Programming Languages

Session 4 – Main Theme
Subprograms

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Adapted from course textbook resources
Programming Language Pragmatics (3rd Edition)
Michael L. Scott, Copyright © 2009 Elsevier
Agenda

1. Session Overview
2. Subprograms
3. Conclusion
What is the course about?

- **Course description and syllabus:**
  - [http://www.nyu.edu/classes/jcf/CSCI-GA.2110-001](http://www.nyu.edu/classes/jcf/CSCI-GA.2110-001)

- **Textbook:**
  - *Programming Language Pragmatics (3rd Edition)*
    - Michael L. Scott
    - Morgan Kaufmann

- **Additional References:**
  - Osinski, Lecture notes, Summer 2008
  - Barrett, Lecture notes, Fall 2008
  - Gottlieb, Lecture notes, Fall 2009
  - Grimm, Lecture notes, Spring 2010
Session Agenda

- Session Overview
- Subprograms
  - Functions and Procedures
  - Parameter Passing
  - Nested Procedures
  - First-class and higher-order functions
- Conclusion
Session 3 Review

- Control Flow
- Control Structures
- Statement Grouping
- Expression Evaluation
- Sequencing
- Semicolons
- Selection
- Lists / Iteration
- Recursion
- Conclusions
Agenda

1. Session Overview
2. Subprograms
3. Conclusion
Subprograms

- Subprogram
- Environment of the Computation
- Review of Stack Layout
- Calling Sequences
- Calling Sequences (C on MIPS)
- Parameter Passing
- Generic Subroutines and Modules
- First Class Functions
- Higher-Order Functions
- Block Structure
- Exception Handling
- Coroutines
- Conclusions
- the basic abstraction mechanism
- functions correspond to the mathematical notion of computation:
  - input $\rightarrow$ 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:

- name -> value (functional)
- name -> location -> value (imperative)

value binding takes place during function invocation

names are bound to locations on scope entry

locations are bound to values by assignment
Allocation strategies

» Static

• Code
• Globals
• Own variables
• Explicit constants (including strings, sets, other aggregates)
• Small scalars may be stored in the instructions themselves
Figure 8.1 Example of subroutine nesting, taken from Figure 3.5. Within B, C, and D, all five routines are visible. Within A and E, routines A, B, and E are visible, but C and D are not. Given the calling sequence A, E, B, D, C, in that order, frames will be allocated on the stack as shown at right, with the indicated static and dynamic links.
• Allocation strategies (2)
  » Stack
    • parameters
    • local variables
    • temporaries
    • bookkeeping information
  » Heap
    • dynamic allocation
Contents of a stack frame

» bookkeeping
  • return PC (dynamic link)
  • saved registers
  • line number
  • saved display entries
  • static link

» arguments and returns

» local variables

» temporaries
Maintenance of stack is responsibility of *calling sequence* and *subroutine prolog* and *epilog* (discussed in Chapter 3 of the textbook)

» space is saved by putting as much in the prolog and epilog as possible

» time *may* be saved by putting stuff in the caller instead, where more information may be known

  • e.g., there may be fewer registers IN USE at the point of call than are used SOMEWHERE in the callee
- each subprogram invocation creates an activation record (or stack frame)
  - The stack pointer contains the address of either the last used location or the next unused location on the stack.
  - The frame pointer points into the activation record of a subroutine so that any objects allocated on the stack can be referenced with a static offset from the frame pointer.
- 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
Why not use an offset from the stack pointer to reference subroutine objects?

» The stack pointer changes, with nested scopes and function/procedure calls

» Also, there may be objects that are allocated on the stack whose size is unknown at compile time

» These objects get allocated above the frame pointer so that objects whose size is known at compile time can still be accessed quickly
Why not use an offset from the stack pointer to reference subroutine objects?

- There may be objects that are allocated on the stack whose size is unknown at compile time.
- These objects get allocated last so that objects whose size is known at compile time can still be accessed quickly via a known offset from the frame pointer.

Example

```haskell
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
  ...
end
```
When a subroutine is called, a new activation record is created and populated with data.

The management of this task involves both the caller and the callee:

- The calling sequence refers to code executed by the caller just before and just after a subroutine call.
- The prologue refers to activation record management code executed at the beginning of a subroutine.
- The epilogue refers to activation record management code executed at the end of a subroutine.

Sometimes the term calling sequence is used to refer to the combined operations of the caller, prologue, and epilogue.
Common strategy is to divide registers into **caller-saves** and **callee-saves** sets
- caller uses the "callee-saves" registers first
- "caller-saves" registers if necessary

Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
- some storage layouts use a separate arguments pointer
- the VAX architecture encouraged this
Figure 8.2 A typical stack frame. Though we draw it growing upward on the page, the stack actually grows downward toward lower addresses on most machines. Arguments are accessed at positive offsets from the fp. Local variables and temporaries are accessed at negative offsets from the fp. Arguments to be passed to called routines are assembled at the top of the frame, using positive offsets from the sp.
Calling Sequences – Calling a subroutine

- **In the caller**
  - Store any caller-saved registers
  - Place arguments in registers and/or stack
  - Compute static link and pass as extra argument
  - (Save return address on stack)
  - Jump to subroutine

- **In the callee**
  - Allocate frame by changing stack pointer
  - Save old frame pointer and update with new value
  - Save any callee-saved registers
  - Initialize objects
Calling Sequences – Finishing a subroutine

- In the callee
  - Move return values (if any) into registers and/or stack
  - Restore callee-saved registers
  - Restore frame and stack pointers
  - Jump back to return address

- In the caller
  - Save return values
  - Restore caller-saved registers
Calling Sequences – Calling a subroutine

Calling Sequence (before)

Stack pointer →

Frame pointer →

Caller
Activation Record
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers

Stack pointer →
- Caller-Saved Registers
- Caller
- Activation Record

Frame pointer ←
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack

Stack pointer →
- Arguments
- Caller-Saved Registers
- Caller
- Activation Record

Frame pointer →
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue

Stack pointer →
- Return address
- Arguments
- Caller-Saved Registers
- Caller Activation Record

Frame pointer →
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers

Stack pointer →
- Variable-length Objects
- Temporaries
- Local Variables
- Callee-Saved Registers
- Saved fp
- Return address

Frame pointer →
- Arguments
- Caller-Saved Registers
- Caller Activation Record
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers

Epilogue
1. Restore callee-save registers

Stack pointer
- Callee-Saved Registers
- Saved fp
- Return address

Frame pointer
- Arguments
- Caller-Saved Registers
- Caller
- Activation Record
Calling Sequences – Calling a subroutine

**Calling Sequence (before)**
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

**Prologue**
1. Save old fp, set new fp
2. Save callee-save registers

**Epilogue**
1. Restore callee-save registers
2. Restore frame pointer
Calling Sequences – Calling a subroutine

= Calling Sequence (before)
  1. Save caller-save registers
  2. Push arguments on stack
  3. Jump to subroutine, saving return address on stack

= Prologue
  1. Save old fp, set new fp
  2. Save callee-save registers

= Epilogue
  1. Restore callee-save registers
  2. Restore frame pointer
  3. Jump to return address
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers

Epilogue
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers

Epilogue
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)
1. Restore caller-save registers
Calling Sequences – Calling a subroutine

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers

Epilogue
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)
1. Restore caller-save registers
Are there advantages to having the caller or callee perform various tasks?
If possible, have the callee perform tasks: task code needs to occur only once, rather than at every call site
However, some tasks (e.g. parameter passing) must be performed by the caller
- 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
Are there advantages to having the caller or callee perform various tasks?

If possible, have the callee perform tasks: task code needs to occur only once, rather than at every call site.

However, some tasks (e.g. parameter passing) must be performed by the caller.
One difficult question is whether the caller or callee should be in charge of saving registers.

What would the caller have to do to ensure proper saving of registers?
   » Save all registers currently being used by caller

What would the callee have to do to ensure proper saving of registers?
   » Save all registers that will be used by callee

Which is better?
   » Could be either one—no clear answer.
   » In practice, many processors (including MIPS and x86) compromise: half the registers are caller-save and half are callee-save
   » Register Windows offer an alternative: each routine has access only to a small window of a large number of registers; when a subroutine is called, the window moves, overlapping a bit to allow parameter passing
Global array of pointers to current activation records

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

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
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.
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
Calling Sequences – Run-Time Access to Globals

procedure Outer is -- recursive
    Gbl : Integer;
procedure Inner is -- recursive
    Loc : Integer;
begin
    ...
    if Gbl = Loc then -- how do we locate Gbl?
    ...
    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
- 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)
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 ?
### Leaf routines

- A leaf routine is one which does not call any subroutines.
- Leaf routines can avoid pushing the return address on the stack: it can just be left in a register.
- If a leaf routine is sufficiently simple (no local variables), it may not even need a stack frame at all.

### Inlining

- Another optimization is to inline a function: inserting the code for the function at every call site.
- What are advantages and disadvantages of inlining?
  - Advantages: avoid overhead, enable more compiler optimizations
  - Disadvantages: increases code size, can’t always do it (i.e. recursive procedures)
**Caller**

- saves into the temporaries and locals area any caller-saves registers whose values will be needed after the call
- puts up to 4 small arguments into registers $4$-$7$ (a0-a3)
  - it depends on the types of the parameters and the order in which they appear in the argument list
- puts the rest of the arguments into the arg build area at the top of the stack frame
- does jal, which puts return address into register ra and branches
  - note that jal, like all branches, has a delay slot
In prolog, Callee
- subtracts framesize from sp
- saves callee-saves registers used anywhere inside callee
- copies sp to fp

In epilog, Callee
- puts return value into registers (mem if large)
- copies fp into sp (see below for rationale)
- restores saved registers using sp as base
- adds to sp to deallocate frame
- does jra
After call, Caller
   » moves return value from register to wherever it's needed (if appropriate)
   » restores caller-saves registers lazily over time, as their values are needed

All arguments have space in the stack, whether passed in registers or not

The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory
This is a normal state of affairs; optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope.
Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases

- particularly LEAF routines (those that don't call other routines)
  - leaving things out saves time
  - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast
Definitions

- Formal parameters are the names that appear in the declaration of the subroutine.
- Actual parameters or arguments refer to the expressions passed to a subroutine at a particular call site.

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

- The rules that describe the binding of arguments to formal parameters using parameter modes, 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)

- Modes:
  - 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 mechanisms have three basic implementations

» value
» value/result (copying)
» reference (aliasing)
» closure/name

Many languages (e.g., Pascal) provide value and reference directly
C: functions

- parameters passed by value (C)
  - no semantic checks. Assignment to formal is assignment to local copy
- parameters passed by reference can be simulated with pointers (C)
  
  ```c
  void proc(int* x, int y) {*x = *x+y } ...
  proc(&a,b);
  ```
  - if argument is pointer, effect is similar to passing designated object by reference
  - no need to distinguish between functions and procedures: void return type indicates side-effects only
C++: functions

- default is by-value (same semantics as C)
- or directly passed by reference (C++)

```cpp
void proc(int& x, int y) {x = x + y }
proc(a,b);
```

- explicit reference parameters:

```cpp
void f ( const double & val ); // passed by reference ,
   // but call cannot
   // modify it
```
Ada goes for semantics: who can do what via parameter modes

- \( \text{In:} \) callee reads only
- \( \text{Out:} \) callee writes and can then read (formal not initialized); actual modified
- \( \text{In out:} \) callee reads and writes; actual modified
- independent of whether binding by value, by reference, or by copy-return
- functions can only have in parameters

Ada in/out is always implemented as

- value/result for scalars, and either
- value/result or reference for structured objects
- 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
Parameter Passing – C#

- As in Java: by-value is default. For class types this means the reference is immutable, but not the object

- Parameter can indicate intent:

  - `out` double X;       // uninitialized, by reference
  - `ref` int X;          // in out, by reference
In a language with a reference model of variables (Lisp, Clu), pass by reference (sharing) is the obvious approach.

It's also the only option in Fortran:

- If you pass a constant, the compiler creates a temporary location to hold it.
- If you modify the temporary, who cares?

Call-by name is an old Algol technique:

- Think of it as call by textual substitution (procedure with all name parameters works like macro) - what you pass are hidden procedures called THUNKS.
 All parameter-passing by value
 no assignment.
 local declarations of constants only.
 consequence: functions have no side-effects.
 referential transparency: two occurrences of the same expression have the same meaning.
 awkward if need to describe computations with history, e.g. a random number generator.
### Figure 8.3 Parameter passing modes.

<table>
<thead>
<tr>
<th>Parameter mode</th>
<th>Representative languages</th>
<th>Implementation mechanism</th>
<th>Permissible operations</th>
<th>Change to actual?</th>
<th>Alias?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>C/C++, Pascal, Java/C# (value types)</td>
<td>value</td>
<td>read, write</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>in, const</td>
<td>Ada, C/C++, Modula-3</td>
<td>value or reference</td>
<td>read only</td>
<td>no</td>
<td>maybe</td>
</tr>
<tr>
<td>out</td>
<td>Ada</td>
<td>value or reference</td>
<td>write only</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>value/result</td>
<td>Algol W</td>
<td>value</td>
<td>read, write</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>var, ref</td>
<td>Fortran, Pascal, C++</td>
<td>reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sharing</td>
<td>Lisp/Scheme, ML, Java/C# (reference types)</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in out</td>
<td>Ada</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>name</td>
<td>Algol 60, Simula</td>
<td>closure (thunk)</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>need</td>
<td>Haskell, R</td>
<td>closure (thunk) with memoization</td>
<td>read, write*</td>
<td>yes*</td>
<td>yes*</td>
</tr>
</tbody>
</table>
C and C++ allow parameters which are pointers to subroutines:

```c
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

void do_it(double d) { ... }

void use_it (PROC);

PROC ptr = &do_it;

use_it(ptr);
use_it(&do_it);
```

Are there any implementation challenges for this kind of subroutine call?
Parameter Passing – Passing Subroutines as Parameters

- Not really: can be implemented in the same way as a usual subroutine call: in particular the referencing environment can stay the same.
- What if a nested subroutine is passed as a parameter?
- A closure must be created and passed in place of the subroutine.
- A closure is a reference to a subroutine together with its referencing environment.
- When a subroutine is called through a closure, the referencing environment from when the closure was created is restored as part of the calling sequence.
# Compose several parsers in sequence

def make_seq(*ps):
    def parse(s, i):
        for p in ps:
            i = p(s, i)
            if -1 == i: return -1
        return i
    return parse

- The make_seq function takes zero or more parser functions as its only argument. It defines a nested parse function, which applies these functions on some string and at some starting index in order. It returns the closure of nested function and environment, because parse needs to access ps to work.
- (Using parser functions to compose other parser functions is a well-known technique, called parser combinators.)
Default values for in-parameters (Ada)

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

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

also available in C++

int f ( int first ,
    int second = 0,
    char * handle = 0);
	named associations (Ada):
    » Incr ( Delt => 17, Base => A(I ));
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
Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects.

Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right.
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)
Higher-Order Functions

- 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
Higher-Order Functions - Restricting

- 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:
  
  ```
  fun compose f g x = f (g x)
  ```
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 ;
Pascal program example of Reference vs. Copy-Return;

```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
• 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, C++, can be declared static instead
• simplest: declare in the outermost context
• three storage classes:
  » static
  » stack-based (automatic)
  » heap-allocated
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
What is an exception?
a hardware-detected run-time error or unusual condition detected by software

Examples
arithmetic overflow
end-of-file on input
wrong type for input data
user-defined conditions, not necessarily errors
What is an exception handler?
- code executed when exception occurs
- may need a different handler for each type of exception

Why design in exception handling facilities?
- allow user to explicitly handle errors in a uniform manner
- allow user to handle errors without having to check these conditions
- explicitly in the program everywhere they might occur
Exception Handling - Summary

- General mechanism for handling abnormal conditions
- Predefined exceptions: Constraint violations, I/O errors, other illegalities
- User-defined exceptions
- Exception handlers specify remedial actions or proper shutdown.
Exception Handling

- From the programmer’s point of view this is a hard construct to use.
  - Where should an exception be handled
  - What information is needed to handle it? Often necessary to combine local information (what error was raised?) with global context (what should be done?).

- Difficult for formal analysis.

- However, for our purposes (PL), not so bad.
In Ada, C++, Java, handlers are attached to statements.
If the current statement has a handler for this exception, execute it.
Else move lexically outward, until finding a handler.
If the routine has no suitable handler, go to calling statement.
Iterate.
If you reach the top level, abort the program.
Any begin-end block can have an exception handler:

```ada
procedure test is
    x : integer := 25;
    y : integer := 0;
    begin
        x := x/y;
        exception
            when Constraint_Error =>
                Put_Line(“Divided by Zero”)
            when others => Put_Line(“Other bug”)
    end
```

function Get_Data return integer is
X: Integer
begin
  loop
    begin Get(X);
    return X;
    exception
      when others =>
        Put_Line("Input must be integer");
    end; end loop;
end;
An exception is a declared identifier, with usual scoping rules.

package Stacks is
    Stack_Empty: exception;
package body Stacks is
    procedure Pop(X : out integer;
        From: in out Stack)
    begin if Empty(From)
        then raise Stack_Empty; ...
Built in package “Ada.occurrence” defines an “exception occurrence” as a type that can hold information like location of occurrence, stack contents, etc.

Built in function “Save_Occurrence” saves this information. Executed as part of handler.
 Exceptions are classes.
 Handlers appear in try blocks

```cpp
try
{
    ComplexCalculation();
}
catch (range_error) { cerr << “range error\n”; }
catch (zero_divide)
{
    cerr << “why is x zero?\n”; }
catch (…) { cerr << “What’s going on?”; }
“…” is the actual C++ code here.
```
A program throws an object. The declaration does not indicate it will be used as an exception.

class selfIntersectErr {
    Polygon P; // useful information
    selfIntersectErr () { …} // constructor

    { … if (P.SelfIntersects())
        throw selfIntersectErr(); … }
A handler names a class and can handle an object of a derived class.
Exception Handling - Exceptions in Java

- Similar to C++
  - Exceptions are objects that are thrown and caught.
  - Try blocks have handlers, which are examined in succession.
  - A handler can handle an object of a derived class.

- Differences
  - All exceptions are part of predefined class Throwable.
  - “finally” clause is executed at the end of a “try” block, whether normally executed or handled by a handler.
  - Method must declare all exceptions that it may pass onto a caller.
public void replace (String name, Object newvalue) throws NoSuch
{
    Attribute attr = find(name);
    if (attr==null) throw new NoSuch(name);
    ...
}

Caller must either have a handler for NoSuch or be declared as throwing it. Compiler checks.
(Hence awkwardness of reading in Java.)
Exception Handling - Implementation

Simple implementation:
Stack of handlers and subroutine calls.
Each handler indicates exceptions it handles and address of statement after try block.
When enter “try” block add handlers to stack in backward order.
When call routine, push call (part of calling sequence).
When exit, pop them.
global variable E;  // raised exception
while (HandlerStack is not empty) {
    H = pop(HandlerStack);
    if (H handles E) {
        execute body of H;
        E = null;
        jump to return address of H; }
    if (H is a subroutine) execute epilogue of H;
}
abort program;
Exception Handling - In epilogue of each subroutine

Just before jump to return address of caller add:

if (E != null) jump to exception handler;
The above procedure requires handlers to be enqueued and dequeued each time a try block is entered. Large runtime cost for rarely used functionality.

Scott describes a cleverer implementation with no runtime cost until exception is actually raised.
For exceptions based on hardware interrupts (signal from I/O device, divide by zero, overflow, underflow, etc.), the interrupt handler in the OS has to transfer control and information to the exception handler in the runtime system. 

I don’t know how this works.
Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name.

Coroutines can be used to implement:
- iterators (Section 6.5.3 of the textbook)
- threads (to be discussed in Chapter 12 of the textbook)

Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack.
Figure 8.6 A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)
In order to understand recursion, you must first understand recursion.

Recursion is when a subroutine is called from within itself.

```c
int fact(int n)
{
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

What are some advantages and disadvantages of using recursion?

- Advantages: often conceptually easier, and easier to understand code
- Disadvantages: usually slower, can lead to stack overflow

There is one case when recursion can be implemented without using a stack frame for every call:

- A tail recursive subroutine is one in which no additional computation ever follows a recursive call.
- For tail recursive subroutines, the compiler can reuse the current activation record at the time of the recursive call, eliminating the need to allocate a new one.
Agenda

1. Session Overview
2. Subprograms
3. Conclusion
Assignments & Readings

- **Readings**
  - Chapter Sections 6.6, 8-1-8.3

- **Assignment #3**
  - Due on July 3, 2014
Next Session: Functional Programming