Binding, Scope & Storage Allocation

CS2210
Lecture 9

Reading

- Chapter 7 except 7.9
- Muchnick: 1-3
  - Just skim, should be nothing new, just different notation

Binding, Scope & Storage

1) Converted input into tokens w/ attribues
2) Parsed token sequence and created abstract syntax tree using semantic actions
3) Typechecked abstract syntax tree
4) To generate (intermediate) code
   1) Have to map names to storage locations
   2) Have to generate code to access data at run time

Binding

- Binding association of attributes with actual values
  - Eg. Storage location of variable x with its location in memory (global, local, heap)
- The time when each of these occurs in a program is the binding time of the attribute
  - Static binding: occurs before runtime and does not change during program execution
  - Dynamic binding: occurs during runtime or changes during runtime

Binding

- Name:
  - Occurs when program is written (chosen by programmer) and for most languages will not change: static
- Address and Lifetime:
  - A memory location for a variable must be allocated and deallocated at some point in a program
  - The lifetime is the period between the allocation and deallocation of the memory location for that variable

Lifetimes

- Static: bound to same memory cell for entire program
  - Ex: static C++ variables
- Stack-dynamic: bound and unbound based on run-time stack
  - Ex: Pascal, C++, Ada subprogram variables
- Explicit heap-dynamic: nameless cells that can only be accessed using other variables (pointers / references)
  - Allocated explicitly by the programmer
  - Ex: result of new in C++ or Java

Lifetimes (2)

- Implicit heap-dynamic variables
  - Binding of all attributes (except name) changed upon each assignment
  - Much overhead to maintain all of the dynamic information
    - Used in Algol 68 and APL
    - Not used in most newer languages except some functional languages like ML
- Value
  - Dynamic by the nature of a variable – can change during run-time

Binding & Types

- Type
  - Dynamic Binding
    - Type associated with a variable is determined at run-time
    - A single variable could have many different types at different points in a program
  - Static Binding
    - Type associated with a variable is determined at compile-time (based on var. declaration)
    - Once declared, type of a variable does not change

Scope

- Scope = determines what names are visible at program points
  - Scope ~ where
  - Lifetime ~ when
- Static scope
  - determined at compile-time
- Dynamic scope
  - determined at run-time
- Choice determines
  - What non-locals are visible
  - How are visible locals accessed

Static Scope

- Most modern languages use static scope
  - If variable is not locally declared, proceed out to the "textual parent" (static parent) of the block/subprogram until the declaration is found
  - Fairly clear to programmer – can look at code to see scope

Dynamic Scope

- Non-local variables are accessed via calls on the run-time stack (going from top to bottom until declaration is found)
  - A non-local reference could be declared in different blocks in different runs
  - Used in APL, SNOBOL4 and through **local** variables in Perl

Dynamic Scope (cont.)

- Flexible but very tricky
  - Difficult for programmer to anticipate different definitions of non-local variables
  - Type-checking must be dynamic, since types of non-locals are not known until run-time
  - More expensive to implement

Scope, Lifetime and Referencing Environments

- More often they are not "the same"
  - Lifetime of stack-dynamic variables continues when a subsequent function is called, whereas scope does not include the body of the subsequent function
  - Lifetime of heap-dynamic variables continues even if they are not accessible at all

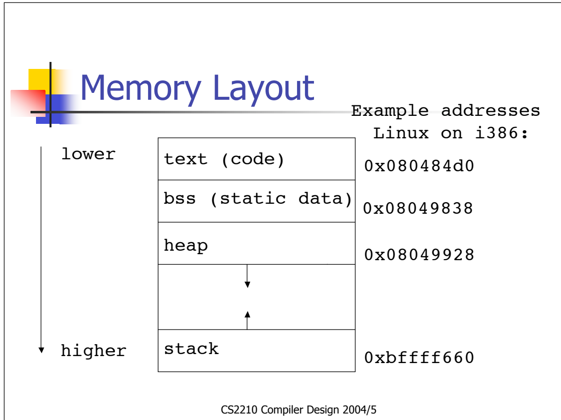
Scope, Lifetime and Referencing Environments

- Referencing Environment
  - Given a statement in a program, what variables are visible there?
  - Depends on the type of scoping used
  - Once we know the scope rules, we can always figure this out
    - From code if static scoping is used
    - From call chains (at run-time) if dynamic scoping is used

Procedure Lifetimes

- Activation
  - Execution of a procedure
  - Active until procedure returns
  - Multiple activations of same procedure possible (recursion)

Memory Layout



CS2210 Compiler Design 2004/5

Activation Records

- Data structure containing information needed by a single execution of a procedure - aka (stack) frame
- Contained fields depend on the language and calling convention of the target architecture

CS2210 Compiler Design 2004/5

Frame Format

return value
actuals
dynamic link
static link
machine status
locals
temporaries

- Dynamic link = control link
  - Pointer to frame of caller
- Static link = access link
  - Pointer to frame of immediately enclosing scope

CS2210 Compiler Design 2004/5

Alpha Processor Format

octaword-aligned

Stack frame	Offset
[fixed temporary locations]	: 0 (from SP)
register save area	: PDSC_RPD_RSA_OFFSET (from SP)
[fixed temporary locations]	
[argument home area]	
[arguments passed in memory]	: PDSC_RDP_FRAME_SIZE (from SP)

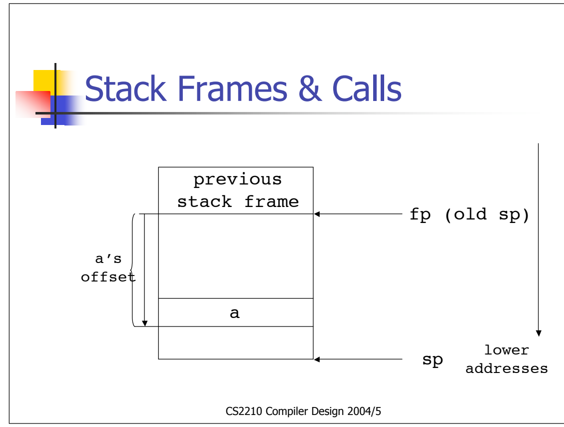
Alpha Calling Convention

- Most arguments passed in registers
  - r16 … r21 on the Alpha
- Return value passed in return register
  - r0 on the Alpha, f0 for floating point values
- Frame pointer register
  - Used if frame size can vary at run time
  - $15 on the Alpha
- Return address register (address of instruction following call instruction)
  - $26 on the Alpha

MIPS Calling Convention

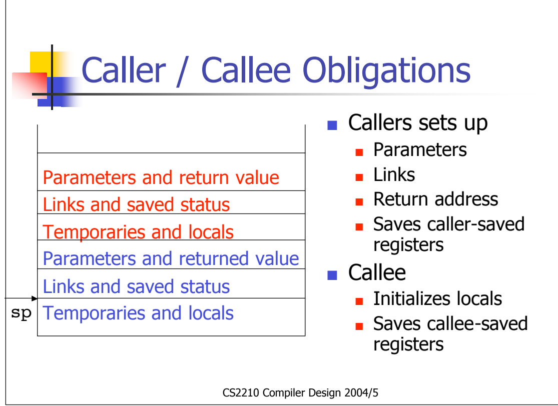
- $a0 … $a3 argument registers
- $t0 … $t9 are temporary registers (caller-saved)
- $ra for return address

Stack Frames & Calls



CS2210 Compiler Design 2004/5

Caller / Callee Obligations



- Callers sets up
  - Parameters
  - Links
  - Return address
  - Saves caller-saved registers
- Callee
  - Initializes locals
  - Saves callee-saved registers

CS2210 Compiler Design 2004/5

Access to Non-locals

- Simple for globals (including C's "static")
- Variables in enclosing scopes:
  - At compile time we know # of levels (=:n)
  - At run-time set up static links = pointer to AR of textually enclosing procedure/scope
  - For run-time lookup: traverse n static links to find AR, use offset from frame pointer

CS2210 Compiler Design 2004/5

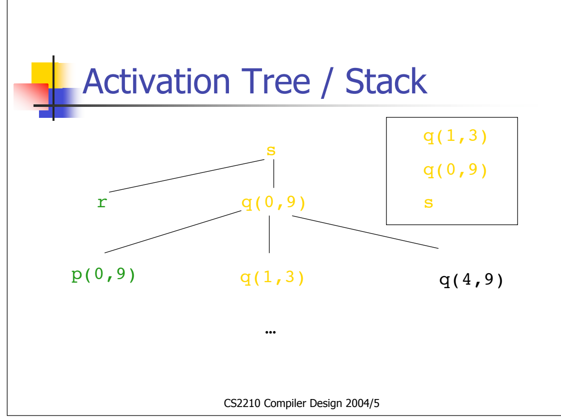
How do we set up the static link?

p calls q, nesting depth n_p and n_q :

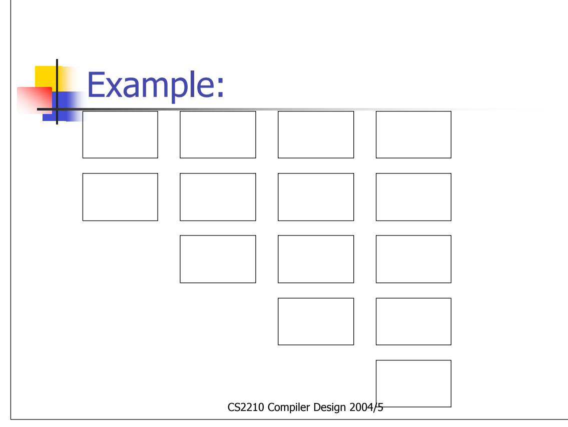
- $n_p < n_q$:
  - q must be declared within p
  - Static link points to immediate caller
- $n_p >= n_q$:
  - Traverse n_p - n_q links to find AR of procedure at level n_q, then use that AR's static link,
    - i.e., we find the most recent AR of q's immediate enclosing by following n_p - n_q +1 static links


```
program sort(input, output);
  var a : array [0..9] of integer;
     x      : integer;

    procedure readarray;
      var i :  integer;
      begin {read array a} end {readarray};

    procedure exchange(i, j     :  integer);
     begin
       x := a[i]; a[i] = a[j]; a[j] = x
    end {exhange};

    procedure quicksort(m,n     :  integer);
       var k, v :  integers;
       function partition(y,z :  integer) : integer;
          var i,j :  integer;
           begin
                {stuff}
                exchange(i,j);
         end {partition};

    begin
         k = partition(m,n); quicksort(m+1,k);
         quicksort(k+1,n);
    end; { quicksort }

begin
    readarray;quicksort(0,9)   {sort}
end.
{sort}
```


Activation Tree / Stack



Example:



CS2210 Compiler Design 2004/5

Static Link Evaluation

- Good
  - Little work at call
    - Only add one static link
  - Access to locals remains fast
- Bad
  - Expensive access to non-locals if many levels are involved

CS2210 Compiler Design 2004/5

Displays

- Alternative to static links
  - Maintain a single array that has pointers to all relevant activation records
  - Have to update array at every call & return
- Pros & Cons
  - Fast access to non-locals
  - Higher maintenance cost
    - Pays off for high nesting levels & many non-local accesses

CS2210 Compiler Design 2004/5

Dynamic Scoping

- Two implementation methods
  - Deep access
  - Shallow access
- Important: both implement the same semantic model
  - Deep and shallow binding are different concepts
    - Refer to the referencing environment of procedural parameters:
    - Deep binding = environment of definition of passed proc.
    - Shallow binding = environment of the calling proc.

Referencing environment

```
Procedure sub1;
    var x:integer;
    procedure sub2;
    begin
       write('x = ', x);
    end;
Procedure sub3;
    var x: integer;
    begin;
       x := 3;
       sub4(sub2);
    end;
Procedure sub4(subx);
    var x: integer;
    begin
       x := 4;
       subx;
    end;
Begin
    x := 1;
    sub3;
End;
```

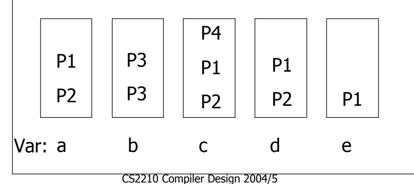
- 3 binding options
  - Shallow
    - Where proc is CALLED (dynamic)
  - Deep
    - Where proc is DEFINED (static)
  - Ad hoc
    - Where proc is PASSED

Deep Access

- Use dynamic link for name lookup
  - May traverse many links, hence "deep access"
  - Have to store variables with their names
    - Can't use offset method because # of traversed links unknown at compile time
      - More expensive than static scoping

Shallow access

- Analogue of displays in the static scope world
- Use a stack for each variable, e.g.:



Parameter Passing

- Three semantic classes (semantic models) of parameters
  - IN: pass value to subprogram
  - OUT: pass value back to caller
  - INOUT: pass value in and back
- Implementation alternatives
  - Copy value
  - Pass an access path (e.g. a pointer)

Parameter Passing Methods

- Pass-by-Value
- Pass-by-Reference
- Pass-by-Result
- Pass-by-Value-Result (copy-restore)
- Pass-by-Name

Pass-by-value

- Copy actual into formal
  - Default in many imperative languages
    - Only kind used in C and Java
  - Used for IN parameter passing
  - Actual can typically be arbitrary expresion including constant & variable

Pass-by-value cont.

- Advantage
  - Cannot modify actuals
    - So IN is automatically enforced
- Disadvantage
  - Copying of large objects is expensive
    - Don't want to copy whole array each call!
- Implementation
  - Formal allocated on stack like a local variable
  - Value initialized with actual
    - Optimization sometimes possible: keep only in register

Pass-by-result

- Used for OUT parameters
  - No value transmitted to subprogram
  - Actual MUST be variable (more precisely lvalue)
    - foo(x) and foo(a[1]) are fine but not foo(3) or foo(x * y)

Pass-by-result gotchas

```
procedure foo(out int x, out int
    y) {
    g := 4;
    x := 42;
    y := 0;
}
main() {
b: array[1..10] of integer;
g: integer;
g = 0;
call to foo:
}
```

- foo(a,a); print(a) what is printed?
- foo(b[g], …): which element is modified?

Pass-by-value-result

- Implementation model for in-out parameters
- Simply a combination of pass by value and pass by result
  - Same advantages & disadvantages
  - Actual must be a lvalue

Pass-by-reference

- Also implements IN-OUT
  - Pass an access path, no copy is performed
- Advantages:
  - Efficient, no copying, no extra space
- Disadvantages
  - Parameter access usually slower (via indirection)
  - If only IN is required, may change value inadvertently
  - Creates aliases

Pass-by-reference aliases

```
int  g;
void foo(int& x) {
   x = 1;
}
foo(g);
```

- g and x are aliased

Pass-by-name

- Textual substitution of argument in subprogram
  - Used in Algol for in-out parameters
  - evaluated at each reference to formal parameter in subprogram
  - Subprogram can change values of variables used in argument expression
  - Programmer must rename variables in subprogram in case of name clashes
  - Evaluation uses reference environment of caller

Jensen’s device

```
real procedure sigma(x, i, n);
  value n;
  real x; integer i, n;
begin
  real s;
  s := 0;
  for i := 1 step 1 until n do
    s := s + x;
  sigma := s;
end
```

What does sigma(a(i), i, 10) do?

Design Issues

- Typechecking
  - Are procedural parameters included?
  - May not be possible in independent compilation
  - Type-loophole in original Pascal: was not checked, but later procedure type required in formal declaration

CS2210 Compiler Design 2004/5

Pass-by-name Safety Problem

```
procedure swap(a, b);
    integer a,b,temp;
    begin
       temp := a;
       a := b;
       b := temp;
end;
swap(x,y):

swap(i, x(i))
```

CS2210 Compiler Design 2004/5

Call-by-name Implementation

- Variables & constants easy
  - Reference & copy
- Expressions are harder
  - Have to use parameterless procedures aka. Thunks

CS2210 Compiler Design 2004/5

Thunk Example

```
real procedure sigma(x, i, n);
  value n;
  real x; integer i, n;
begin
  real s;
  s := 0;
  for i := 1 step 1 until n do
    s := s + x;
  sigma := s;
end
```

```
sigma(a(i)*b(i), i, 10)
function iThunk() :IntVarAddress;
begin
    iThunk := ADDRESS(I)
end;
function xiThunk()
    :RealVarAddress;
var expi;
begin
    expi := a(I)*b(I);
    xiThunk := ADDRESS(expi);
end;
```

Thunk Evaluation

```
real procedure sigma(x, i, n);
  value n;
  real x; integer i, n;
begin
  real s;
  s := 0;
  for i := 1 step 1 until n do
    s := s + x;
  sigma := s;
end
function iThunk() :IntVarAddress;
begin
    iThunk := ADDRESS(I)
end;
function xiThunk() :RealVarAddress;
var expi;
begin
    expi := a(I)*b(I);
    xiThunk := ADDRESS(expi);
end;
```

```
real procedure sigma(xiThunk(),
    iThunk(), 10);
begin
  real s;
  s := 0;
  for iThunk()^ := 1 step 1
   until n do
    s := s + xiThunk()^;
  sigma := s;
end
```

Procedures as Parameters

- In some languages procedures are first-class citizens, i.e., they can be assigned to variables, passed as arguments like any other data types
- Even C, C++, Pascal have some (limited) support for procedural parameters
- Major use: can write more general procedures, e.g. standard library in C:
  `qsort(void* base, size_t nmemb, size_t size, int(*compar)(const void*, const void*));`

Design Issues

- Typechecking
  - Are procedural parameters included?
  - May not be possible in independent compilation
  - Type-loophole in orignal Pascal: was not checked, but later procedure type required in formal declaration

Referencing environment

```
Procedure sub1;
    var x:integer;
    procedure sub2;
    begin
       write('x = ', x);
    end;
Procedure sub3;
    var x: integer;
    begin;
       x := 3;
       sub4(sub2);
    end;
Procedure sub4(subx);
    var x: integer;
    begin
       x := 4;
       subx;
    end;
Begin
    x := 1;
    sub3;
End;
```

- 3 binding options
  - Shallow (aka activation environment)
    - Where proc is CALLED (dynamic)
  - Deep (aka lexical environment)
    - Where proc is DEFINED (static)
  - Ad hoc (aka passing environment)
    - Where proc is PASSED

Procedures as parameters

- How do we implement static scope rules?
  - = how do we set up the static link?

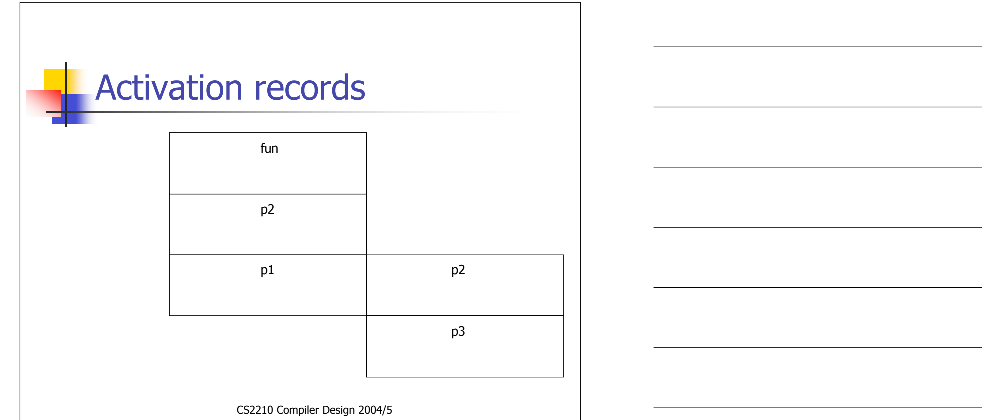
```
program param(input, output);

procedure b(function h(n : integer):integer);
  begin writeln(h(2)) end {b};

procedure c;
var m : integer;

   function f(n :  integer) : integer;
   begin f := m + n end {f};

begin m: =  0; b(f) end {c};

begin
   c
end.
```

Solution: pass static link:

param
c
b

Another Example

```
program fun;
procedure p1;
begin {...} end; {p1}

procedure p2(procedure x);
var n :  integer;

   procedure p3;
   begin n = 0; end; {p3}

begin x; p2(p3); end; {p2}

begin {main} p2(p1) end. {main}
```


Activation records
fun
p2
p1
p2
p3
CS2210 Compiler Design 2004/5