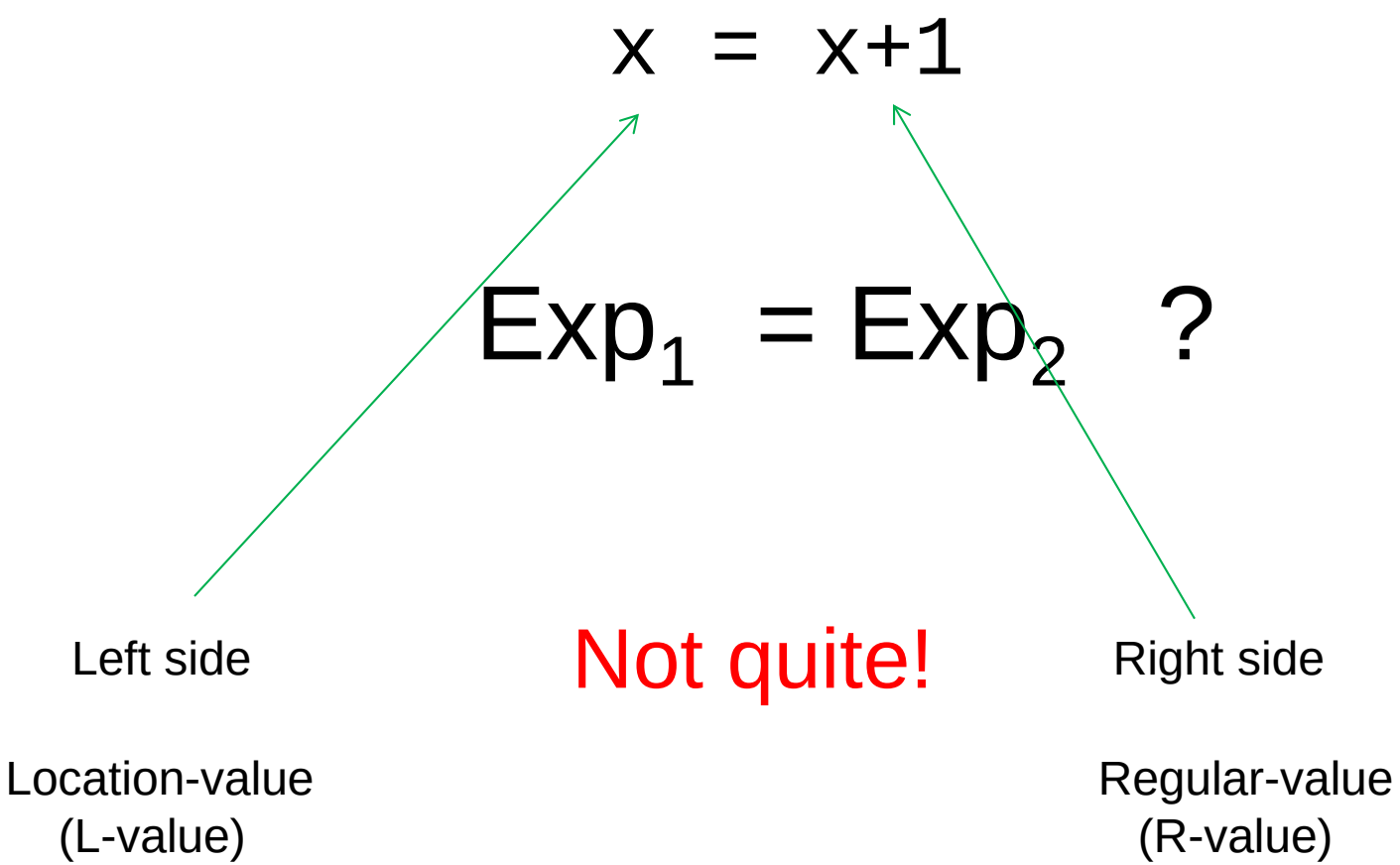
```
                    i = (a>b) ? j : k
                m[i] = m[(a>b)? j : k]
m[(a>b) ? j : k] = m[i]
```

$$Exp_1 = Exp_2 \ ?$$

Assume x is 5 x = x+1 means 5 = 6 ????

What ***exactly*** does assignment mean?

Assignment (you thought you knew)



Assignment

- On the RHS of an assignment, use the variable's R-value; on the LHS, use its L-value
  - Example: x = x+1
  - Meaning: "get R-value of x, add 1, store the result into the L-value of x"
- An expression that does not have an L-value cannot appear on the LHS of an assignment
  - What expressions don't have l-values?
    - Examples: 1=x+1, x++ (why?)
    - What about a[1] = x+1, where a is an array? Why?

Locations and Values

- When a name is used, it is bound to some memory location and becomes its identifier
  - Location could be in global, heap, or stack storage
- L-value: memory location (address)
- R-value: value stored at the memory location identified by l-value
- Assignment: A (target) = B (expression)
  - Destructive update: overwrites the memory location identified by A with a value of expression B
    - What if a variable appears on both sides of assignment?

l-Values and r-Values (1)

- Any expression or assignment statement in an imperative language can be understood in terms of l-values and r-values of variables involved
  - In C, also helps with complex pointer dereferencing and pointer arithmetic
- Literal constants
  - Have r-values, but not l-values
- Variables
  - Have both r-values and l-values
  - Example: x=x*y means "compute rval(x)*rval(y) and store it in lval(x)"

l-Values and r-Values (2)

- Pointer variables
  - Their r-values are l-values of another variable
    - Intuition: the value of a pointer is an address
- Overriding r-value and l-value computation in C
  - &x always returns l-value of x
  - *p always return r-value of p
    - If p is a pointer, this is an l-value of another variable

```
int x = 5;  // lval(x) is some (stack) address, rval(x) == 5
int *p = &x // rval(p) == lval(x)
*p = 2 * x; // rval(p) <- rval(2) * rval(x)
```

What are the values of p and x at this point?

Copy vs. Reference Semantics

- Copy semantics: expression is evaluated to a value, which is copied to the target
  - Used by imperative languages
- Reference semantics: expression is evaluated to an object, whose pointer is copied to the target
  - Used by object-oriented languages

Copy vs. Reference Semantics

In Java/C/C++:

```
x = 1;
x = 3;
```

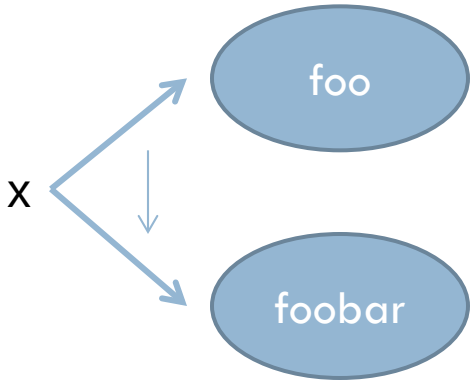
Copy semantics

Overwrites the r-value of x from int 1 to int 3

In Java/C++/Python/Ruby:

```
x = new Foo;
x = new FooBar;
```

Reference semantics

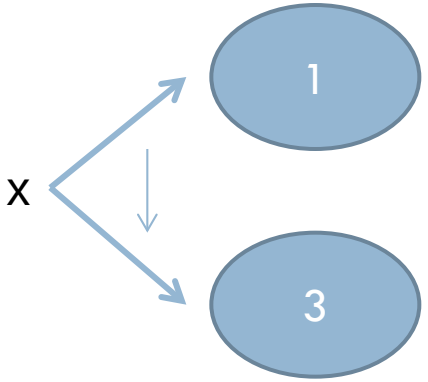


Overwrites the r-value of x too, but that value is a “pointer”

In Python/Ruby:

```
x = 1;
x = 3;
```

Reference semantics



Overwrites the r-value of x too, but that value is a “pointer”

l-Values and r-Values (3)

- Declared functions and procedures
  - Have l-values, but no r-values

```
int f(int y); // lval(f) is some global address
typedef int (*IFP)(int); // pointer to an int function that takes an int argument
IFP g = &f; // lval(g) <- lval(f)
(*g)(5);    // (rval(g))== lval(f), so *g invokes f with argument rval(5)
            // the function call operator () has higher precedence than * so
            // we have to write (*g)(5) to deference g to invoke f(5)
```

Typed Variable Declarations

- Typed variable declarations restrict the values that a variable may assume during program execution
  - Built-in types (int, char …) or user-defined
  - Initialization: Java integers to 0. What about C?
- Variable size
  - How much space needed to hold values of this variable?
    - C on a 32-bit machine: sizeof(char) = 1 byte, sizeof(short) = 2 bytes, sizeof(int) = 4 bytes, sizeof(char*) = 4 bytes (why?)
    - What about this user-defined datatype:

```
typedef struct TreeNode {
        int x;
        TreeNode *front, *back;
};
```

Variables without declarations (names)

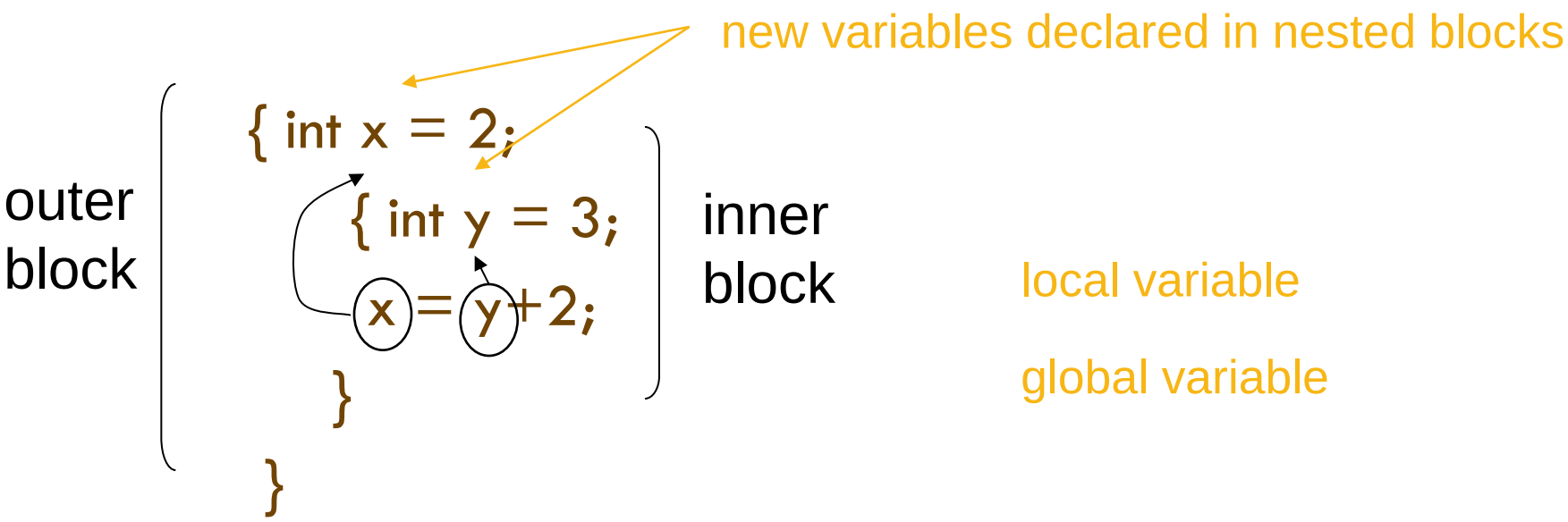
- Names that bind to values
- **Names don't have types; values do**

- Python, Perl, Ruby, …

```
x = 1
x = "hello"
```

Block-Structured Languages

- Nested blocks with local variables



  - Storage management
    - Enter block: allocate space for variables
    - Exit block: some or all space may be deallocated

Blocks in Common Languages

- Examples
  - C, ~~JavaScript~~ * { … }
  - Algol begin … end
  - ML let … in … end
- Two forms of blocks
  - Inline blocks
  - Blocks associated with functions or procedures
    - We'll talk about these later

* JavaScript functions provides blocks

Scope and Lifetime

- Scope
  - Region of program text where declaration is visible
- Lifetime
  - Period of time when location is allocated to program

```
{ int x = ... ;
    {  int y = ... ;
        {  int x = ... ;
           ....
         };
    };
};
```

Inner declaration of x hides outer one
("hole in scope")
Lifetime of outer x includes time when
inner block is executed
Lifetime ≠ scope

Inline Blocks

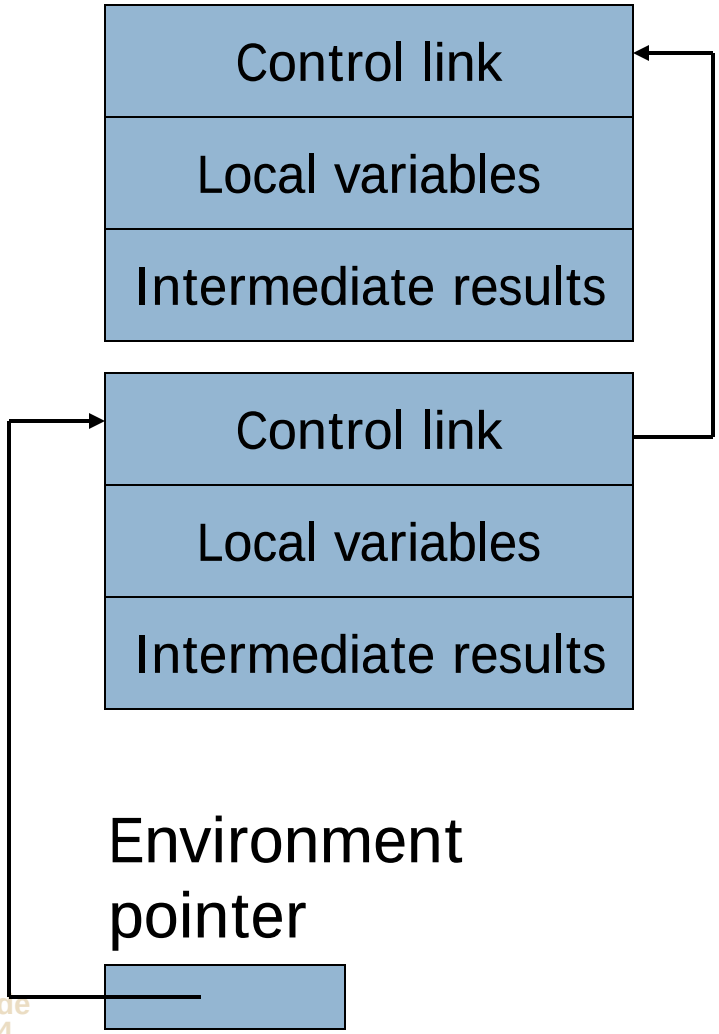
- Activation record
  - Data structure stored on run-time stack
  - Contains space for local variables

```
{ int x=0;
    int y=x+1;
        {   int z=(x+y)*(x-y);
        };
};
```

```
Push record with space for x, y
Set values of x, y
        Push record for inner block
        Set value of z
        Pop record for inner block
Pop record for outer block
```

May need space for variables and intermediate results like (x+y), (x-y)

Activation Record For Inline Block



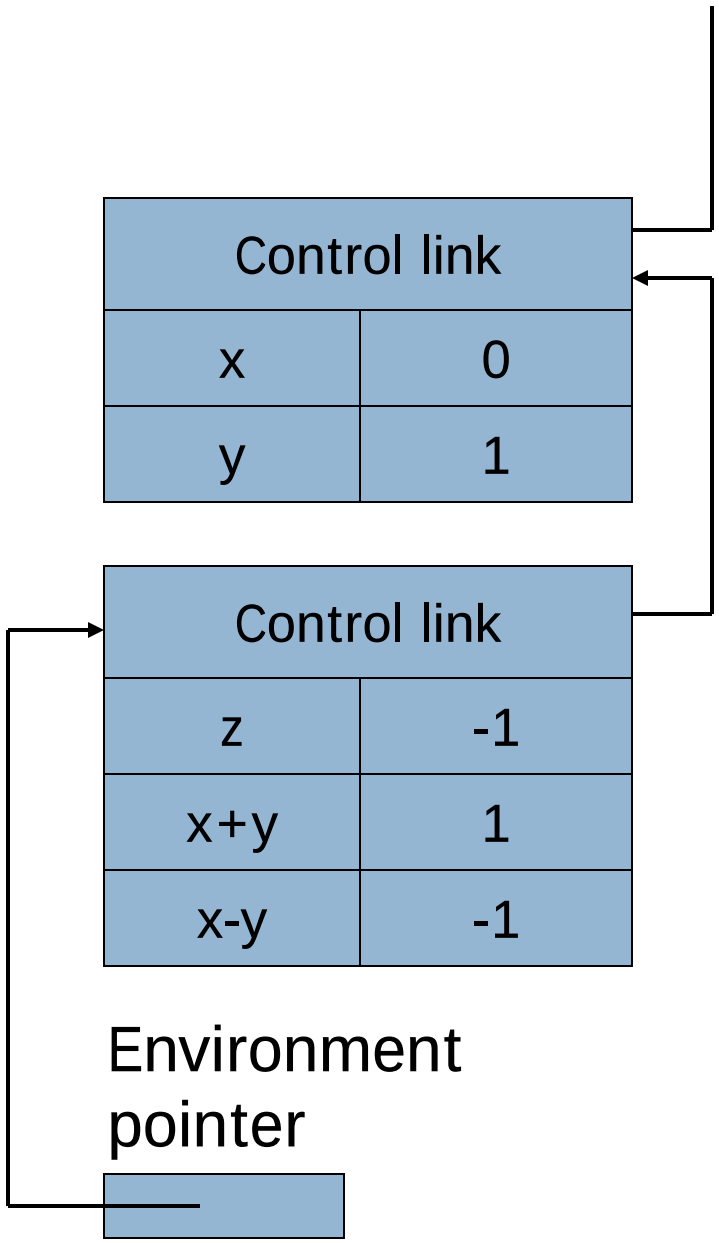
- Control link
  - Pointer to previous record on stack
- Push record on stack
  - Set new control link to point to old env ptr
  - Set env ptr to new record
- Pop record off stack
  - Follow control link of current record to reset environment pointer

In practice, can be optimized away

Example

```
{ int x=0;
   int y=x+1;
      {  int z=(x+y)*(x-y);
      };
};
```

Push record with space for x, y
Set values of x, y
    Push record for inner block
    Set value of z
    Pop record for inner block
Pop record for outer block



Expressions vs. Statements

- Expressions: mathematical expressions
  - x
  - a*(b+c)+d
  - No **side effects**
  - Evaluate to a value (pleonasm!)
- Statements (or commands)
  - x = *expr*
  - writeline(f, line)
  - Affect/interact with the world (**side effects**)
  - *Executed* rather than *evaluated*

Expressions vs. Statements

- print x ?
- [1, 2, 3] + [4, 5, 6] ?
- x = [1, 2, 3] ?
- readline() ?
- raise e ?