Programming Languages

Subprograms

Spring 2006
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 awkward)
hybrid model most common: functions can have (limited) side effects
declarations introduce names that denote entities

at execution-time, entities are bound to values or to locations:

name $\mapsto$ value \hspace{2cm} \textit{functional}

name $\mapsto$ location $\mapsto$ value \hspace{1cm} \textit{imperative}

value binding takes place during function invocation

names are bound to locations on scope entry

locations are bound to values by assignment
Parameter passing

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.

- **by value**: formal is bound to value of actual
- **by reference**: formal is bound to location of actual
- **by name**: formal is bound to expression for actual
- **by copy-return**: formal is bound to value of actual; upon return from routine, actual gets copy of formal
Parameter passing in Ada

- separate semantic intent from implementation
- parameter modes:
  - `in`: read-only in subprogram
  - `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
- history: can assign to global variables
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 (Delt => 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 indicates side-effects only
Parameter-passing in C++

- 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, but call cannot modify it
  ```
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;
Parameter passing anomalies

```pascal
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 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
- simplest: declare in the outermost context
- three storage classes:
  - static
  - stack-based (automatic)
  - heap-allocated
Bounded Nesting

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

- **Ada:**
  - arbitrary nesting of packages and subprograms
  - packages provide static storage

- **early C++, Java:** 3 levels:
  - static objects
  - class members
  - entities local to a member

- **current C++, Java:** nested classes provide arbitrary nesting
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
- actual
- return addr
- save area
- local
- local

Stack pointer →

{ Handled by caller

{ Handled 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)

```
actual n
actual n-1
...
actual 1 (format string)
return address
```
declare
    X: String (1..N); -- N global, non-constant
    Y: String (1..N);
begin ...
procedure Outer is -- recursive
  Gbl: Integer;
procedure Inner is -- recursive
  Loc: Integer;
begin
  ...
  if Gbl = Loc then -- how do we locate Gbl?

- 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 \( n \) frames out, need \( n \) dereference operations.
Global array of pointers to current activation records

To retrieve entity $n$ frames out, need 1 indexing operation.
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;

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

How does Proxy know what Action’s environment is?

Simplest implementation if environment is a pointer (static link) but can also be display.
void (*pf) (string);
// pf is a pointer to a void function
// that takes a string argument

typedef void (*PROC)(int);
// type abbreviation clarifies syntax

void use_it (PROC);
PROC ptr = &do_it;
use_it (ptr);
use_it (&do_it);
type Ptr is access function (X: Integer) return Integer;

function Make_Incr (X: Integer) return Ptr is
  function New_Incr (Base: Integer) return Integer is
    begin
      return Base + X;  -- reference to formal of Make_Incr
    end;
  begin
    return New_Incr’access;  -- will it work?
  end;
Add_Five: Ptr := Make_Incr (5);
Total: Integer := Add_Five (10);  -- where does Add_Five
  -- find X ?
Allowing functions as first-class values forces heap allocation of activation record.

- 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
  - 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:

```ada
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; 
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 restriction – compose is easily definable:

```haskell
fun compose f g x = f (g x)
```
intermediate problem: functions that return values of non-static sizes:

```plaintext
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)