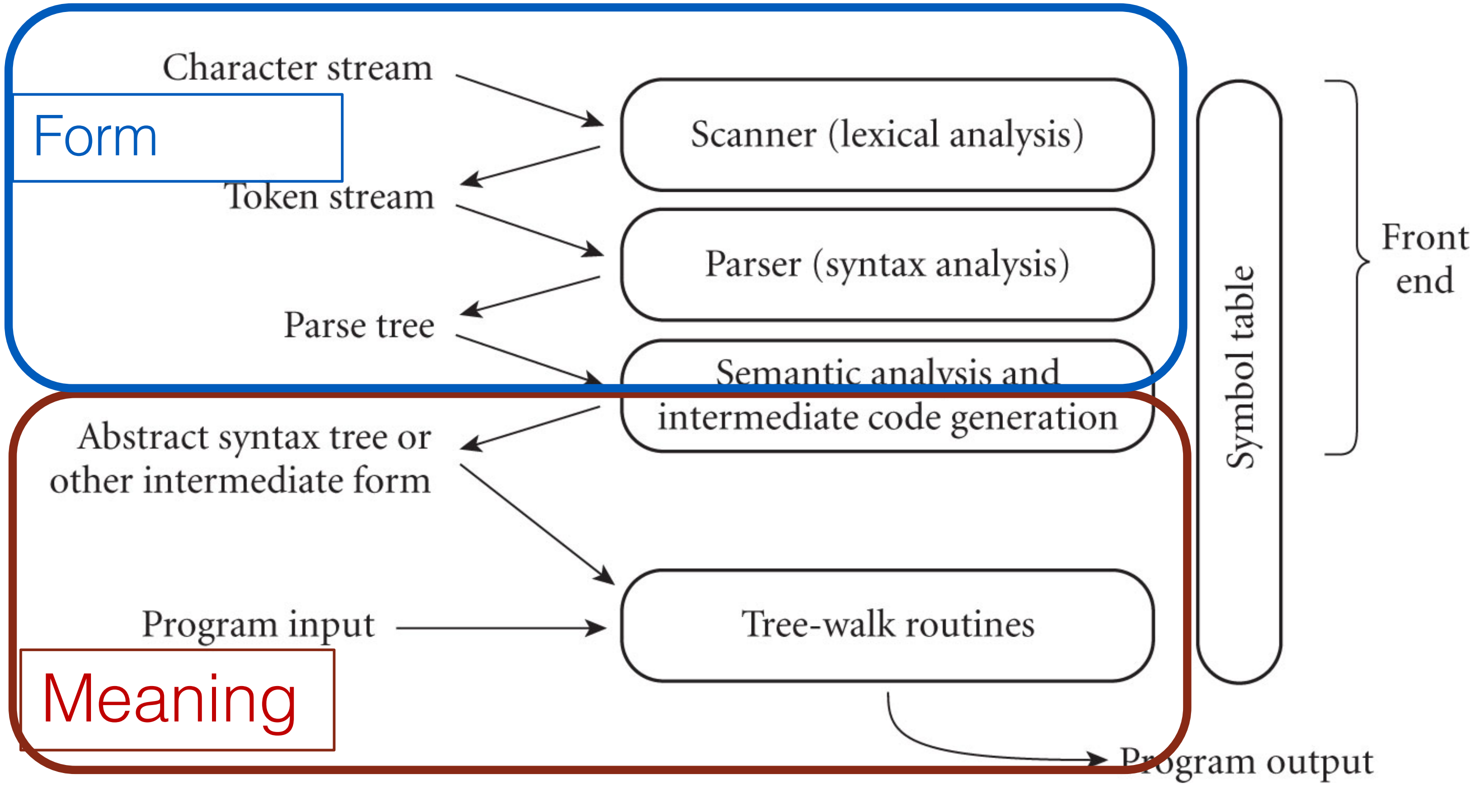
CS 320 : Scope and functions

Marco Gaboardi
MSC 116
gaboardi@bu.edu

Form and Meaning



Operational Semantics

- It is usually provided at a level of abstraction that is independent from the machine.
- The detailed characteristics of the particular computer would make actions difficult to describe/understand.
- Different formalism has been developed to describe the operational semantics in a machine-independent way.

We will look into formal rules and derivations.

Variables

- Functional languages use variables as names (where the association name-value is stored in an environment).
  - We can remember the association, or read the value, but we cannot change it.
- Imperative languages are abstractions of von Neumann architecture
  - A variable abstracts the concept of memory location
- Understanding how variables are managed is an important part to understand the semantics of a programming language.

Operational semantics for arithmetical expressions

$$(e/m) \longrightarrow (e/m)$$

Here (e/m) is a configuration where e is an expression and m is a memory. We call these pairs configurations because we think in terms of an "abstract machine".

We can think about a memory as a set of (unique) assignments of variables to values:

$$m = ((x_1=v_1), (x_2=v_2) \ldots, (x_n=v_n))$$

Extending an environment

What happens if `m` already contains `u`?

Suppose that we have

```
m=((x=1),(z=5),(y=3))
```

Then, if we extend m with the new pair (u=4), in symbols

```
m@(u=4)
```

We get:

```
m@(u=4)=((x=1),(z=5),(y=3),(u=4))
```

Updating an environment

What happens if `m` does not contain `x`?

Suppose that we have

```
m=((x=1),(z=5),(y=3))
```

Then, if we update m with the following command

```
update(x,4,m)
```

We get:

```
update(x,4,m)=((x=4),(z=5),(y=3))
```

Mutable vs Immutable Variables

- When we consider variables as names we are working with immutable variables (e.g. the part of OCaml we studied)
- When we consider variables as memory locations we are working with mutable variables (e.g. Python, c, etc.)
- Understanding how variables are managed is an important part to understand the semantics of a programming language.

Tips for interpreter part2: Operational semantics for the interpreter with variables

$$(p/S, m) \rightarrow (p'/S', m')$$

Here $(p/S, m)$ is a configuration where p is a program and S is a stack, and m is an environment.

We can think about the stack as a list of values:

$$v_n :: \ldots :: v_2 :: v_1 :: []$$

We can think about an environment as a set of (unique) assignments of variables to values:

$$m = ((x_1 = v_1), (x_2 = v_2) \ldots, (x_n = v_n))$$

Tip for interpreter part2: Implementation of OCaml `let`

We could imagine the let construction we saw in OCaml and in the last class:

```
let x=v in …
```

to be implemented as

```
push v
push x
bind
```

…

Tip for interpreter part2: rea variables

We could imagine the let construction we saw in OCaml and in the last class:

```
let x=v in … x …
```

to be implemented as

```
Push v
Push x
 Bind
  …
Push x
Lookup
```

The placement of v and x on the stack can happen at distance

Scoping rules

Variable names

How shall we evaluate this expression?

Is it referring to this definition?

```
let k=3 in (let k=2 in k+5)
```

Is it referring to this definition?

What is the value of k here?

Scope of a variable

- The scope of a variable is the range of statements over which it is visible
- The scope rules of a language determine how references to names are associated with variables

```
let k=3 in (let k=2 in k+5)
```

OCaml scoping rule says that a variable name is statically associated with the closest definition in the abstract syntax tree.

Back to our example

This is the first declaration we find

Start from here

```
let k=3 in (let k=2 in k+5)
```

To find the value of k we look search declarations, first locally, then in increasingly larger enclosing scopes

Another example

This is the first declaration we find

Start from here

```
let k=3 in (let z=2 in k+5)
```

To find the value of k we look search declarations, first locally, then in increasingly larger enclosing scopes

Another example

This is the first declaration we find

Start from here

```
let k=3 in k + (let k=2 in k+5)
```

This is the first declaration we find

Start from here

Another example

This is the first declaration we find

Start from here

```
let k=3 in (let k=2 in k+5) + k
```

This is the first declaration we find

Start from here

Static Scope

- Based on program text
- To connect a name reference to a variable, we (or the compiler) must find the declaration
- Some languages allow nested subprogram definitions, which create nested static scopes
- Search process:
  search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

Static Scope

- Search process:
  search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

How do we associate scopes here?

```
let x=3 in (let x=4 in x + 2) + x
```

How do we associate scopes here?

```
let x=3 in
 x + (let x=4 in
 x + (let x=3 in
 x + (let x= 6 in
 x + 2) +
 x) +
 x + 1)
```

Scope Blocks

A method of creating static scopes inside program units (ALGOL 60)

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

Program constructs (“blocks”) create scopes

Dynamic Scope

- Based on calling sequences of program units, not their textual layout,
- You can think about it more as temporal rather than spatial,
- References to variables are connected to declarations by searching back through the chain of subprogram calls that brought execution to this point.

Dynamic Scope Example

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

big calls sub1
sub1 calls sub2
sub2 uses x

- Static scoping -- Ref to x in sub2 is to big's x
- Dynamic scoping-- Ref to x in sub2 is to sub1's x

Dynamic Scope Example in bash

```
x=1
function g () { echo $x ; x=2 ; }
function f () { x=3 ; g ; }
f # does this print 1, or 3?
g # does this print 1, 2 or 3?
echo $x # does this print 1,2 or 3?
```

`echo $x` corresponds to printing the value of the variable `x`.

What does this program print?

Another Example in bash

```
x=1
function h () { echo $x ; x=2 ; }
function g () { echo $x ; x=3 ; h; }
function f () { x=4 ; echo $x; g ; }
f # What does this print?
g # What does this print?
h # What does this print?
echo $x # What does this print?
```

What does this program print?

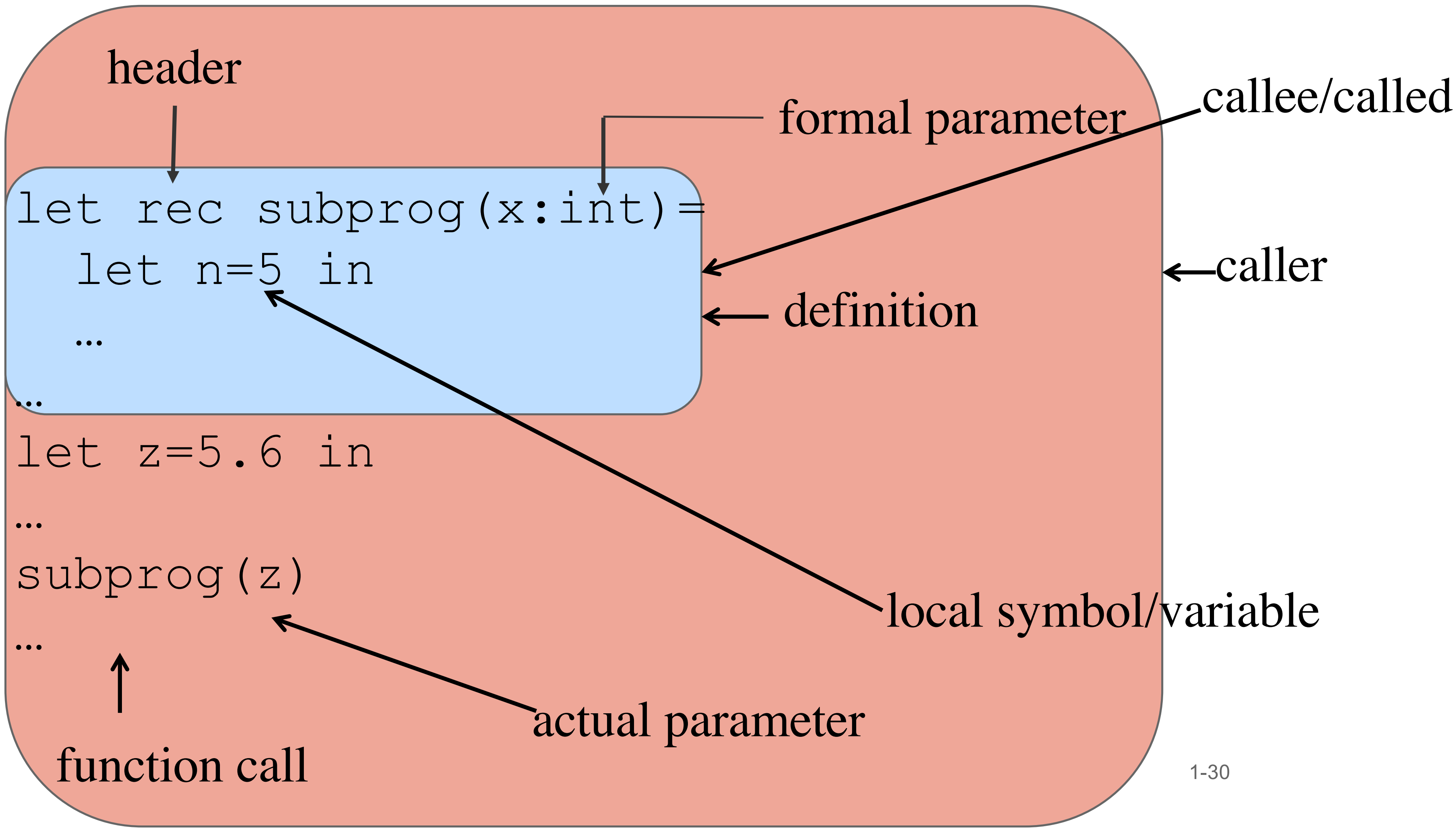
Another Example in bash

```
x=1
function h () { echo $x ; x=2 ;}
function g () { echo $x ; h; x=3 ;}
function f () { x=4 ; g ; echo $x; }
f
g
h
echo $x
```

What does this program print?

Functions

Terminology



Tip for interpreter part2: Language for basic stack manipulations with local variables definitions and functions

```
…
<com>    ::= Push <const> | Add | Sub | Mul | Div
           | Bind | Lookup |…
           | Fun <coms> End |Call|Return
```

Tip for interpreter part2:
Language for basic stack manipulations
with local variables definitions and functions

Instructions of the function

The name of the function is the top element of the stack

This is a function definition, when executed it will create a closure on the stack.

`fun <coms> End`

The formal parameter is not in the signature

Keyword to mark where a function declaration begins.

Keyword to mark where a function declaration ends.

Tip for interpreter part2:
Language for basic stack manipulations with local variables definitions and functions

Call will create a closure for the current continuation.

`Call`

The actual parameter is the top element in the stack

Keyword to mark when a function needs to be called.

Tip for interpreter part2: Language for basic stack manipulations with local variables definitions and functions

```
Push factorial;
Fun
  Push n;
  Bind;

  Push 2;
  Push n;
  Lookup;
  Gt;
```

formal parameter

What are the design considerations for functions?

We need to think about:

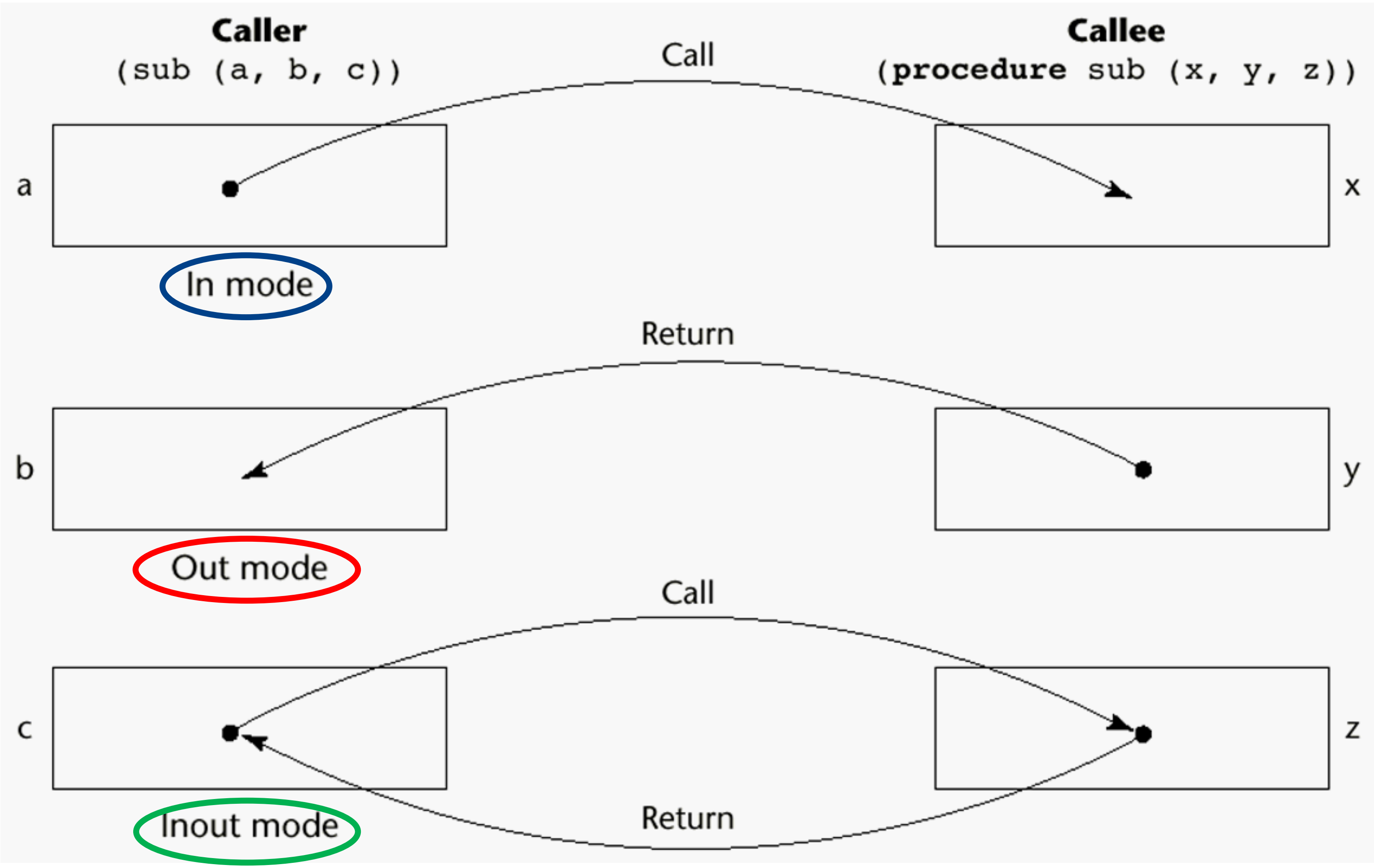
- parameter passing
- parameters returning
- variables: local vs global
- scope of variables
- nesting of subprograms
- referencing environment

Parameter Passing

Parameter passing methods are ways in which parameters are transmitted to and from sub programs.

- Semantic Models of Parameter Passing
- Implementation Models for these semantic models

Semantic Modes of Parameter Passing



How to transfer a value

- We have different ways to provide access to a value to a subprogram
  - Physically move a value
  - An access path is transmitted (e.g. pointer or reference)
- These are orthogonal to the mode of the parameters

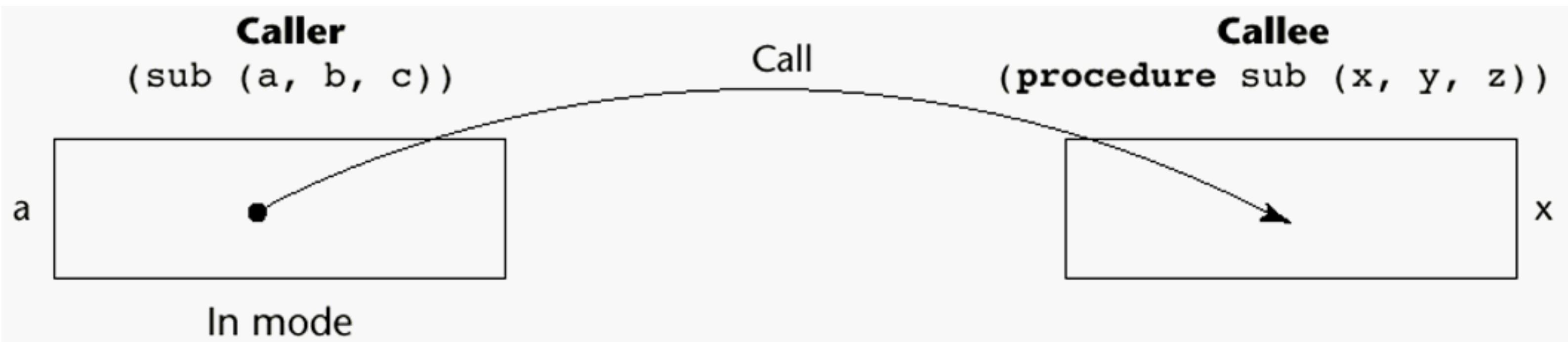
Implementation Models

Techniques used for parameter passing :

- Call by Value (In mode)
- Call by Result (Out mode)
- Call by Value-Result (In-out mode)
- Call by Reference (In-out mode)

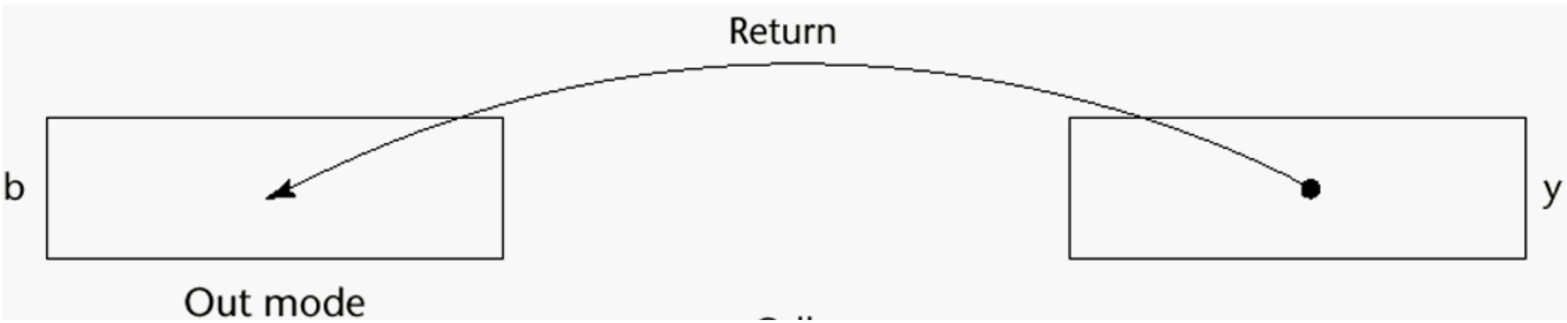
Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialized the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but then one need to enforce write protection.



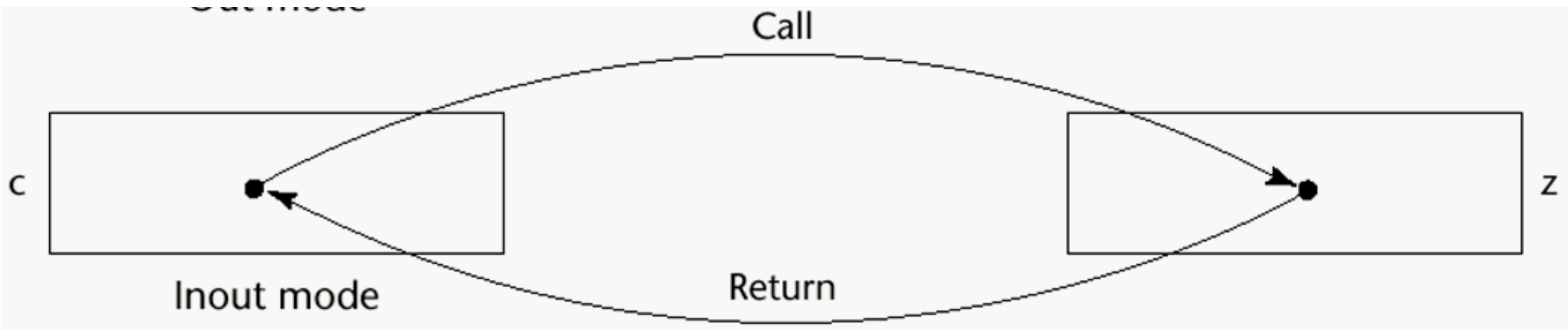
Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram;
  -the corresponding formal parameter acts as a local variable;
  -its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move



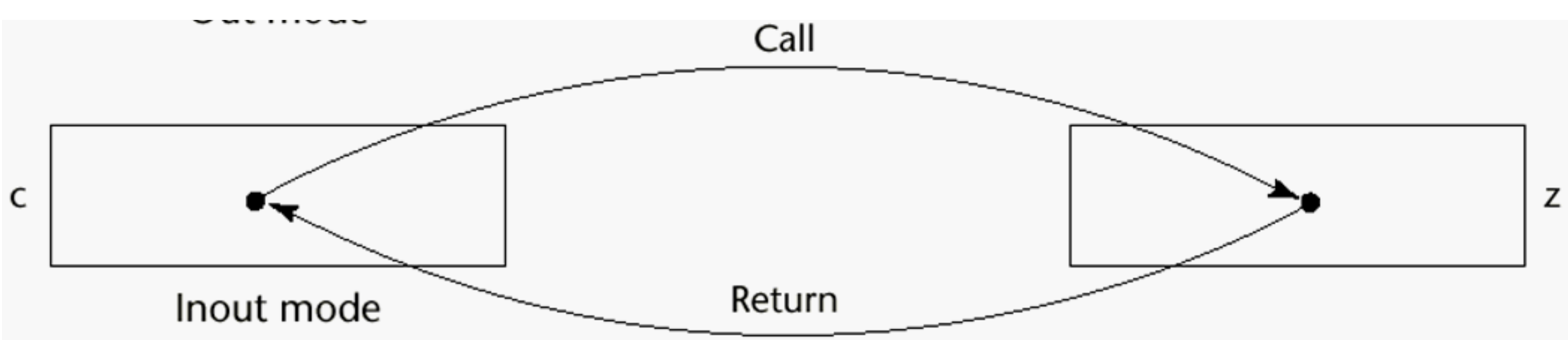
Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Actual values are copied in both directions.
- Formal parameters have local storage



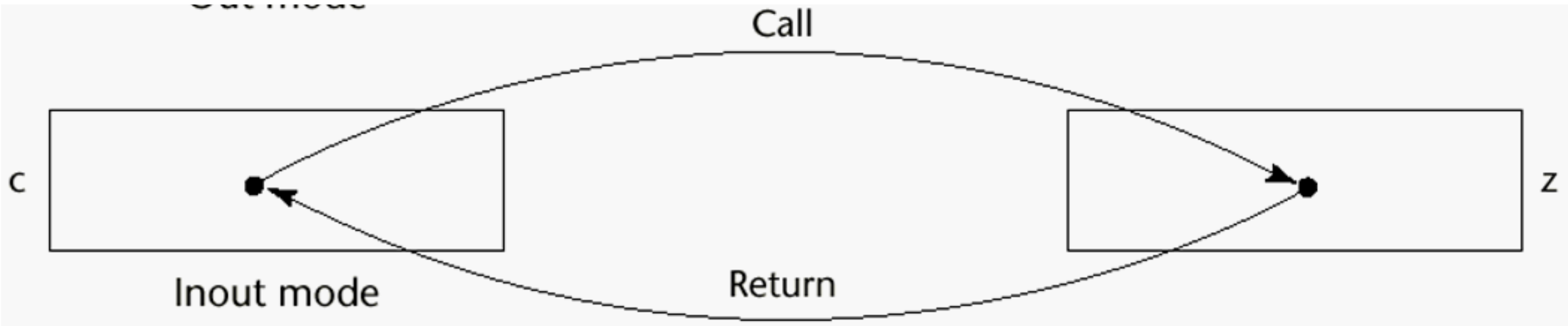
Pass-by-Reference (Inout Mode)

- Pass an access path to the value
- Passing process is efficient (no copying and no duplicated storage)
- Slower accesses (compared to pass-by-value) to formal parameters
- Potentials for unwanted side effects (collisions)
- Unwanted aliases (access broadened)



Pass-by-Name (Inout Mode)

- By textual substitution
- Formal parameters are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment

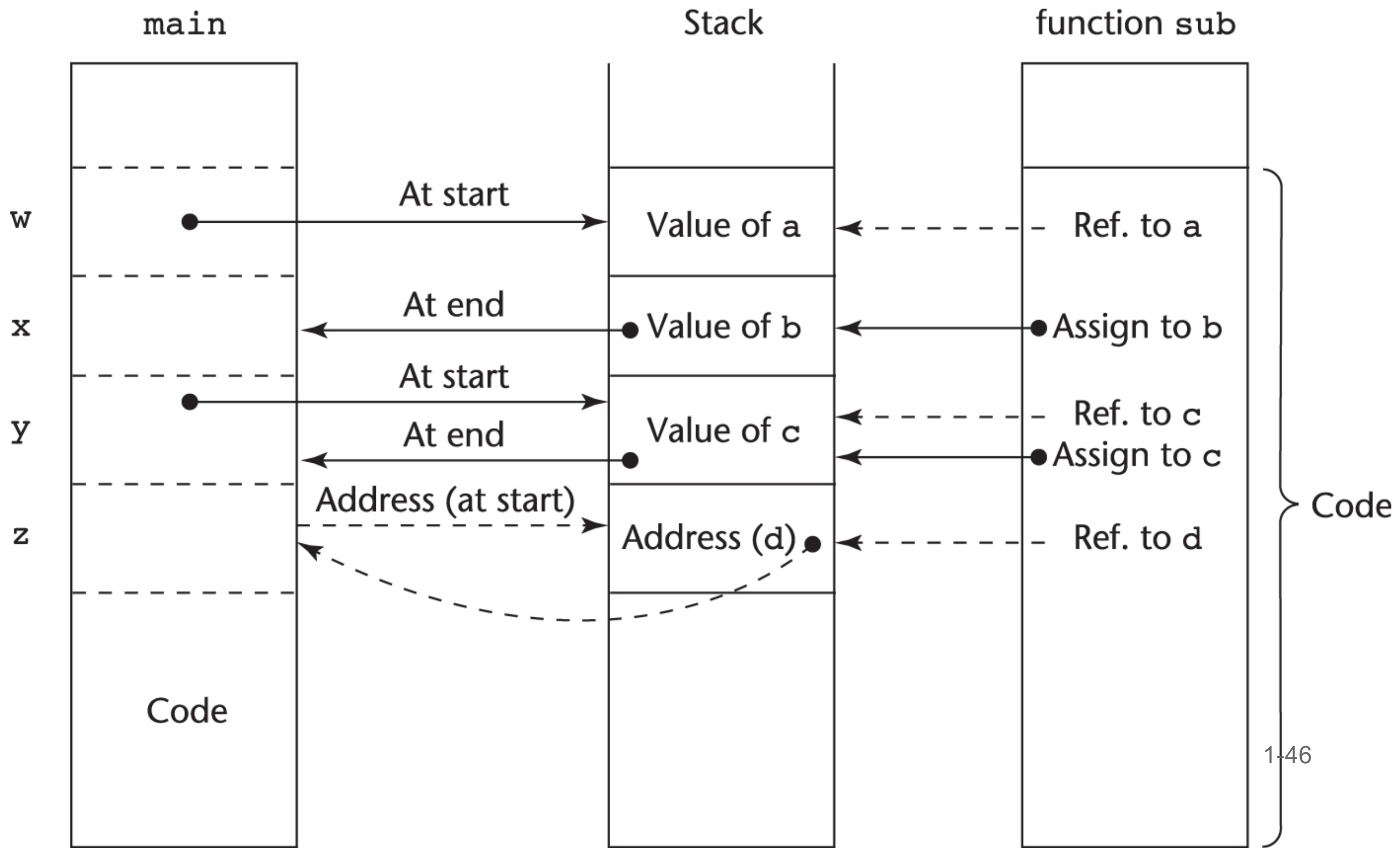


Implementing Parameter-Passing Methods

- In most languages parameter communication takes place through the run-time stack (more in the future)
- Pass-by-value parameters have their values copied into stack locations.
- Pass-by-reference are the simplest to implement; only an address is placed in the stack
- In Pass-by-result the caller reads from the stack the final value of the parameter before the stack of the callee is disposed

Implementing Parameter-Passing Methods

Function header: `void sub(int a, int b, int c, int d)`
Function call: `sub(w, x, y, z)`
(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)



Local variables

- Variables whose scope is usually the body of the subprogram in which they are defined

Local variables

```
let plus2 = fun x->
                let y = 2 in x + y
```

Here y is a local variable to the function `plus2`

Local variables?

What is the value of `y` here?

```
let y = 2 in
let plus2 = fun x-> x + y in plus2 (plus2 4)
```

How about y here?
Is it local?

And here?

Local variables?

```
Push v
Push x
 Bind
  Fun
Push x
Lookup
   …
  End
   …
 Call
```

What is the value of `x` here?