Subprograms

- the basic abstraction mechanism
- *functions* correspond to the mathematical notion of computation:
  \[ \text{input} \rightarrow \text{output} \]
- *procedures* affect the environment, and are called for their side-effects
- pure functional model possible but rare (Haskell, Clean)
- hybrid model most common: functions can have side effects
declarations introduce names that denote entities
at execution-time, entities are bound to values or to locations:

\[
\text{name} \rightarrow \text{value} \quad \text{functional} \\
\text{name} \rightarrow \text{location} \rightarrow \text{value} \quad \text{imperative}
\]

value binding takes place during function invocation
names are bound to locations on scope entry
locations are bound to values by assignment
The rules that describe the binding of arguments to formal parameters, i.e., the meaning of a reference to a formal in the execution of the subprogram.

```
function f (a, b, c) ... // parameters: a, b, c
f(i, 2/i, g(i,j));       // arguments: i, 2/i, g(i,j)
```

- **by value**: formal is bound to value of actual
- **by reference**: formal is bound to location of actual
- **by copy-return**: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- **by name**: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- **by need**: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters
Parameter passing in Ada

- goal: separate semantic intent from implementation
- parameter modes:
  - **in**: read-only in subprogram (default)
  - **out**: write in subprogram
  - **in out**: read-write in subprogram
- independent of whether binding by value, by reference, or by copy-return
- functions can only have **in** parameters
Default values for in-parameters (Ada)

```ada
function Incr (Base: Integer;
    Inc: Integer := 1) return Integer;
```

```
Incr(A(J)) equivalent to Incr(A(J), 1)
```

also available in C++

```c++
int f (int first,
       int second = 0,
       char *handle = 0);
```

named associations (Ada):

```ada
Incr(Inc => 17, Base => A(I));
```
C: parameter passing by value, no semantic checks. Assignment to formal is assignment to local copy

if argument is pointer, effect is similar to passing designated object by reference

```c
void incr (int *x) {
    (*x)++;
}
incr(&counter); /* pointer to counter */
```

no need to distinguish between functions and procedures: `void` return type indicates side-effects only
default is by-value (same semantics as C)

explicit reference parameters:

```cpp
void incr (int& y) {
    y++;
}
incr(counter); // compiler knows profile of incr,
                // builds reference
```

semantic intent indicated by qualifier:

```cpp
void f (const double& val); // passed by reference,
                            // cannot be unbound
```
Parameter-passing in Java

- by value only
- semantics of assignment differs for primitive types and for classes:
  - primitive types have value semantics
  - objects have reference semantics
- consequence: methods can modify objects
- for formals of primitive types: assignment allowed, affects local copy
- for objects: *final* means that formal is read-only
procedure Outer (X: Integer) is
    Y: Boolean;
procedure Inner (Z: Integer) is
    X: Float := 3.0; -- hides outer x
function Innermost (V: Integer) return Float is
    begin
        return X * Float(V * Outer.X); -- use Inner.X
            -- and Outer.X
    end Innermost;
begin
    X := Innermost(Z); -- assign to Inner.X
end Inner;
begin
    Inner(X); -- Outer.X, the other one is out of scope
end;
Program example;

```
var
global: integer := 10;
another: integer := 2;
procedure confuse (var first, second: integer);
begin
  first := first + global;
  second := first * global;
end;
begin
  confuse(global, another); /* first and global */
  /* are aliased */
end
```

- Different results if by reference or by copy-return
- Semantics should not depend on implementation of parameter passing
- Passing by value with copy-return is less error-prone
Storage outside of the block

- With block structure, the lifetime of an entity usually coincides with the invocation of the enclosing construct.
- If the same entity is to be used for several invocations, it must be global to the construct.
  - In C++ can be declared `static` instead.
- Simplest: declare in the outermost context.
- Three storage classes:
  - `static`
  - Stack-based (automatic)
  - Heap-allocated
Bounded Nesting

- **C, C++, Java:**
  - no nested functions
  - blocks are merged with activation record of enclosing function
  - static storage available

- **Pascal, Ada:**
  - arbitrary nesting of packages and subprograms
  - packages provide static storage
Run-time organization

- each subprogram invocation creates an activation record
- recursion imposes stack allocation
- activation record hold actuals, linkage information, saved registers, local entities
- caller: place actuals on stack, return address, linkage information, then transfer control to callee
- prologue: save registers, allocate space for locals
- epilogue: place return value in register or stack position, update actuals, restore registers, then transfer control to caller
- binding of locations: actuals and locals are at fixed offsets from frame pointers
- complications: variable # of actuals, dynamic objects
Activation record layout

Frame pointer →
- actual 1
- actual 2
- return addr
- save area
- local 1
- local 2

Stack pointer →

Handled by caller

Handed by callee
printf("this is %d a format %d string", x, y);

- within body of `printf`, need to locate as many actuals as placeholders in the format string
- solution: place parameters on stack in reverse order (actuals at positive offset from FP, locals at negative offset from FP)

<table>
<thead>
<tr>
<th>actual n</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual n-1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>actual 1 (format string)</td>
</tr>
<tr>
<td>return address</td>
</tr>
</tbody>
</table>
Objects of dynamic size

declare
    X: String(1..N); -- N global, non-constant
    Y: String(1..N);
begin ...

Where is the start of Y in the activation record?

- **Solution 1**: use indirection: activation record holds pointers
  *simpler implementation, costly dynamic allocation/deallocation*

- **Solution 2**: local indirection: activation record holds offset into stack
  *faster allocation/deallocation, complex implementation*
procedure Outer is  -- recursive
    Gbl: Integer;
procedure Inner is  -- recursive
    Loc: Integer;
begin
    ...
    if Gbl = Loc then  -- how do we locate Gbl?
        ...
    end;
end;
begin
    ...
end;

- Need run-time structure to locate activation record of statically enclosing scopes.
- Environment includes current activation record and activation records of parent scopes.
Global linkage

- **static chain**: pointer to activation record of statically enclosing scope
- **display**: array of pointers to activation records
- does not work for function values
  - functional languages allocate activation records on heap
- may not work for pointers to functions
  - simpler if there is no nesting (C, C++, Java)
  - can check static legality in many cases (Ada)
Activation record holds pointer to activation record of enclosing scope. Set up as part of call prologue.

To retrieve entity $n$ scopes out, need $n$ dereference operations.
Global array of pointers to current activation records

To retrieve entity $n$ scopes out, need 1 indexing operation.
intermediate problem: functions that return values of non-static sizes:

```haskell
function Conc3 (X, Y, Z: String) return String is
  begin
    return X & "::" & Y & "::" & Z;
  end;

Str := Conc3(This, That, The_Other);
```

best not to use heap, but still need indirection

simple solutions: forbid it (Pascal, C) or use heap automatically (Java)
void (*pf)(double);
// pf is a pointer to a function that takes
// a double argument and returns void.

typedef void (*PROC)(int);
// Type abbreviation clarifies syntax.
// PROC is the type of a pointer to a function
// that takes an int argument and returns void.

void do_it(double d) { ... }

void use_it(PROC f) { ... f(5) ... }

PROC ptr = &do_it;

use_it(ptr);
use_it(&do_it);
procedure Outer (...) is
  type Proc is access procedure (X: Integer);
  procedure Perform (Helper: Proc) is begin
    Helper(42);
  end;
  procedure Action (X: Integer) is ...
  procedure Proxy is begin
    Perform(Action’access);
  end;
begin
  ...
end;

Action’access creates pair: (ptr to Action, env of Action)

*How does Proxy know what Action’s environment is?*

Simplest implementation of environment is a pointer (static link); can be display instead.
type Ptr is access function (X: Integer) return Integer;

function Make_Incr (X: Integer) return Ptr is
  function Incr (Base: Integer) return Integer is
  begin
    return Base + X;  -- reference to formal of Make_Incr
  end;
begin
  return Incr’access;  -- will it work?
end;

Add_Five: Ptr := Make_Incr(5);

Total: Integer := Add_Five(10);  -- where does Add_Five
      -- find X ?
First-class functions: implementation implications

Allowing functions as first-class values forces heap allocation of activation records.

- environment of function definition must be preserved until the point of call: activation record cannot be reclaimed if it creates functions
- functional languages require more complex run-time management
- higher-order functions: functions that take (other) functions as arguments and/or return functions
  - powerful
  - complex to implement efficiently
  - imperative languages restrict their use
  - (a function that takes/returns pointers to functions can be considered a higher-order function)
Both arguments and result can be (pointers to) subprograms:

type Func is access function (X: Integer) return Integer;
function Compose (First, Second: Func) return Func is
declare
  function Result (X: Integer) return Integer is
  begin
    return Second(First(X); -- implicit dereference
    -- on call
  end;
begin
  return Result’Access;
end;

This is illegal in Ada, because First and Second won't exist at point of call.
Restricting higher-order functions

- C: no nested definitions, so environment is always global
- C++: ditto, except for nested classes
- Ada: static checks to reject possible dangling references
- Modula: pointer to function illegal if function not declared at top-level
- ML, Haskell: no restrictions – compose is easily definable:

```haskell
define compose f g x = f (g x)
```