Overview of Course

- Names, References, Pointers
- Expressions, Control Flow
- Types
- Blocks, Procedures
- Exceptions
- Modularity
- Object-Oriented Programming
Next Few Sections

- block structure
  - procedures and parameter passing
  - dynamic versus static scoping
  - implementation of non-local access
  - downward/upward funarg

- exception handling

- modules
  - data abstraction
  - transparent versus opaque data types

- OO programming
Overview of Blocks Procedures

- Significance of blocks and scope
- Scope rules
- Dynamic versus static scoping
- Procedures and parameter passing
- Implementation of non-local access
- Downward/upward funarg
- Implications for programming language design
Abstraction

*Abstraction*: conceptualization without reference to specific instances. The act of determining the fundamental, essential, important, intrinsic properties disassociated from the specific details.

*Abstraction*: Any generalization technique that ignores or hides details to capture some kind of commonality between different instances for the purpose of controlling the intellectual complexity of engineered systems, particularly software systems. Two kinds of abstraction: *process abstraction* and *data abstraction* (cf. modularity)
The fundamental difference between (old) Fortran and Algol-like languages is the block. A *block* has two intertwined meanings. First, a block is the access-defining unit of the language. Second, a block is a sequence of executable statements that are treated as a unit. In most languages subprocedures are blocks in both senses of the word. In Pascal the *only* access-defining unit is the procedure. In contrast, Algol and Ada have other access-defining constructs called blocks.

Blocks play a diminished role in OO languages like Java and C++.
Blocks

declare
    -- declarations
begin
    -- executable code
end;

In Java declarations can appear anywhere in a block (syntactically like another form of statement, but declarations are not runtime actions). The Ada syntax is more explicit (and verbose).

/* Java */  -- Ada
declare
int x;
X: Integer;
x = 3;
begin
int y=x+4;
Y: Integer := X+4;
y = 2*y;
begin
    Y := 2*y;
end;
end;
Names of many kinds of things are introduced by declaration in blocks. Not just variables.

```plaintext
declare
    -- declarations of exceptions
E,F,G,H: exception;
    -- procedure declaration
procedure R;
    -- declaration: variable
    -- introduction with attributes
X: Integer;
    -- declaration and runtime action
Y: Integer := 3;
    -- declaration, but possibly
    -- no action during runtime
Z: constant Integer := 3;
begin
    X := 3;     -- action
end;
```
Blocks can be nested in other blocks and in subprocedures. Blocks without names must be “called” or executed at the point in the program where they are defined. On the other hand, subprocedures (blocks with names) may be called from a place different from the place in the program where they are defined.
Blocks and procedures are similar. A block is a simple form from a subprocedure.

Main: declare
procedure R is
  I: Integer;
begin
  null; -- do
end R;
begin
  R; -- call R
end Main;

Main: begin
  R: declare
    I: Integer;
    begin
      null; -- do
    end R;
end Main;
Unlike blocks, one may parametrize subprocedures and reuse them. This is a tremendous advantage and extremely important, but it is the commonality of the extent of the variables we focus on. One may reuse subprocedures and avoid repeating the code (again very important), and the reuse of subprocedures in different contexts gives rise to important semantic and implementation considerations.
Recall . . .

**Scope.** The *scope* of a declaration is the portion of the program text in which the identifier introduced in the declaration is visible; i.e., has the attributes given to it by the declaration. We also speak of *visibility*. More about scope after introducing “block structure” . . .

**Extent.** The *extent* of a location in memory is the period of program execution in which the location can be accessed by some name or expression in the program.
From the perspective of some point in the program, the *local environment* is the collection of bindings declared in current block. The *nonlocal environment* is the collection of all the accessible variables not declared in the current block.
Blocks are an important programming tool in managing the complexity of programs.

*Localization of scope*. An important principle in making efficient, clear programs is *localization of scope*. Identifiers should be declared in the smallest scope containing its uses.

Though not directly related to blocks, the following definition captures the essence of an additional technique to reduce the complexity of programs.

*Single assignment style*. In single assignment style every variable is assigned exactly once.

Such variables can be declared `final` in Java.
Here is an example of using single assignment style in Java to compute the month of Easter in the Gregorian calendar. Notice that it was deemed preferable to assign to the variable epact twice, and so it cannot be declared final. All the variables are declared just before they are needed.

```java
final int year = Integer.parseInt(args[0]);
final int golden = (year % 19) + 1;
final int century = (year / 100) + 1;
final int gregorian = (3*century / 4) - 12;
final int clavian = ((8*century + 5)/25) - 5;
int epact = (11*golden+20+clavian-gregorian)%30;
if (epact==24 || (epact==25&&golden==11)) epact ++;
final int moon = (44-epact<21 ? 74:44)-epact;
final int extra = (5*year / 4) - gregorian - 10;
final int sunday = moon + 7 - ((extra+moon)%7);
final String month = (sunday>31)?"April":"March";
```
Here is an example of using single assignment style in Java to compute the month of Easter in the Gregorian calendar.

```java
final int year = Integer.parseInt(args[0]);
final int golden = (year % 19) + 1;
final int century = (year / 100) + 1;
final int gregorian = (3*century / 4) - 12;
final int clavian = ((8*century + 5)/25) - 5;
final int pre_epact = (11*golden+20+clavian-gregorian)%30;
final int epact =
    (epact==24 || (epact==25&&golden==11)) ? pre_epact+1 : pre_epact;
final int moon = (44-epact<21 ? 74:44)-epact;
final int extra = (5*year / 4) - gregorian - 10;
final int sunday = moon + 7 - ((extra+moon)%7);
final String month = (sunday>31)?"April":"March";
```
Epact

The epact (Latin epactae, from Greek: epaktai hèmerai = added days) has been described as the age of the moon in days on January 1, and occurs primarily in connection with tabular methods for determining the date of Easter. It varies (usually by 11 days) from year to year, because of the difference between the solar year of 365–366 days and the lunar year of 354–355 days.

ʻee-pakt
Call graph

Call graph of

funarg/main.adb

See also Sebesta, 8e, Figure 5.3, page 230.
Semantics of Blocks

Since the execution of a subroutine continues until all the subroutines it has called have returned, calls behave in a LIFO (stack) order. The static enclosing block is always (usually) in the stack of activation records making it possible to find a block’s nonlocal variables by skipping through the stack.
Scott 4e, Example 9.4, page 413.

Whether or not a subroutine is called directly by the lexically surrounding routine, we can be sure that the surrounding routine is active; there is no other way that the current routine could have been visible, allowing it to be called.
Interplay of static and dynamic factors.

 runtime/calls.adb
 runtime/nested.adb
 runtime/blocks.adb
 runtime/recurse.adb
 runtime/var_access.adb
The nested structure of scoping can be represented pictorially via contour diagrams or nested boxes. The local environment of every block appears in a box. The nonlocal environment comprises all the variables in surrounding boxes. This rule corresponds to **implicit inheritance**, i.e., variables declared in outer blocks are visible to inner blocks. Variables declared in inner blocks are not visible. If a nested block introduces an identifier declared in an outer block, then the outer declaration is eclipsed and there is a hole in its scope for the scope of the inner block. Almost all languages use implicit inheritance, although Euclid does not.
Scope Rules

Static scoping or lexical scoping means that a procedure is called in the environment of its definition. A procedure is defined in one place in the program, but may well be called in an entirely different environment or environments. If a procedure is called in the environment of its caller, then we have dynamic scoping sometimes known as dynamic binding, or fluid binding. This is the most recently occurring and still-active binding of a variable.
Static Versus Dynamic Scoping

Outer: declare
  b: Boolean := true; -- 1st decl of variable b
procedure P is
begin
  print (b);        -- variable b not local to P
end P;
begin
  Inner: declare
    b: Boolean := false; -- 2nd decl of b
  begin
    P;                  -- call procedure P
  end Inner;
end Outer;
Write a simple, Algol-like program that distinguishes dynamic scoping from static scoping. *Explain.*

[See previous program.]

In dynamic scoping “false” is printed because the environment of the caller (“Inner” block) is used; in static scoping the environment in which the procedure is defined is used (“Outer” block), so “true” is printed.
Grading Exam Question

10  obviously correct, exemplary explanation
9   -1 for absurd syntax, but clear explanation
8   -1 for too many declarations
7   -1 for too many blocks
6   clearly different behavior
5   separate environment for call site; weak explanation
4   a non-local variable
3
2   two declarations, same variable
1
0   blank sheet
Static versus Dynamic

Illustration of non-local and global variables in Python.

scope/scope1.py
scope/scope2.py
scope/scope3.py
scope/scope4.py – static versus dynamic
Static versus Dynamic

Scott 4e, Figure 3.9, 143.

n: integer
procedure first
    n:=1       -- non-local variable
procedure second
    n: integer
    first () -- call first
n :=2
if readInteger() > 0
    second()
else
    first()
writeInteger(n)
n: integer = 2
procedure first:
    third ()
    writeInteger(n) -- non-local variable
procedure second:
    n: integer = 1
    first () -- call first
procedure third:
    n: integer = 3
second()
Main: declare
N : Integer
procedure First is
begin
Third; Put (N);
end First;
procedure Second is
N : Integer;
begin
N := 1; First;
end Second;
procedure Third is
N : Integer;
begin
N := 3;
end Third;
begin
N := 2; Second;
end Main;
A procedure is a block with a name so that it can be called many times, perhaps in different contexts.
The subprocedure is an important building block in writing complex programs, because the details of a subprocedure can be ignored. Caller needs to know what it does; not how it does it.
If you find yourself writing the “same” code again and again, it is time for a subprocedure.
Interface

*Interface.* In general, an *interface* is the boundary between distinct systems. Specifically, the specification or protocol governing their interaction. *Procedure specification.* A *procedure specification* describes the interface between the procedure and the caller of the procedure. This may include the number, mode, default values and type of the procedure’s parameters, and, if a function, the return type. Many different terms are in use for this concept: *parameter profile, profile, protocol, prototype, type.*

*Side effect.* A *side effect* is some action or effect that a subprocedure has on the caller that is *not* advertised through the procedure specification. For example, a change to a non-local variable is a side effect. Even the use of a non-local may effect the outcome of a procedure call.
Procedures

*Actual argument.* An expression supplied as an argument in the procedure call is said to be an *actual argument.*

*Formal parameter.* The name used within the subroutine or function to refer to the actual argument supplied when the subroutine is called is said to be *formal parameter.* The actual arguments may vary every time the subroutine is called in the program, but the names of the formal parameters are given once in the definition of the subroutine. These names are picked by the programmer when the subroutine is written. Usually, the formal parameters and their types are introduced at the beginning of the subroutine’s definition.
Some common ways of associating formal parameters to the actual arguments are:

*Positional association*

*Named association* or *keyword association* or

*Default parameters*

Sebesta 8e, 9.2.3 Parameters, page 388 (formal parameters, positional parameters, keyword parameters, default values)
Sometimes parameters can have a variable number of arguments—even strongly-types languages, like Java.

Examples of these follow.
Ada Positional, Named, and Default

Python has positional, named (called keyword), and default parameters. But because it is not strongly typed, there is no overloading. Java has only positional association of parameters. Because of overloading there is no necessity for default parameters and the added complication of overloading resolution was judged to be too much.
Default Parameters in Python

def f(a, L=[]): -- L is the same mutable list
    L.append(a)
    return L

print f(4,[1,2,3]), f(1), f(2), f(3)

def g(a, L=None):
    if L is None:
        L = []
        L.append(a)
    return L

print g(4,[1,2,3]), g(1), g(2), g(3)

The output:

[1, 2, 3, 4] [1] [1, 2] [1, 2, 3]
[1, 2, 3, 4] [1] [2] [3]
Default Parameters

See also Scala’s call-by-name default parameters.
Variable Number Of Arguments

Ever wondered how `printf` worked in Java with any number of arguments. It treats them exactly as an homogeneous array. Similarly, Python treats the arguments a list.
A *parameter passing mechanism* is a facility provided by a programming language to pass data between the caller and the callee.

Webber, Chapter 18 Parameters, page 350. Sebesta, 8e, Section 9.5 Parameter Passing Methods.

We identify seven distinct means.

- call by reference – pass l-value
- call by value – pass r-value to local variable
- copy-out – local variable, initially undefined, copied back
- copy-in/copy-out – previous two mechanisms combined
  - text substitution
  - call by name
  - call by need
The common parameter-passing mechanisms are defined by how they are implemented. On the other hand it might make more sense to define the mechanisms by how the subprocedure is supposed to act.
Many programs pass data to subroutines using non-local variables. In the past FORTRAN’s COMMON block was used in this way. And today this is common in the instance variables of OO languages like Java. Because this reliance on non-local or external data sources is not described in the subprocedure specification, this practice leads to programs that are difficult to understand. Ideally the procedure specification (declaration) documents all communication in and out of a subprocedure. (Consider this issue in the context of raising exceptions, too.)
Side Effects Are Evil

Mike Myers as Dr. Evil
Call-By-Reference

The earliest parameter passing mechanism was call-by-reference in FORTRAN. Pass the l-value of actual to the procedure. Hence, the formal refers directly to same location as the actual argument.

```plaintext
declare
    A: Integer;
    B: Integer;
begin
    call P (A, B);
end;

procedure P
    X: Integer;
    Y: Integer;
begin
    ... X ...
    ... Y ...
end P;
```
Gries recounts that you could change the constant 2 to 3 in an early FORTRAN compiler!

Main: declare  
    procedure Change(out X: Int) is  
    begin  
        X := 3;  
    end Change;  
Y : Integer := 2  
begin  
    Change (Y);  
    Put (Y);  -- Y=3, of course  
end Main;  

Main: declare  
    procedure Change(out X: Int) is  
    begin  
        X := 3;  
    end Change;  
begin  
    Change (2);  -- illegal arg  
    Put (2);  
end Main;
Issues with Call-By-Reference

Different names referring to the same thing is the definition of aliasing.

1 `procedure/main.adb`
2 `procedure/parameter_aliasing.adb`
Call-By-Value

Allocate local storage for each formal parameter. Copy the r-value of each actual argument at the beginning of the call.

```
declarate
A: Integer;
B: Integer;
begind call P (A, B);
end;

procedure P
X: Integer;
Y: Integer;
begind ... X ...
... Y ...
end P;
```
In C parameters passed by value (only); how does information ever flow back to the caller? Parameters passed by reference can be simulated with pointers.

```c
void proc (int* x, int y) { *x = *x+y }
proc (&a,b);  // pass reference to a
```

But in Pascal and C++ the writer of the procedure can ask for call-by-reference.

```c
void proc (int& x, int y) { x = x + y }
proc(a,b);  // compiler knows to pass reference
```
Java uses only call-by-value. Primitive types cannot be passed by reference (wrapper classes are not a workaround). For primitive types this can be annoying. Everything else is an object. A pointer is passed as the “value” of an object. The efficiency of call-by-reference is achieved for references to objects. The appearance of call-by-reference is obtained with mutable objects (like arrays).
declare
  A: array 1..10 of Integer;
  i: Integer := 1;
procedure P (x: out Integer) is
begin
  i := 2;  x := 3;
end P;
begin
  -- L-value computed just before call
  P (A[i]);
end;
Copy-In/Copy-Out

Call-by-value (copy-in) combined with copy-out. The actual argument must have an l-value.
Parameter Passing

Copy-in/copy-out is not the same as call by reference.

declare
  G: Integer := 1;
procedure P (X: Integer) is
  -- reference ** copy-in/copy-out
begin
  -- ** X:=G:=1
  X := 6; -- G:=X:=6 ** X:=6
  G := 9; -- G:=9 ** G:=9
end P;

begin
  P (G); -- call P
end;
Implmentation or Intention?

Parameter passing defined by the method of **implementation** of the formal/actual association (call-by-reference, call-by-value, copy-in/copy-out).
Parameter passing defined by the **intent** of programmer in using the parameters: in, out, in/out.
Parameter passing defined by the **outcome** of the procedure call with the actuals substituted for the formals. Might be considered the correct way, but there are some implementation issues (text substitution, call-by-name, call-by-need.)
During the running of the program every subprocedure needs to know where the code is to execute next (a subprocedure is not always called from the same point in the program). In the absence of recursion, a global place for each subprocedure’s return address could be allocated. However, in the presence of recursion, it seems necessary and advantageous for the runtime system of a language to keep (during execution) a stack of addresses for all the pending subprocedure calls in order that each subprocedure can continue execution at the point in was suspending in order to execute another subroutine.
Implementing Block-Structured Languages

As a matter of fact, this can be extended to a structured record of a subprocedure’s invocation that includes not only the return address but arguments and local variables as well. The LIFO order makes it very efficient to reuse the space allocated to local variables. And, and..., the nonlocal variables seem to be in some other subprocedure’s (or block’s) local variables.
This activation record is a data structure providing parameters, local variables, the non-local environment, and the return address of each invocation of a subprocedure. Because of the nature of subprocedure call and return (LIFO), it is reasonable to create instances of these activation records on a stack. This stack, part of the runtime system and is called the run-time stack or simply the stack.
Static, hierarchical structure of `runtime/nested.adb` and dynamic, run-time behavior of the program.

```
R
  Q Q Q  R
  P P P P Q Q Q
N N N N N N N N N N
```

```
begin == time ==> end
```
The tricky part about implementing the run-time behavior of the program correctly is non-local variable access. Decisions about how to implement this access have a profound impact on programming language *design* as we shall see.
Block Structure

```plaintext
procedure Access is
  G: Integer := 2;

procedure Q is
  N: Integer := 3;

procedure R is
  L: Integer := 5;
  begin
    G := G + L; -- access G, L
  end R;

begin
  R;
  end Q;

procedure P is
begin
  Q;
  end P;

begin
  P; -- call subprocedure "P"
  Q; -- call subprocedure "Q"
end Access;
```
procedure Access is
  G: Integer := 2;

procedure Q is
  H: Integer := 3;

procedure R is
  L: Integer := 5;
  begin
    G := 3 + L;  -- access G, L
    end R;
  begin
    R;
    end Q;

procedure P is
  begin
    Q;
    end P;
  begin
    P;  -- call subprocedure "P"
    Q;  -- call subprocedure "Q"
  end Access;
Block Structure

procedure Access is
  G : Integer := 2;

procedure Q is
  H : Integer := 3;

procedure R is
  L : Integer := 5;
  begin
    G := 3+L;  -- access G, L
    end R;
  begin
    R;
    end Q;

procedure P is
  begin
    Q;
    end P;
  begin
    P;  -- call subprocedure "P"
    Q;  -- call subprocedure "Q"
  end Access;
A key observation about the stack implementation of the run-time environment is that two values pinpoint a variable in static binding. One, the *static distance* is the difference in static nesting between a variable use and its declaration. The *static nesting* is how deeply nested a declaration or variable use is. Two, *offset* is a more or less arbitrary displacement within an activation record locating where the variable is stored. At compile time this information can easily be stored in the symbol table and used to generate code to access the local or nonlocal variable. A local variable (static distance equal zero) is located in the current activation record. A nonlocal variable is located $n$ hops (where $n$ is the static distance) along the static chain. This fetch can be done in constant time if the static links are kept in an array.
The static link is a pointer (during the execution of the program) to the activation record of the \textit{statically} enclosing procedure or block. The static chain of the currently active procedure is the sequence of activation records representing the statically enclosing procedure or blocks. This static link must be maintained at runtime.
The static link is not equal to the dynamic link. Also note that in the presence of recursive calls more than one activation record for a given subprocedure may be on the call stack.

block/main.adb – static versus dynamic link, recursion
Implementing Variable Access

fetch M[EP + offset(v)] ; local variable

AR := M[EP].SL ; activation record of parent
fetch M[AR + offset(v)] ;

AR := M[EP].SL ; activation record of parent
AR := M[AR].SL ; traverse second static link
:
AR := M[AR].SL ; traverse last static link
fetch M[AR + offset(v)] ;
Subprogram Call

When the compiler encounters a subprogram call ... it determines the subprogram that declared the called subprogram, which must be a static ancestor of the calling routine. It then computes the nesting_depth, or number of enclosing scopes between the caller and the subprogram that declared the called subprogram. This information is stored and can be accessed by the subprogram call during execution. At the time of the call, the static link of the called subprogram’s activation record instance is determined by moving down the static chain of the caller the number of links equal to the nesting_depth computed at compile time.

Sebesta 11th, section 10.4.2 Static Chains, page 434–435
In a language with nested subroutines and static scoping (e.g., Ada, Common Lisp, ML, Scheme, or Swift), objects that lie in surrounding subroutines, and that are thus neither local nor global, can be found by maintaining a *static chain* (Figure 9.1). Each stack frame contains a reference to the frame of the lexically surrounding subroutine. This reference is called the *static link*. By analogy, the saved value of the frame pointer, which will be restored on subroutine return, is called the dynamic link. The static and dynamic links may or may not be the same, depending on whether the current routine was called by its lexically surrounding routine, or by some other routine nested int hat surrounding routine. Scott 4e,
Calling A Procedure

; transmit parameters
M[SP].PAR[1] := evaluation of parameter 1
M[SP].PAR[n] := evaluation of parameter n

M[EP].IP := resume ; save resume location

AR := M[EP].SL ; static parent, and so on
AR := M[AR].SL ;
M[SP].SL := AR ; set static link
EP := SP ; push new activation record
; callee should increment stack size
; SP := SP + size(callee’s activation record)
goto entry(callee)
resume:
OK, fine. The program of non-local variable access is marvelously solved. Let us push the boundaries.

“First-class subroutines in a language with nested scopes introduce an additional level of complexity: they raise the possibility that a reference to a subroutine may outlive the execution of the scope in which that routine was declared.”
Scott, page 144.
Passing Procedures

Consider programs in which procedure pass other procedures around as arguments.
Passing Procedures

1. What gets passed to Q to represent P or T?
2. What code must be emitted to implement the call fp(5)?

For any procedure call we must:

1. transmit the parameters,
2. save the return address,
3. set the dynamic link,
4. start the execution of the new procedure,
5. set the static link.

Closure. The value of a procedure is a closure—an ep, ip pair: environment pointer and code (instruction) pointer.
Closure

A closure is a program and the referencing environment where it was defined.

Sebesta, 11th, page 406.

In a language with first-class functions and static scope, a function value is generally represented by a closure, which is a pair consisting of a pointer to function code and a pointer to an activation record.

Mitchell, page 182.
Calling A Formal Procedure

; transmit parameters
M[SP].PAR[1] := evaluation of parameter 1

M[SP].PAR[n] := evaluation of parameter n

M[EP].IP := resume ; save resume location

M[SP].SL := fp.EP ; set static link
EP := SP ; push new activation record
; callee should increment stack size
; SP := SP + size(callee’s activation record)
goto fp.IP
resume:
Ada Program Examples

1. `funarg/main.adb` – nested blocks, non-local variables
2. `funarg/pass.adb` – passing procedures
3. `funarg/diff.adb` – procedures with different environment pointers
4. `funarg/down.adb` – again passing procedures, but trouble
5. `funarg/ret.adb` – returning procedures with different environment pointers
6. `funarg/up.adb` – trouble with returning procedures
7. `funarg/abuse.adb` – slipping closure out of its environment
Ada Program Examples

1 funarg/accessibility.adb – illustrate dynamic accessibility check
Passing a procedure down is not a problem; set the ep to the correct parent. Here is the stack of the program `funarg/down.adb` at different times during the execution of the program.
Passing a procedure up is a problem; the ep is set to an activation record that is popped off the stack. Here is the stack of the program `funarg/up.adb` at different times during the execution of the program.
Upward Funarg Problem

So, what is the conclusion? Either

1. Forbid returning and assigning subprocedures. E.g., Pascal.
2. Fine print with pointer types. E.g., Ada.
3. Restrict all local procedures. E.g., Modula-3. Even though downward function args would be quite safe, assigning procedure values makes it impossible to know if its use is upward or downward. Modula-3 encourages a “flat” structure.
4. No local procedures. E.g., C, C++, Java.
   1. recent lambdas implemented as closures as classes
5. Throw out the stack-based implementation. E.g., functional languages (discussed later).
Here are the details from the Ada reference manual:

**3.10.2 Operations of Access Types ¶24 X’Access**

The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X’Access is the general access type A:

¶29 b A check is made that the accessibility level of X is not deeper than that of the access type A. If this check fails, Program_Error is raised.

In general, this is determined statically, but access parameters and generics require dynamic checks. We saw this error in the program `funarg/up.adb`
procedure Up is

  type F is access procedure (X: Integer);

  procedure P (X: Integer) is begin null; end P;

  function R (B: Boolean) return F is
    N: Integer;
    procedure T (X: Integer) is begin N:=X; end T;
  begin
    if B then
      return P'Access; -- no problem
    else
      return T'Access; -- subprogram "T" deeper than access type "F"
    end if;
  end R;
2.3.1 Assignment An expression $e$ is assignable to a variable $v$ if:

- the type of $e$ is assignable to the type of $v$, and
- the value of $e$ is a member of the type of $v$, is not a local procedure, and if it is an array, then it has the same shape as $v$.

The first point can be checked statically; the others generally require runtime checks. Since there is no way to determine statically whether the value of a procedure parameter is local or global, assigning a local procedure is a runtime rather than a static error.

and …
... return is handled like an assignment ...

2.3.11 Return A RETURN statement for a function procedure has the form:

```
RETURN Expr
```

where Expr is an expression assignable to the result type of the procedure

funarg/Down.m3

funarg/Up.m3
The programming language C has no nested procedures, so a closure need only contain a pointer to the code. No restrictions on passing procedure pointers are needed.

In C++ and Java you can pass subprocedures only inside classes. Pointers to instances of classes are “closures;” they point to all the instance variables and all the methods.

Passing subprocedures in C++ and Java is clumsy.

inner/Searcher.java
member/function.cc
Summary

1. Block-structured languages required the development of new run-time techniques
2. An activation record represents the state of a procedure
3. Activation records contain the local variables
4. Instruction counter represents the site of control; the environment pointer represents the context
5. A static link points to the activation record of the lexical parent
6. Variables are addressed by two coordinates: static distance and offset
7. Accessing a variable requires two steps; can be done in one machine instruction