Subprograms

Based on the slides of Maurizio Gabbrielli

Abstraction of control

- Subprograms, blocks, parameters

```
double P (int x) {
    double z;
    /* Function body
    return expr;
}
```

- Without not knowing the code it is possible to
  - Specify P
  - Write P
  - Use P

Abstraction of control

- Provides functional abstraction to the project
  - each component provides services to its environment
  - its abstraction describes the external behavior
  - and hides the interior details needed to produce it

- Interaction limited to external behavior

- Communication through:
  - Parameters
  - Global environment (but destroys the abstraction)

Parameters

- Terminology:
  - Declaration/definition

```
int f (int n) {return n + 1;}
```

Formal parameter

  - Use/Call

```
x = f (y + 3);
```

Actual parameter

- Concrete: information flow between caller and callee
  - main $\longrightarrow$ proc
    - x = f(y+3);
  - main $\longleftarrow$ proc
    - procedure one (var y:integer); begin y:=1 end;
  - main $\longleftrightarrow$ proc

    procedure succ (var y:integer); begin y:=y+1 end;

How parameters are passed

- Two main ways:
  - By Value:
    - The value of the actual parameter is assigned to the formal, which behaves like a local variable
    - Main → Proc
    - Actual parameter any; changes to the formal do not pass to the actual
  - By reference
    - is passed a reference (address) at the actual parameter, references to the formal are references to the actual (*aliasing*)
    - Main ↔ Proc
    - Actual: variable; changes to the formal pass to actual

Call by value

```
void foo(int x) {x = x + 1;}
…
y = 1;
foo(y + 1);
```

Here Y is worth 1

- The formal x is a local var (on the stack)
- At the call, the actual y+1 is assessed and the value is assigned to the formal x
- No connection between x in the body of foo and y in the caller
- On return from foo, x is destroyed (removed from the stack)
- You cannot pass info from foo to the caller using the parameter
- Expensive for large data: copy
- Java, Scheme, Pascal (default), C

Call by reference (of variable)

```
void foo (reference int x){ x = x+1;}
…
y = 1;
foo(y);
```

Here Y is worth 2

- A reference is passed (address, pointer)
- The formal x is an alias of y
- The actual must be an l-value ("a variable")
- On return from foo, it is destroyed only the bond between x and the address of y
- Two-way transmission between caller and callee
- Efficient in the passage
- Pascal (var); in C done by passing a pointer…, in Java ?

Call by constant (or read-only)

- The call by value guarantees main → proc at the expense of efficiency
    - large data are copied even when they are not modified
- Read-only call (Modula-3; ANSI C: `Const`)
    - The procedure is not allowed to change the formal parameter (static compiler control: no assignment, no passage for ref to other proc)
    - Implementation depends on the compiler:
        - "small" parameters passed by value
        - "large" parameters passed by reference
- In Java: final

```
void foo(final int x) {// here x cannot be changed }
```

Call by result

- Opposite of call by value (main ← proc)
- Exists (-va) in Algol-W.

```
void foo (result int x) {x = 8;}
…
y = 1;
foo(y);
```

Here y is worth 8

- The formal x is a local var (on the stack)
- On return from foo the value of x is assigned to the actual y
- No connection between x in the body of foo and y in the caller
- On return from foo, x is destroyed (removed from the stack)
- It is not possible to pass info from the caller to foo using the parameter
- Expensive for large data: copy
- Ada: `Out`

Call by value/result

- Value + Result. Pragmatic: Main ↔ Proc
- Exists (-va) in Algol-W

```
void foo(value-result int x) {x = x + 1;}
…
y = 8;
foo(y);
```

Here y is worth 9

- The formal x is in all respects a local var (on the stack)
- To the call, the value of the actual is assigned to the formal
- On return, the value of the formal is assigned to the actual
- No connection between x in the body of foo and y in the caller
- On return from foo, x is destroyed (removed from the stack)
- Expensive for large data: copy
- Ada: `In Out` (but only for small data; for large data pass reference)

value-result ≠ reference

```
void foo (int x, int y, int z) {
   x = 2;
   y = 2;
   x = 4;
   if (x == y) z = 1;
   }
...
int a = 3;
int b = 0;
foo(a,a,b);
```

Java syntax (but these calls not available in Java)

Value-Result

z		0	0	0	0	
y		3	2	2	2	
x		3	2	4	4	
b	0	0	0	0	0	0
a	3	3	3	3	3	2 o 4

Reference

z b	0	0	0	1	1
y x a	3	2	4	4	4

Value and reference: summary

- Call by value:
  - Simple semantics: the body does not need to know how the procedure will be called (*referential transparency*)
  - Fairly simple implementation
  - Potentially expensive pass, efficient reference to the formal (direct access)
  - Need for other mechanisms to communicate main ← proc
- Call by reference:
  - Complex semantics; *aliasing*
  - Simple implementation
  - Efficient passage; a little more expensive the reference to the formal parameter (indirect access)

Value, and not reference

The benefits of passing by value

simple to see what happens → no changes in caller

Suggests languages with call by value only + separate mechanisms to get the call by reference:

pointers in C

classes types in Java

All this we know after more than 40 years of experience...

Back to basics (call by name)

The ALGOL Committee:

- How to give a simple mechanism to uniquely define the semantics of a procedure call?
- You see a call as a *macro expansion*:
  - the semantics of a call consist in the execution of the body as if it were textually replaced there
- How to manage parameters?
  - in the same way: semantics consist in the execution of the body after the current parameters are syntactically substituted to the formal
- These are *prescriptive* rules of semantics

Call by name

- In Algol-W was the default

```
int y;
void foo(name int x) {x = x + 1;}
...
y = 1;
foo(y);
```

$\boxed{\texttt{y = y + 1;}}$

- in this case the effect is similar to call by reference (which does not exist in Algol W)
- Haskell is using lazy-evaluation aka Call by Need
  - Call by need is a variant of call by name (if the function argument is evaluated, that value is stored for subsequent use)
- Let's see a more delicate case...

Call by name

```
int x=0;
int foo (name int y){
    int x = 2;
    return x + y;
}
...
int a = foo(x+1);
```

- Blindy applying the substitution
  - 5 (return x + x + 1)
- The variable x is *captured* by the local declaration
- If instead of x in foo we write z what does it change?
  - z=2; return z + x + 1 → 3
- We avoid this by requiring that the substitution is always understood without capture (imagine to use a fresh variable)
- Equivalent of asking formal parameter evaluated in the environment of the caller

Call by name: side effects

```
int i = 2;
int fie (name int y) {
   return y+y;
}
...
int a = fie(i++);
```

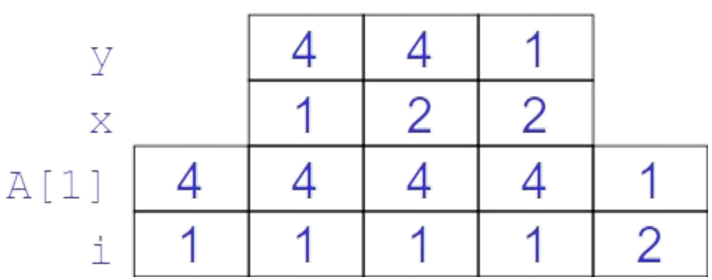
- Actual parameter evaluated every time
- Is that what we intended?

value-result ≠ name

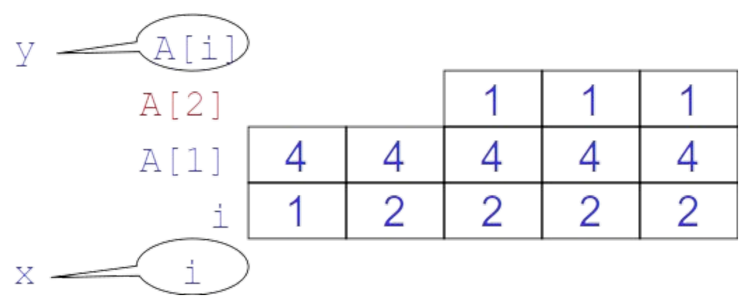
```
void fie (int x, int y) {
   x = x+1;
   y = 1;}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fie (i,A[i]);
```

Exercise: reference and value-result in this case have the same behavior.

Value-Result



Name



Call by name: implementation

- A pair is passed: <Exp, Env>
  - Exp is the actual parameter (text, not evaluated)
  - Env is the evaluation environment (as in static scoping)
- *Each time* that the formal is used, Exp is evaluated in Env.

```
int y ;
   void fie (int x ){
      int y;
      x = x + 1; y = 0;
      }
…
y = 1;
fie(y);
```

x |⟶ <y, □>

- Expensive: passage of environment + evaluation every time
  - Only ALGOL 60 and W

Call by name: implementing

- How to pass the pair <Exp,Env>?
  - A pointer to the text Exp
  - A (static chain) pointer on the stack to the activation record of the caller
  - this is a closure ("closes" the expression "deleting" the free variables by binding them to the environment)
  - closures are used to pass functions as arguments to other procedures

Higher order functions

- Some languages allow you to:
  - Pass functions as procedure arguments
  - Return functions as a result of procedures
- Both cases: how to manage the function environment?
- Simplest case
  - functions as an argument
  - Need a pointer to the activation record inside the stack: pass a closure!
- More complicated case
  - function returned by a procedure call
  - you must keep the activation record of the function returned → stack data structure does not work anymore

Functions as a procedure parameter

```
int x=4; int z=0;
int f (int y){
     return  x*y;}
void g ( int h(int n) ) {
     int x;
     x = 7;
     z = h(3) + x ;
     end;
…
{int x = 5;
 g(f);
}
```

- three declarations of x
- when f will be called (via h) which x will be used?

- With static scope the external x
- With *dynamic*, it makes sense both the x of the call block and the internal x

Deep vs Shallow binding

- When a procedure is passed as a parameter, a reference is created between a name (formal par h) and a procedure (actual par f).

  Problem: *which non-local environment applies at the execution time of f (as called via h)?*

  - Environment at the time of the creation of the link h → f?
    - *Deep binding*
  - Environment when calling f via h?
    - *Shallow binding*

Always used with static scope

Can make sense with dynamic scope

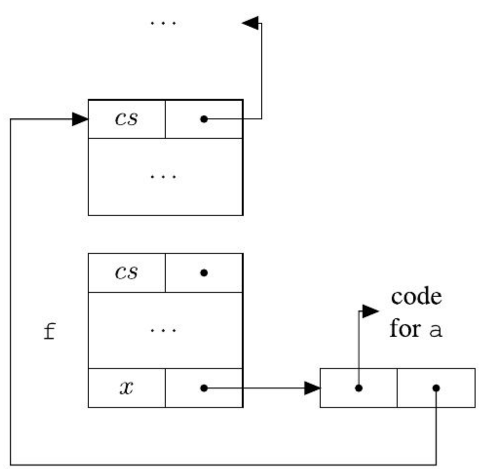
Deep and Shallow binding: example

```
int x=4; int z=0;
int f (int y){
      return  x*y;}
void g ( int h(int n) ) {
      int x;
      x = 7;
      z = h(3) + x ;
      end;
...
{int x = 5;
 g(f);
}
```

- **`h(3)`** invokes **`f`** that accesses **`x`** Which **`x`**?
- Static Scope
  - Deep: **`x`** red (external)
  - Shallow: **`x`** red (external)
  - *The scope rule is enough!*
- Dynamic Scope
  - Deep: **`x`** blue (call block)
  - Shallow: **`x`** black (internal lock)

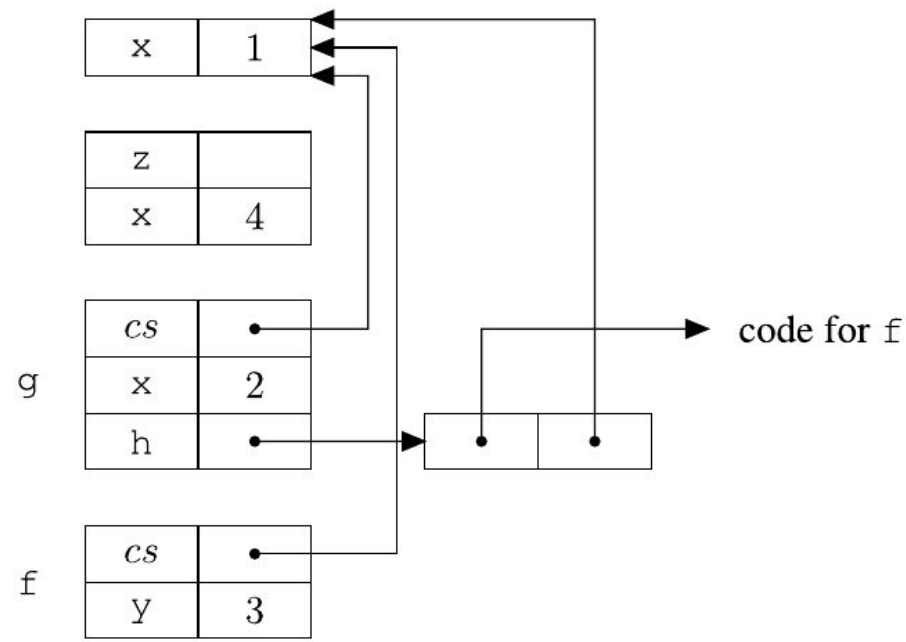
Closures (Implementation)

- Dynamically pass both the link with the code of the function, and its non-local environment
- When you call a procedure passed by parameter
  - Allocate (as always) the activation record
  - Get the static chain pointer from the closure



Static Scoping with functions as parameter

```
{int x = 1;
int f(int y){
      return x+y;
}
void g (int h(int b)){
      int x = 2;
      return h(3) + x;
}
...
 {int x = 4;
  int z = g(f);
 }
}
```



Dynamic Scope: implementation

- Shallow binding
  - Does not require any attention
    - To access x, use the stack
    - Use of usual data structures (A-list, CRT)

- Deep binding
  - Necessary to use some form of closure to "freeze" a scope to reactivate later

Deep vs Shallow binding

- Summarizing:
  - Dynamic scope
    - Deep binding
      - Implemented with closures
    - Shallow binding
      - Does not require further implementation
  - Static scope
    - Always used deep binding
      - Implemented with closures
    - At first glance deep or shallow makes no difference
      - it is the static scope rule to determine which non-local to use
      - not always true (more than one block declaring the non-local name can be available; e.g., when recursion is present)

Deep and shallow binding with static scope

- What value is assigned to Z in static scope and
  - Deep binding?

    1
  - Shallow binding?

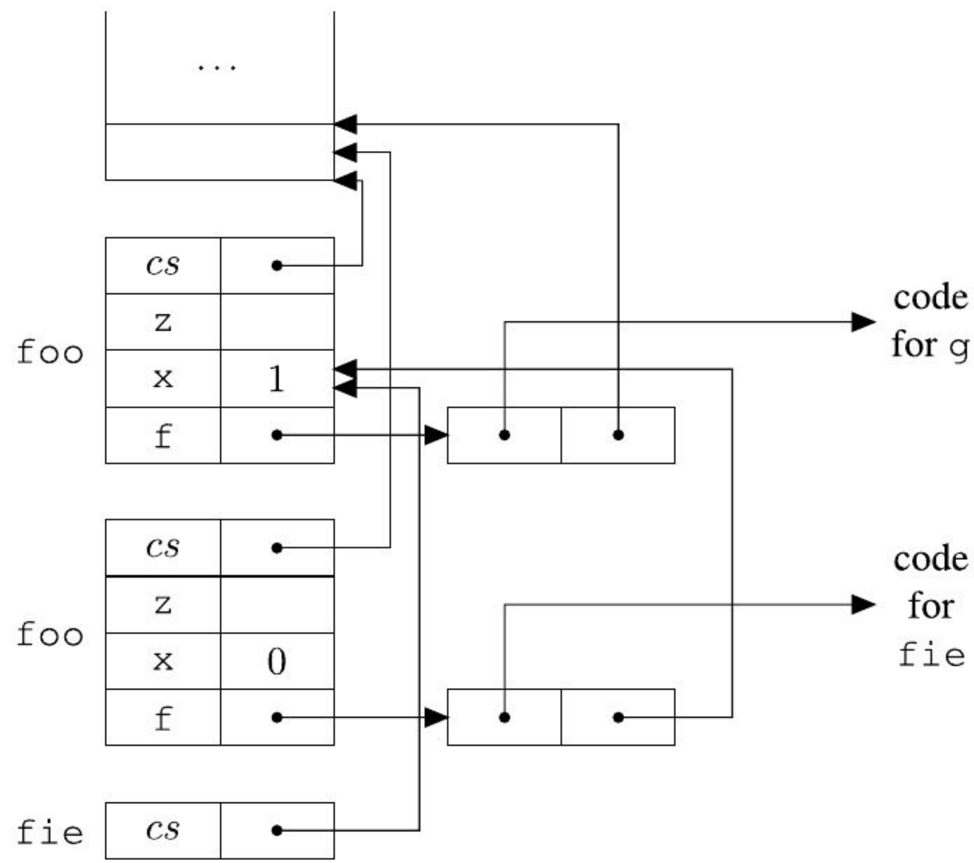
    (Not used!)

    0

```
{void foo (int f(), int x){
     int fie(){
         return x;
     }
     int z;
     if (x==0) z=f();
     else foo(fie,0);
 }
 int g(){
     return 1;
 }
 foo(g,1);
}
```

Deep Binding with static scope

```
{void foo (int f(), int x){
     int fie(){
        return x;
     }
     int z;
     if (x==0) z=f();
     else foo(fie,0);
 }
 int g(){
     return 1;
 }
 foo(g,1);
}
```



Returning a Function

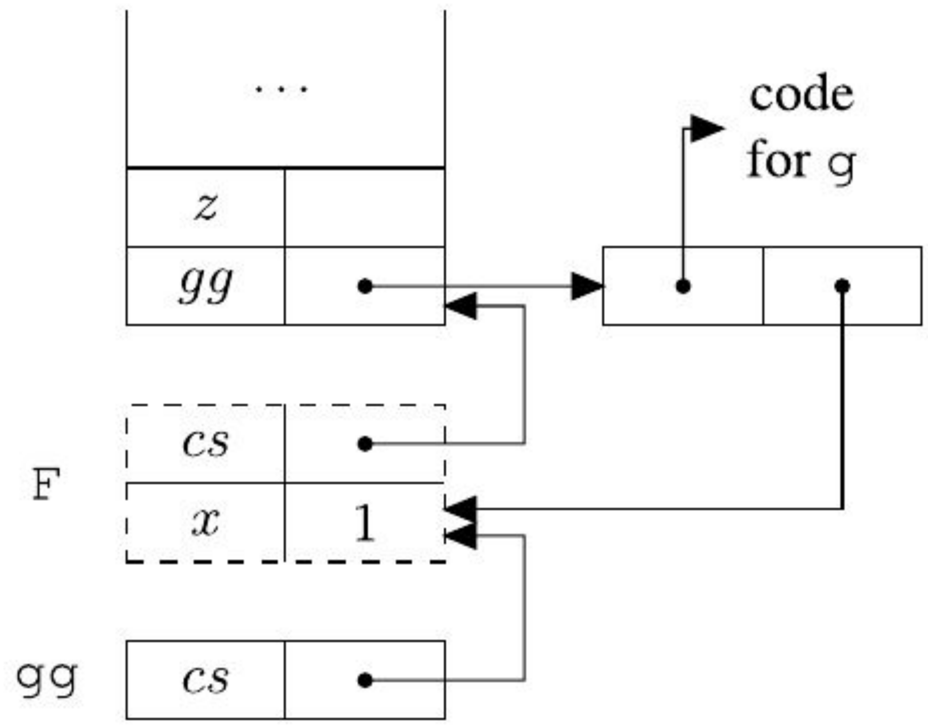
Problem:

- Some languages allow to *return a function*
- If the function has local variables these must survive independently of the stack structure used (they have a indefinitely long life)

Example: returning a function

What happens to binding to X after F ends? Can we call gg?

```
void->int  F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```



Morale: functions as a result

- Use of closures, but...
- Activation records persist indefinitely
  - Loss of stack properties
- How to implement:
  - do not explicitly deallocate
  - activation record on the heap
  - invokes the garbage collector when necessary

Suggested Exercises

- Chapter 7 exercises 1-5 (parameter passing)
- Chapter 6 exercise 7 (call by name)
- Chapter 5 exercise 7
- Chapter 4 exercises 6-13