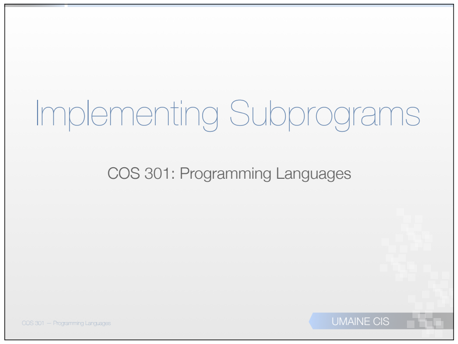
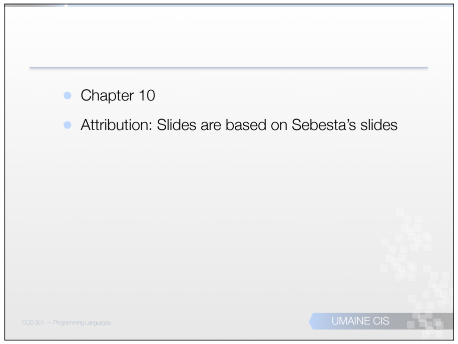
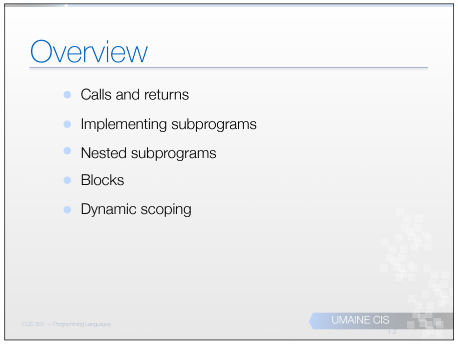


Implementing Subprograms

COS 301: Programming Languages

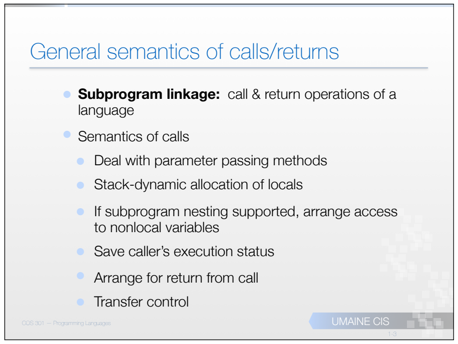


- Chapter 10
- Attribution: Slides are based on Sebesta's slides



Overview

- Calls and returns
- Implementing subprograms
- Nested subprograms
- Blocks
- Dynamic scoping



General semantics of calls/returns

- **Subprogram linkage:** call & return operations of a language
- Semantics of calls
  - Deal with parameter passing methods
  - Stack-dynamic allocation of locals
  - If subprogram nesting supported, arrange access to nonlocal variables
  - Save caller's execution status
  - Arrange for return from call
  - Transfer control

General semantics of calls/returns

- Semantics of returns:
  - Out mode and in-out mode parameters → return their values
  - Arrange for return value (if any)
  - Deallocate stack-dynamic locals
  - Restore execution status of caller
  - Return control to caller

Storage required by subprograms

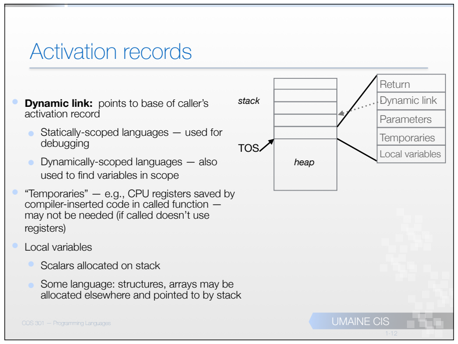
- Return value, status information
- Parameters
- Return address
- Locals
- Any temporary storage needed (e.g., by code inserted by the compiler to hold CPU registers of caller, etc.)

Implementing subprograms

- Two parts of a subprogram:
  - code
  - non-code: local variables, anything that can change
- Non-code ⟹ **activation record**
- For languages with no stack-dynamic variables:
  - No recursion, so ⟹ only one activation record active at once
  - Can allocate a single activation record per subprogram
- With dynamic variables:
  - Multiple invocations ⟹ multiple **activation record instances** (**stack frames**)
  - Typically stored on the runtime stack
  - ⟹ compiler has to add code to allocate/deallocate AR

Activation records

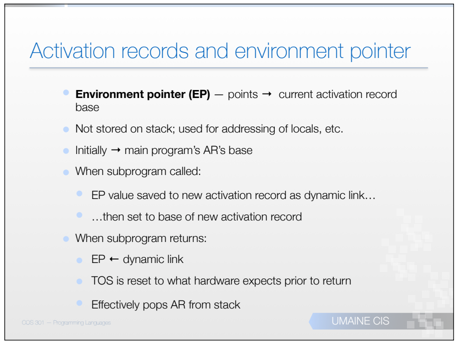
- Format of AR is static
  - For some languages, size is static
  - For others size may be dynamic — for example, if variably-sized arrays are allowed
  - E.g.: in Ada, size of local array can vary based on parameter
- AR instance: created when a subprogram is called



Activation records

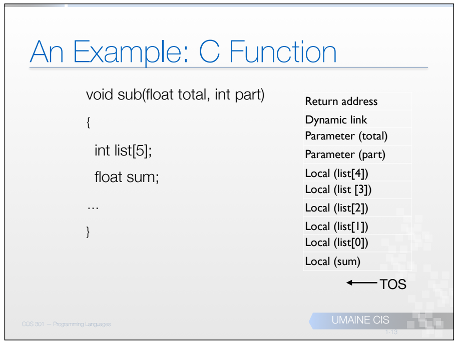
- **Dynamic link:** points to base of caller's activation record
  - Statically-scoped languages — used for debugging
  - Dynamically-scoped languages — also used to find variables in scope
- "Temporaries" — e.g., CPU registers saved by compiler-inserted code in called function — may not be needed (if called doesn't use registers)
- Local variables
  - Scalars allocated on stack
  - Some language: structures, arrays may be allocated elsewhere and pointed to by stack

---



Activation records and environment pointer

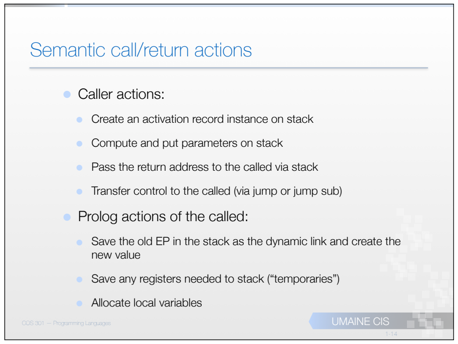
- **Environment pointer (EP)** — points → current activation record base
- Not stored on stack; used for addressing of locals, etc.
- Initially → main program's AR's base
- When subprogram called:
  - EP value saved to new activation record as dynamic link…
  - …then set to base of new activation record
- When subprogram returns:
  - EP ← dynamic link
  - TOS is reset to what hardware expects prior to return
  - Effectively pops AR from stack



An Example: C Function

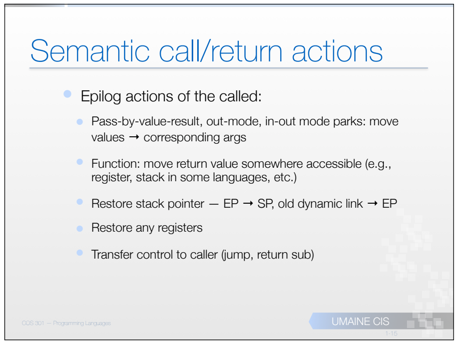
```
void sub(float total, int part)
{
 int list[5];
 float sum;
…
}
```

Return address
Dynamic link
Parameter (total)
Parameter (part)
Local (list[4])
Local (list [3])
Local (list[2])
Local (list[1])
Local (list[0])
Local (sum)



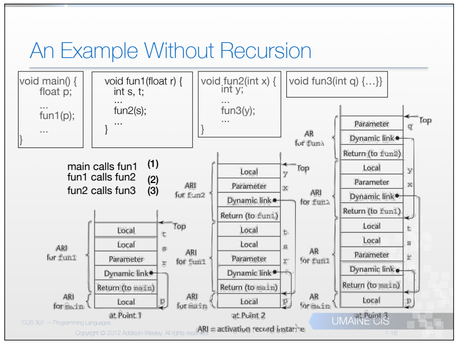
Semantic call/return actions

- Caller actions:
  - Create an activation record instance on stack
  - Compute and put parameters on stack
  - Pass the return address to the called via stack
  - Transfer control to the called (via jump or jump sub)
- Prolog actions of the called:
  - Save the old EP in the stack as the dynamic link and create the new value
  - Save any registers needed to stack ("temporaries")
  - Allocate local variables



Semantic call/return actions

- Epilog actions of the called:
  - Pass-by-value-result, out-mode, in-out mode parks: move values → corresponding args
  - Function: move return value somewhere accessible (e.g., register, stack in some languages, etc.)
  - Restore stack pointer — EP → SP, old dynamic link → EP
  - Restore any registers
  - Transfer control to caller (jump, return sub)



An Example Without Recursion

```
void main() {
    float p;
    ...
    fun1(p);
    ...
}
```

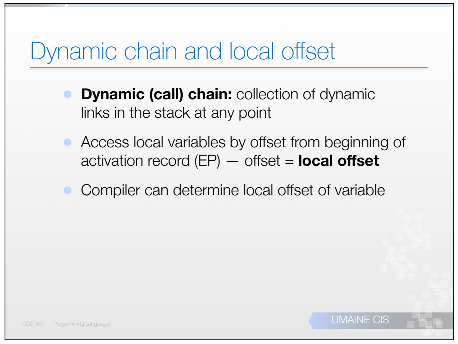
```
void fun1(float r) {
  int s, t;
  ...
  fun2(s);
  ...
}
```

```
void fun2(int x) {
  int y;
  ...
  fun3(y);
  ...
}
```

```
void fun3(int q) {...}}
```

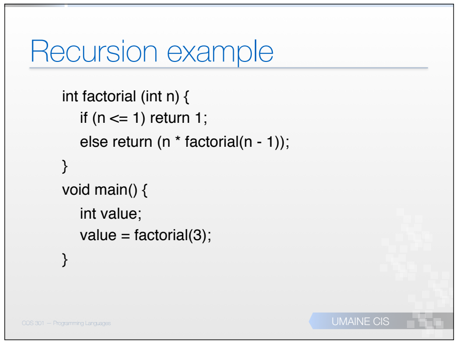
main calls fun1 (1)
fun1 calls fun2 (2)
fun2 calls fun3 (3)

ARI = activation record instance



Dynamic chain and local offset

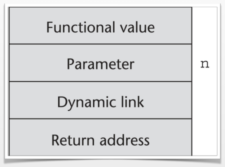
- **Dynamic (call) chain:** collection of dynamic links in the stack at any point
- Access local variables by offset from beginning of activation record (EP) — offset = **local offset**
- Compiler can determine local offset of variable



Recursion example

```
int factorial (int n) {
    if (n <= 1) return 1;
    else return (n * factorial(n - 1));
}
void main() {
    int value;
    value = factorial(3);
}
```

Activation record for **factorial**



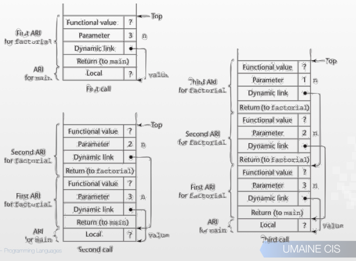
Functional value

Parameter n

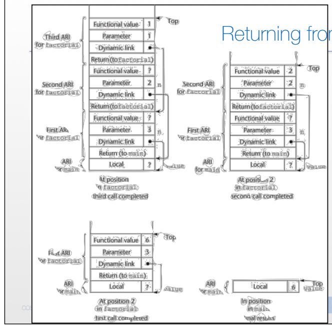
Dynamic link

Return address

Stacks for calls to factorial



Returning from factorial



Implementing dynamic scoping

1. How would you find non-local variables: be precise!

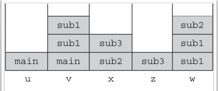
2. Cool. Now think of another way.

Implementing dynamic scoping

1. **Deep access:**
   - Nonlocal references found by searching ARIs in dynamic chain
   - Length of chain not known ahead of time
   - Every ARI has to have variable names (unlike static scoping)
2. **Shallow access:**
   - Put all locals in central place
   - Create a separate stack for each variable name
   - Central table maps variable names to stacks

Implementing dynamic scoping with shallow access

```
void sub3() {
  int x, z;
   x = u + v;
       …
      }
void sub2() {
    int w, x;
       …
      }
void sub1() {
    int v, w;
       …
      }
void main() {
    int v, u;
       …
      }
```



(The names in the stack cells indicate the program units of the variable declaration.)

Static scoping

- Some non-C-based, static-scoped languages (e.g., Fortran 95+, Ada, Python, JavaScript, Ruby, and Lua) use stack-dynamic local variables and allow subprograms to be nested
- Variables in enclosing subprograms can be accessed — in scope of enclosed subprogram
- These are local variables ⟹ reside in some AR
- How to locate?
  - Find the right AR instance
  - Find the correct offset in the AR instance

Locating a non-local reference

- Finding the offset: local offset in correct AR instance
- But how to find the correct AR instance?
  - Lexical scoping: only **static ancestors'** variables can be accessed
  - Nested subprograms ⟹ static ancestors' AR is somewhere on the stack already
  - Can't just search back through the stack (as per dynamic scoping) for a name
  - Have to find the static ancestor, not any intervening ARs that have the same variable name

Static scoping

- **Static link:** a (new) entry in an activation record → instance of static parent's AR — also: **static scope pointer**
- **Static chain** — static links connecting static ancestors of an AR instance
- To find non-local reference → traverse static chain
- Compiler can determine how far back to search based on a scope's **static_depth** — how deeply it's nested in the outermost scope

---

Static scoping

- Compiler/interpreter can determine the **chain_offset** (**nesting_depth**)
  - How far back in the chain to find the nonlocal reference
  - chain_offset = static_depth - (static_depth of scope in which it was declared)
- References to variables can be represented by:

(chain_offset, local_offset)

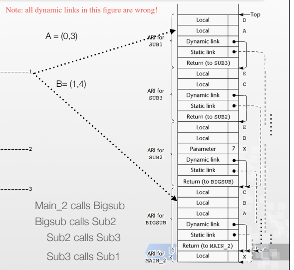
where local_offset = offset of var in AR instance at chain_offset

---

Example Ada Program

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C;  <-----------------------1
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        Sub1;
        E := B + A:   <--------------------2
        end; -- of Sub3
      begin -- of Sub2
      Sub3;
      A := D + E;  <-----------------------3
      end; -- of Sub2 }
    begin -- of Bigsub
    Sub2(7);
    end; -- of Bigsub
  begin
  Bigsub;
end; of Main_2 }
```

Note: all dynamic links in this figure are wrong!

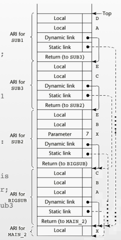


Main_2 calls Bigsub
Bigsub calls Sub2
Sub2 calls Sub3
Sub3 calls Sub1

Static chain maintenance

- On return —
  - Nothing need be done
  - Static chain of remaining stack frames is still valid

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin -- of Sub1
      A := B + C;
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin -- of Sub3
        Sub1;
        E := B + A:
```

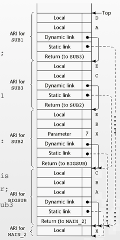


2 static links back

Static chain maintenance

- On call —
  - Dynamic link in AR instance — just old environment pointer (EP) , TOS → EP
  - Static link → most recent ARI of static parent
  - But how to find it?
    1. Search the dynamic chain; or
    2. Treat subprogram calls like variable references/definitions
       - Compiler: determine nesting_depth of called relative to caller
       - Store this, use it during call
       - Works — unless caller is subprogram passed as parameter

```
procedure Main_2 is
 X : Integer;
 procedure Bigsub is
  A, B, C : Integer;
  procedure Sub1 is
   A, D : Integer;
   begin -- of Sub1
   A := B + C;
  end;  -- of Sub1
  procedure Sub2(X : Integer) is
   B, E : Integer;
   procedure Sub3 is
    C, E : Integer;
    begin -- of Sub3
    Sub1;
    E := B + A:
```



2 static links back

Static chain problems

- Nonlocal reference can be slow if nesting depth large —usually not too bad, since few nonlocal refs
- What to do with subprograms as parameters?
- Difficult to determine timing
  - Cost of nonlocal references difficult to determine
  - Nesting depth can change when code is changed ⟹ different timing
  - May need to know timing for realtime or time-critical applications

Blocks

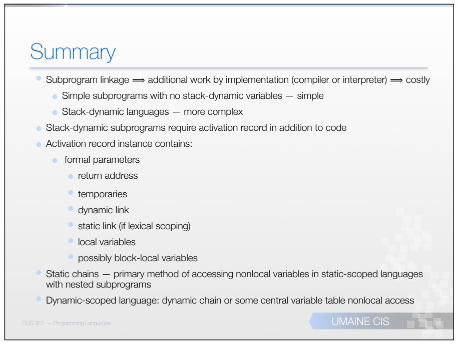
- *Blocks:* user-defined local scopes
- Example (C):

```
{int temp;
 temp = list [upper];
 list [upper] = list [lower];
 list [lower] = temp
}
```

- Temp's storage is created on block entry, goes away on exit
- Advantage of blocks:
  - variables declared within won't clash with other names elsewhere in program
  - only use storage as needed

Implementing blocks

- One way:
  - Treat blocks as parameterless subprograms — always "called" from same location
  - Each block has activation record (stack frame)
  - Instance is created each time block is executed
  - But costly to create activation records!
- Another way:
  - Compiler knows max storage required for block
  - Allocate that amount of space after local vars in activation record



Summary

- Subprogram linkage ⟹ additional work by implementation (compiler or interpreter) ⟹ costly
    - Simple subprograms with no stack-dynamic variables — simple
    - Stack-dynamic languages — more complex
- Stack-dynamic subprograms require activation record in addition to code
- Activation record instance contains:
    - formal parameters
        - return address
        - temporaries
        - dynamic link
        - static link (if lexical scoping)
        - local variables
        - possibly block-local variables
- Static chains — primary method of accessing nonlocal variables in static-scoped languages with nested subprograms
- Dynamic-scoped language: dynamic chain or some central variable table nonlocal access