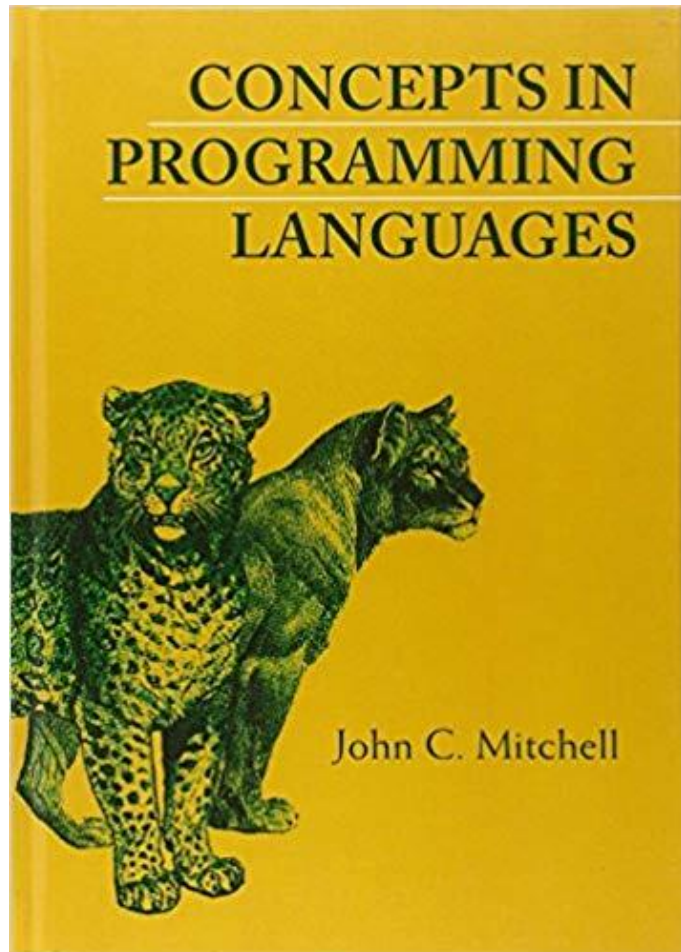


Lecture 9: Scope, Functions, and Storage Management – Part I

Mitchell chapter 3.4 & 7



CS-3160 – Spring 2024

M/W @ 8:00AM - 9:15 PM

Room: Centennial Hall 191

Instructor: Adham Atyabi

Office: Engineering 194

Office Hours: Mon 12:30 PM-13:30 PM

Email: aatyabi@uccs.edu

Teaching Assistant: Raisa Nusrat
(rnusrat@uccs.edu)

Name, Scope, and Binding

- A name is exactly what you think it is
 - Usually think of identifiers but can be more general
 - symbols (like '+' or '_') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually) in which the binding is active

Binding

- Binding Time is the point at which a binding is created
 - language design time
 - *program structure, possible type*
 - language implementation time
 - *I/O, arithmetic overflow, stack size, type equality (if unspecified in design)*
 - program writing time
 - *algorithms, names*
 - compile time
 - *plan for data layout*
 - link time
 - *layout of whole program in memory*
 - load time
 - *choice of physical addresses*

Binding

- Implementation decisions (continued):
 - run time
 - *value/variable bindings, sizes of strings*
 - *subsumes*
 - program start-up time
 - module entry time
 - elaboration time (point at which a declaration is first "seen")
 - procedure entry time
 - block entry time
 - statement execution time

Binding

- The terms Static and Dynamic are generally used to refer to things bound before run time and at run time, respectively
 - “static” is a coarse term; so is "dynamic"
- Binding times are very important in the design and implementation of programming languages

Binding

- In general, early binding times are associated with **greater efficiency**
- Later binding times are associated with **greater flexibility**
- Compiled languages tend to have **early binding times**
- Interpreted languages tend to have **later binding times**

The Lisp Abstract machine

- Abstract machine
 - The runtime system (software simulated machine) based on which a language is interpreted
 - In short, the internal model of the interpreter that implements the language
- Lisp Abstract machine has four parts:
 - A **Lisp expression**: the current expression to evaluate
 - A **continuation**: the rest of the computation
 - A **list** : variable->value mapping
 - A **set of cons cells** (dynamic memory, a heap)
- Garbage collection
 - Automatic collection of non-accessible cons cells

The Lisp Memory Model

- Each cons cell is a pair

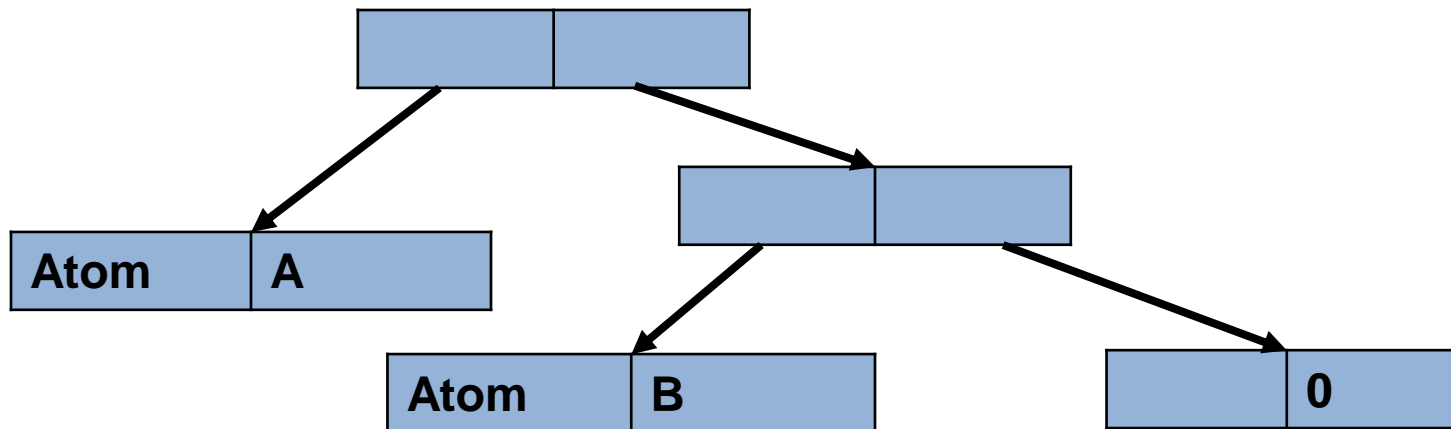
Address	Decrement
---------	-----------

– pointed to by pointers in A-list

- ***(car cdr) => linked data structures (lists)***
- ***(atom a) => a single atom***

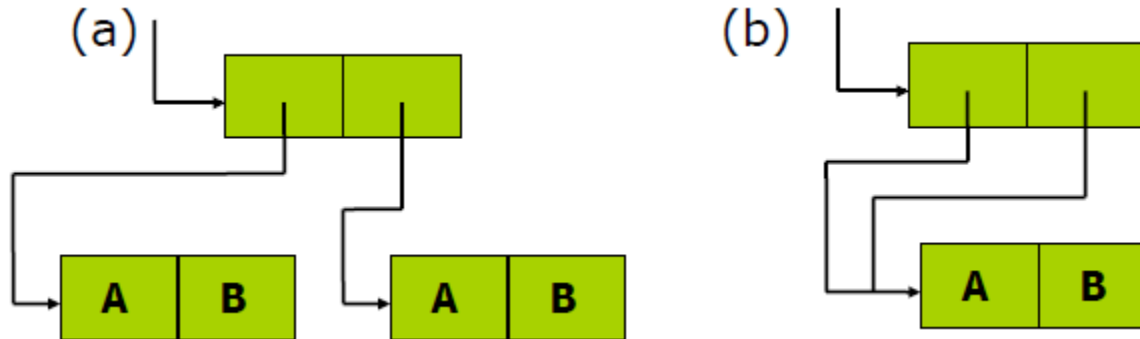
The Lisp Memory Model

- Atoms and lists represented by cells
 - Tag each value to remember its type



- There are five basic functions on cons cells which are **atom**, **eq**, **cons**, **car**, **cdr**.

Sharing

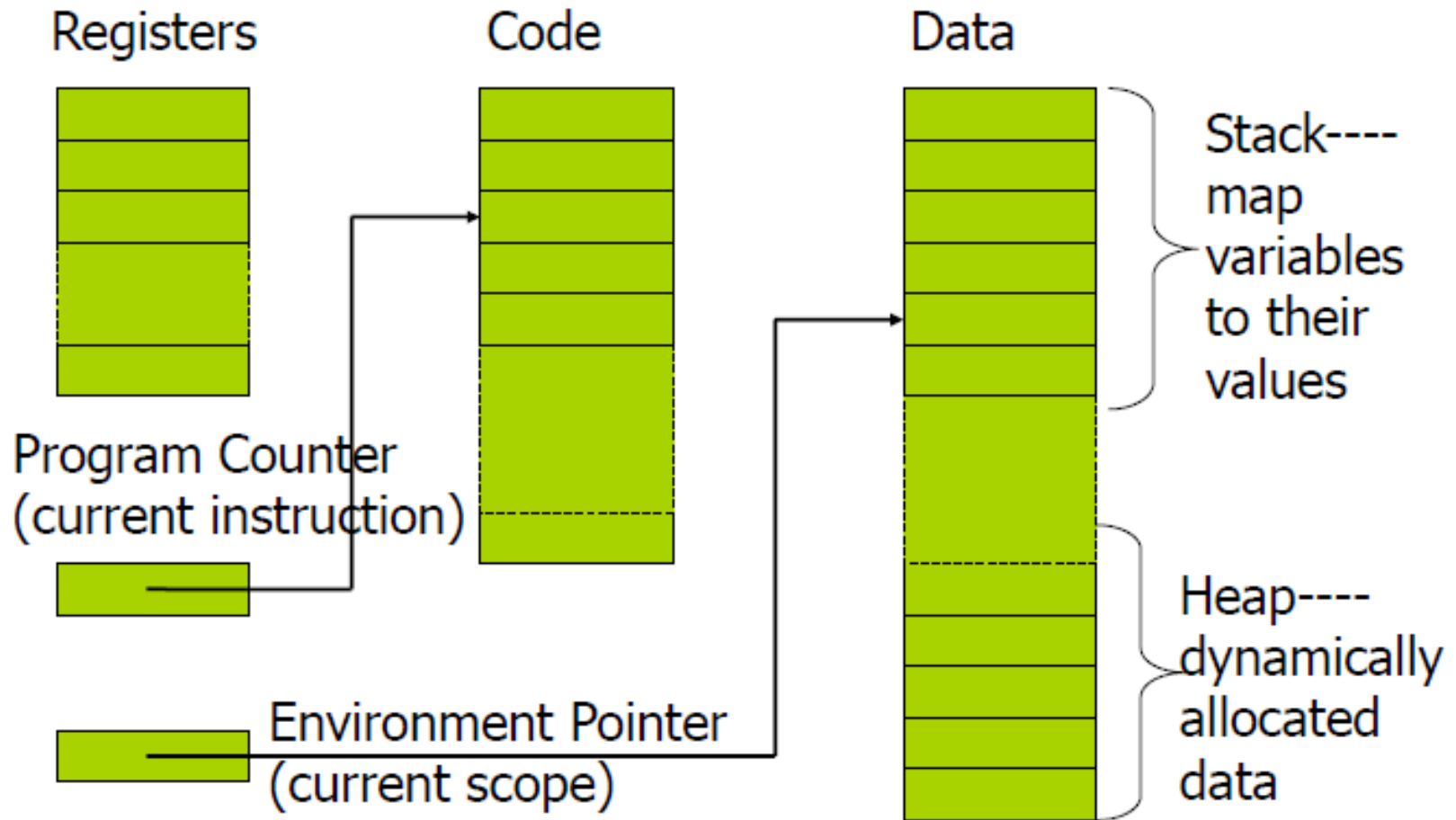


- Both structures could be printed **as (A.B).(A.B)**
- Which are the results of evaluating
 - **(cons (cons 'A 'B) (cons 'A 'B)) ?**
 - **((lambda (x) (cons x x)) (cons 'A 'B))**
- Equality of compound structures
 - What is the result of (eq? 'a 'a) ?
 - What is the result of (eq? '(a b) '(a b)) ?

Machine Model In Compiled Languages (Compare To List Abstract Machine)

- Components of Machine Model in compiled languages
 - Runtime stack
 - Heap
 - Code space
 - Program counter
 - Environment pointer
 - registers

Machine Model In Compiled Languages (Compare To List Abstract Machine)



Data Storage Management

- Runtime stack: mapping variables to their values
 - When introducing new variables: push new stores to stack
 - When variables are out of scope: pop outdated storages
- Heap: dynamically allocated data of varying lifetime
 - Variables that last throughout the program
 - Data pointed to by variables on the runtime stack
 - Target of garbage collection
- The code space: the whole program to evaluate
- Program counter: current/next instruction to evaluate
 - keep track of instructions being evaluated
- Environment pointer: current stack position
 - Used to keep track of storages of all active variables
- Registers: temporary storages for variables

Blocks in C/C++

```
outer block {  
    {  
        int x = 2;  
        {  
            int y = 3;  
            x = y+2;  
        }  
    }  
}
```

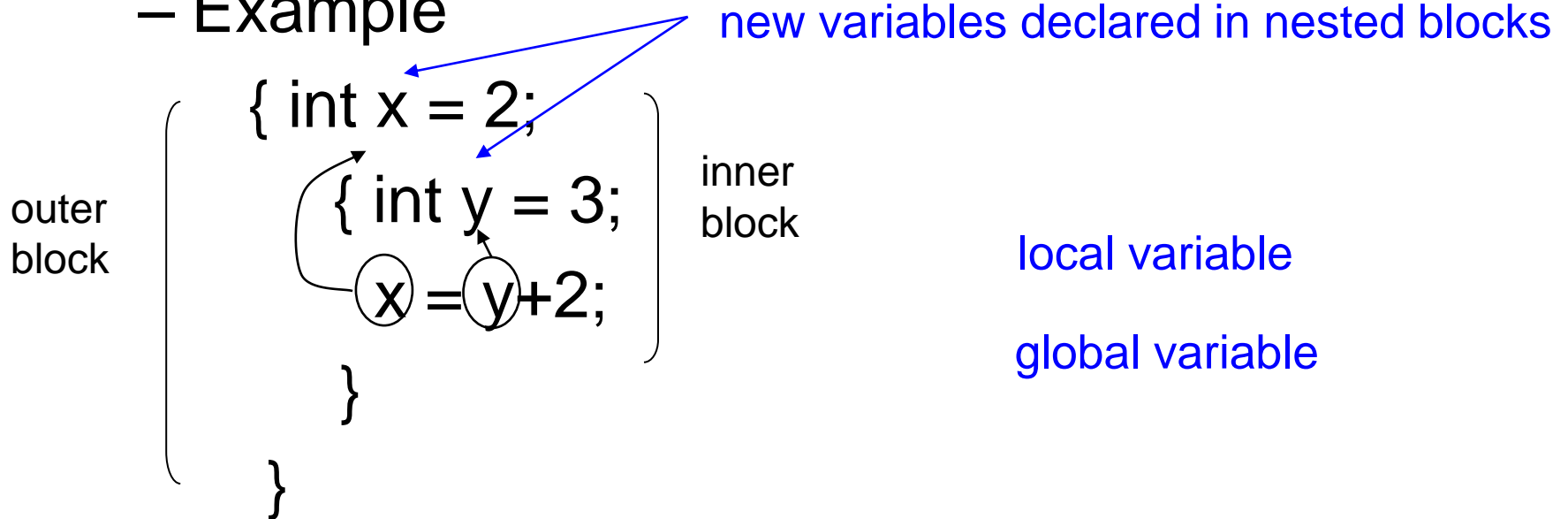
inner block

- Blocks: regions of code that introduces new variables
 - Enter block: allocate space for variables
 - Exits block: some or all space may be deallocated
- Blocks are nested but not partially overlapped
 - Jumping out of a block
 - ***Make sure variables are freed before exiting***
 - What about jumping into the middle of a block?
 - ***Variables in the block have not yet been allocated***

Block-Structured Languages

- Nested blocks, local variables

- Example



- Storage management

- ***Enter block: allocate space for variables***
 - ***Exits block: some or all space may be deallocated***

ML Nested Blocks

- Syntax: **let** <varDecls> **in** <exp> **end**

- Examples

let val x = 3; val y = 4 in x + y end;

let fun foo(x) = x + 1 in foo(4) end;

let val x = 3; val y = 4 in

let fun foo(x) = x + 1 in foo(x + y) end

end;

- Each let ... in ...end introduces a number of local variables (or functions)
 - These variables can be used only within the local expression
 - **NOTE**: function definitions are not evaluated until they are called with arguments

Blocks in Functional languages

- ML

```
let fun g(y) = y + 3
in
  let
    fun h(z) = g(g(z))
  in h(3)
  end
end;
```

- Lisp

```
( (lambda (g)
  ((lambda (h) (h 3)) (lambda (z) (g (g z)))))
  (lambda (y) (+ y 3)))
```

Summary of Blocks

- Blocks in common languages
 - C { ... }
 - Algol begin ... end
 - ML let ... in ... end
- Two forms of blocks
 - In-line blocks
 - Blocks associated with functions or procedures
- Topic: block-based memory management



University of Colorado
Colorado Springs

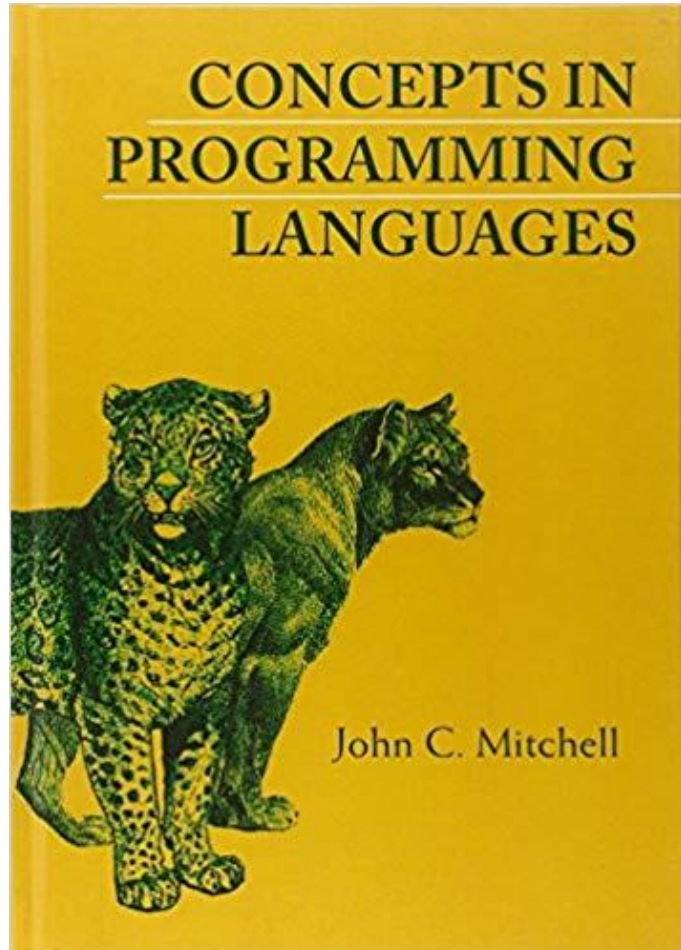


University of Colorado

Boulder | Colorado Springs | Denver | Anschutz Medical Campus

Lecture 9: Scope, Functions, and Storage Management – Part II

Mitchell chapter 3.4 & 7



CS-3160 – Spring 2024

M/W @ 8:00AM - 9:15 PM

Room: Centennial Hall 191

Instructor: Adham Atyabi

Office: Engineering 194

Office Hours: Mon 12:30 PM-13:30 PM

Email: aatyabi@uccs.edu

Teaching Assistant: Raisa Nusrat
(rnusrat@uccs.edu)

Lifetime and Storage Management

- Key events
 - creation of objects
 - creation of bindings
 - references to variables (which use bindings)
 - (temporary) deactivation of bindings
 - reactivation of bindings
 - destruction of bindings
 - destruction of objects
- The period of time from creation to destruction is called the **Lifetime** of a binding
 - If object outlives binding it's **garbage**
 - If binding outlives object it's a **dangling reference**
- The textual region of the program in which the binding is *active* is its scope

Storage Management mechanisms - Static

- Storage Allocation mechanisms
 - Static
 - Stack
 - Heap
- Static allocation is used for followings:
 - code
 - globals
 - static or own variables
 - explicit constants (including strings, sets, etc)
 - scalars may be stored in the instructions

Storage Management mechanisms - Static

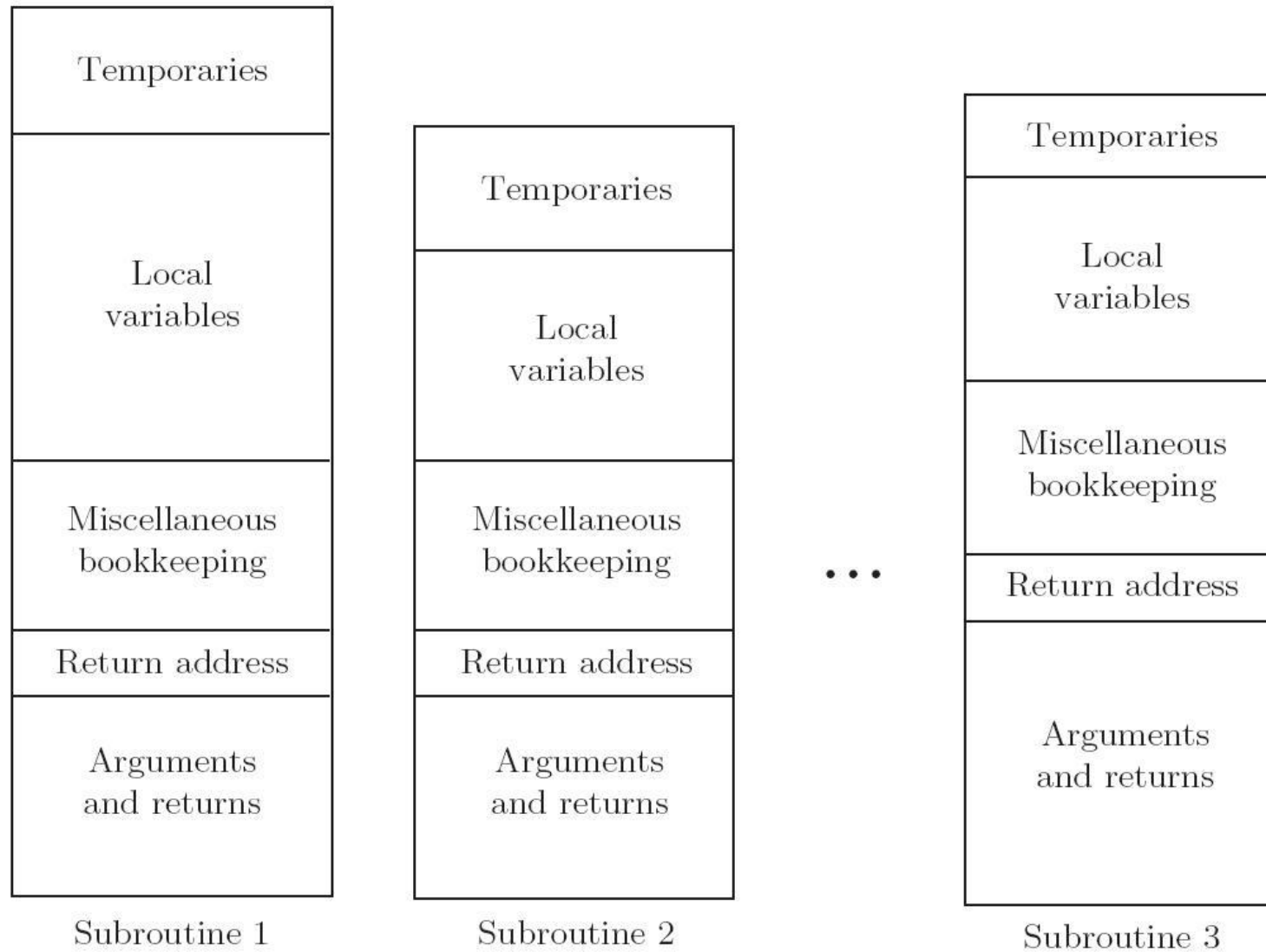


Figure 3.1: **Static allocation of space for subroutines in a language or program without recursion.**

Storage Management mechanisms - Stack

- Central stack for
 - parameters
 - local variables
 - temporaries
- Why a stack?
 - allocate space for recursive routines
(not possible in old FORTRAN – no recursion)
 - reuse space
(in all programming languages)

Storage Management mechanisms - Stack

- Contents of a stack frame
 - arguments and returns
 - local variables
 - temporaries
 - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed **offsets** from the stack pointer or frame pointer at compile time

Storage Management mechanisms - Stack

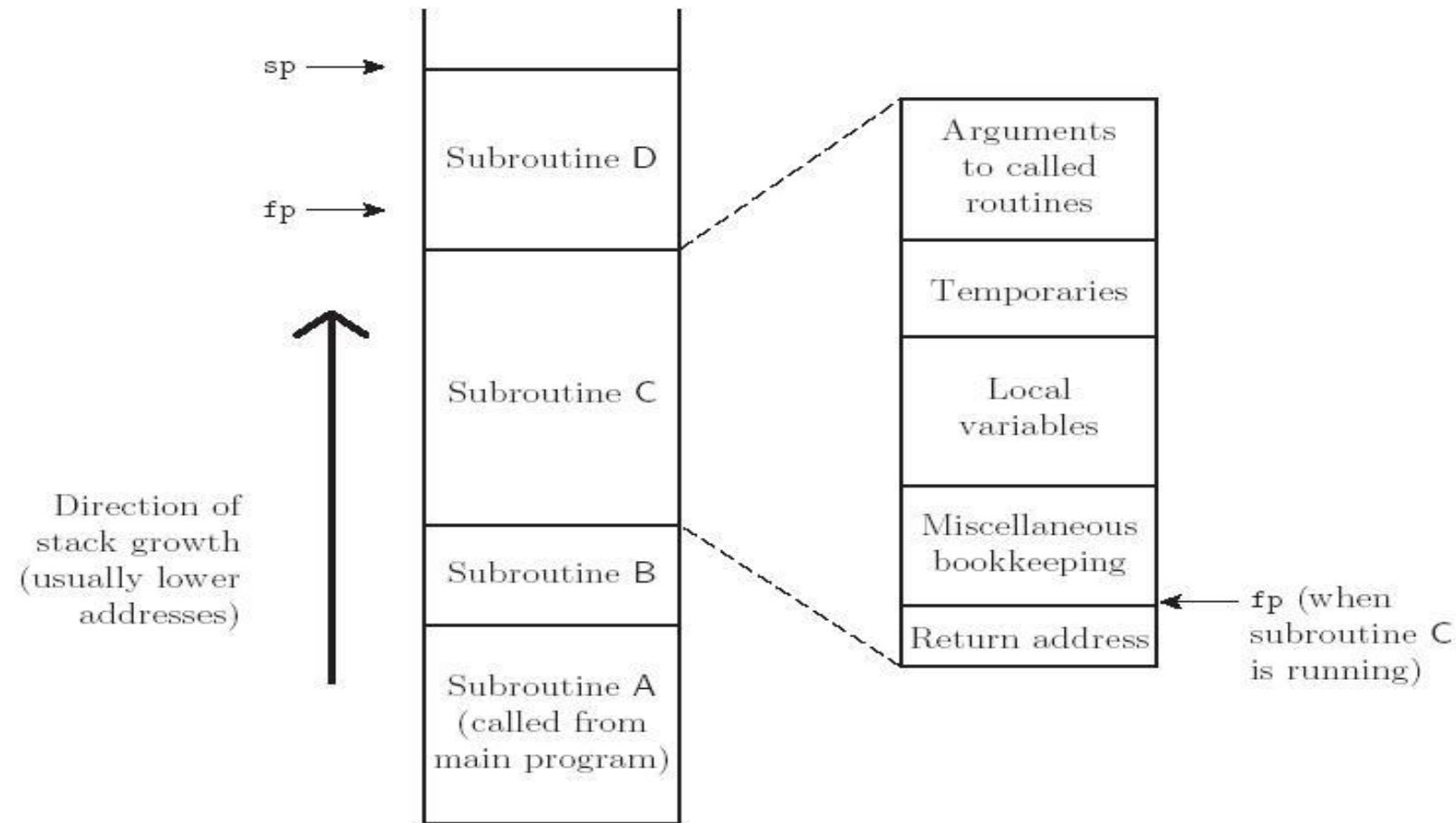


Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (**sp**) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (**fp**) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Example: Examine Stack for the C Program

```
int bar(int x)
{
    int z=5;
    return z;
}

int foo(int x)
{
    int y=3;
    x = x + y;
    y = bar(x);
    return x;
}

int main(int argc, char* argv[])
{
    int a=1, b=2, c=3;
    b = foo(a);
    printf("%d %d %d\n",a,b,c);
    return 0;
}
```

Storage Management mechanisms - Heap

- Region of memory where subblocks are allocated and deallocated dynamically
- More unstructured
- Allocation and deallocation may happen in arbitrary order
 - Memory may become fragmented
 - Need for garbage collection
 - We will describe garbage collection algorithms later
- Heap Management
 - Often managed with a single linked list – the free list – of blocks not in use
 - ***First Fit?***
 - ***Best Fit? (smallest block large enough to handle request)***

Storage Management mechanisms - Heap

- Allocation and deallocation may happen in arbitrary order
 - Memory may become fragmented
 - Need for garbage collection
 - We will describe garbage collection algorithms later
- Deallocation methods:
 - Programmer-controlled (e.g., in C using the free function)
 - **Memory leak**
 - **Dangling Pointer** → **type insecurities**
 - Automatic → garbage collection

Common techniques for garbage collection

- Common techniques for garbage collection
 - **Mark-Sweep**
 - **Problem:** Every object must have a mark bit. We need to keep a little extra memory available for performing the collection. The runtime of mark-sweep is linear in the heap size. Memory becomes fragmented.
 - **Copying**
 - **Problem:** *No mark bits. Copying can be slow for very large objects. Allocation is generally faster than mark-sweep, because free memory is contiguous, and the next free block is immediately available. The runtime is proportional to the amount of reachable memory. Need to allocate $2N$ memory for a heap of size N . Compaction is free.*
 - **Generational**
 - **Incremental**
 - **Reference Counting**
 - **Problem:** adds overhead to every pointer operation. It is, however, inherently incremental. It cannot collect cyclic structures.

Storage Management mechanisms - Heap

- Heap for dynamic allocation



Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Managing Data Storage In a Block

- **Local variables**
 - Declared inside the current block
 - ***Enter block: allocate space***
 - ***Exit block: de-allocate space***
- **Global variables**
 - Declared in a previously entered block
 - ***Already allocated before entering current Block***
 - ***Remain allocated after exiting current block***
- **Function parameters**
 - Input parameters
 - ***Allocated and initialized before entering function body***
 - ***De-allocated after exiting function body***
 - Return values
 - ***Address remembered before entering function body***
 - ***Value set after exiting function body***
- **Scoping rules**: where to find memory allocated for variables?
 - Need to find the block that introduced the variable

Scope Rules

- A *scope* is a program section of maximal size in which **no bindings change**, or at least in which **no re-declarations are permitted**
- In most languages with subroutines, we open a new scope on subroutine entry, e.g.,:
 - create bindings for new local variables,
 - deactivate bindings for global variables that are re-declared (these variable are said to have a "**hole**" in their scope)
 - make references to variables

Scope Rules

- On subroutine exit:
 - destroy bindings for local variables
 - reactivate bindings for global variables that were deactivated
- Algol 68:
 - **ELABORATION** = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
 - storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations

Scope Rules

- With static (LEXICAL) Scope Rules, a scope is defined in terms of the physical (lexical) structure of the program
 - The determination of scopes can be made by the compiler
 - All bindings for identifiers can be resolved by examining the program
 - Typically, we choose the most recent, active binding made at compile time
 - Most compiled languages, C++ and Java included, employ static scope rules

Scope Rules

- The classical example of static scope rules is the most closely nested rule used in block structured languages such as Algol 60 and Pascal
 - An identifier is known in the scope in which it is declared and in each enclosed scope, unless it is re-declared in an enclosed scope
 - To resolve a reference to an identifier, we examine the local scope and statically enclosing scopes until a binding is found

Scope Rules

- Note that the bindings created in a subroutine are **destroyed** at subroutine exit
- Obvious consequence when you understand how stack frames are allocated and deallocated
- The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime. This means that:
 - Bindings to variables declared in a module are inactive outside the module, not destroyed
 - The same sort of effect can be achieved in many languages with **own** (Algol term) or **static** (C term) variables

Scope Rules

- Access to non-local variables Static Links
 - Each frame points to the frame of the (correct instance of) the routine inside which it was declared
 - In the absence of formal subroutines, *correct* means closest to the top of the stack
 - You access a variable in a scope k levels out by following k static links and then using the known offset within the frame thus found

Scope Rules: Static Links a.k.a Static Chain

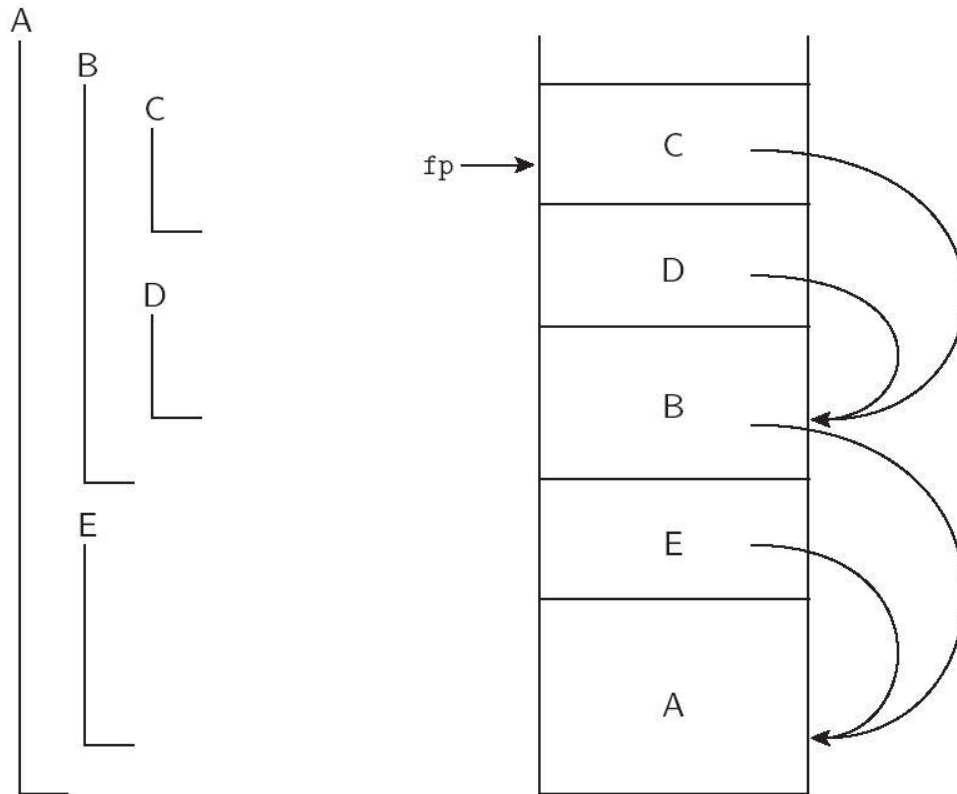


Figure 3.5: **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Scope Rules

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
 - They cannot always be resolved by examining the program because they are dependent on calling sequences
 - To resolve a reference, we use the most recent, active binding made at run time

static vs dynamic scope rules

```
int a
```

```
  proc first:
```

```
    a := 1
```

```
  proc second:
```

```
    int a
```

```
    first()
```

```
a := 2; second(); write(a)
```

Scoping rules

Finding non-local (global) variables

- Global and local variables

outer block	x	0
h(3)	z	3
	x	1
g(12)	z	3

Which x?

```
{ int x=0;  
  fun g(z) = x+z;  
  fun h(z) = let x = 1 in  
              g(z) end;  
  h(3)  
};
```

- Static scope
 - Find global declarations in the closest enclosing blocks in program text
- Dynamic scope
 - Find global variables in the most recently entered blocks at runtime

Binding of Referencing Environments

- In order to access variables with dynamic scope:
 - (1) keep a stack (*association list*) of all active variables
 - (2) keep a central table with one slot for every variable name

Binding of Referencing Environments

- In order to access variables with dynamic scope:
 - (1) keep a stack (*association list*) of all active variables
 - ***When you need to find a variable, hunt down from top of stack***
 - ***This is equivalent to searching the activation records on the dynamic chain***

Binding of Referencing Environments

- In order to access variables with dynamic scope:
 - (2) keep a central table with one slot for every variable name
 - ***If names cannot be created at run time, the table layout (and the location of every slot) can be fixed at compile time***
 - ***Otherwise, you'll need a hash function or something to do lookup***
 - ***Every subroutine changes the table entries for its locals at entry and exit (push / pop on a stack).***

Binding of Referencing Environments

- (1) gives you slower access but fast calls
- (2) gives you slower calls but fast access
- In effect, variable lookup in a dynamically-scoped language corresponds to symbol table lookup in a statically-scoped language
- Because static scope rules tend to be more complicated, however, the data structure and lookup algorithm also have to be more complicated

Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
 - early LISP dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

Scope Rules

Example: Static vs. Dynamic

```
int a;  
void first()  
{  
    a = 1;  
}  
void second()  
{  
    int a = 3;  
    first();  
}
```

```
void main()  
{  
    a = 2;  
    second;  
    printf("%d\n", a);  
}
```


Scope Rules

Example: Static vs. Dynamic

- Static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable `a`
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding at run time
 - Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines

```
int a;
void first()
{   a = 1;}
void second()
{   int a = 3;
    first();
}
```

```
void main()
{
    a = 2;
    second;
    printf("%d\n", a);
}
```

Binding of Referencing Environments

- Referencing environment of a statement at run time is the set of active bindings
- A referencing environment corresponds to a collection of scopes that are examined (in order) to find a binding
- SCOPE RULES determine the collection and its order
- First-class status: objects that can be **passed** as parameters or **returned** and **assigned**
- Second-class status: objects that can be **passed** but not **returned** or **assigned**
- Some programming languages allow subroutines to be first-class

Binding within a Scope

- Aliasing
 - Two or more names that refer to a single object in a given scope are aliases
 - What are aliases good for?
 - *space saving - modern data allocation methods are better*
 - *multiple representations*
 - *linked data structures*
 - Also, aliases arise in parameter passing
 - *Sometimes desirable, sometimes not*

Aliases

```
int i;  
int &ri = i;
```

Java:

```
public static void foo(MyObject x)  
{  
    x.val = 10;  
}  
  
public static void main(String[] args)  
{  
    MyObject o = new MyObject(1);  
    foo(o);  
}
```

Aliases

```
int i;  
int &ri = i;
```

pointers are *explicit* references.
references are *implicit* pointers.

```
(&i == &ri)
```

```
const Location& p = irresponsibly_long_object_name.retrieve_location();
```

```
void increment(int& parameter) { ++parameter; }
```

```
int main(int argc, char** argv) {  
    int variable = 0;  
    increment(variable); }
```

```
void increment(int* parameter) { ++*parameter; }
```

```
int main(int argc, char** argv) {  
    int variable = 0;  
    increment(&variable); }
```

Parameter passing

- Each function have a number of formal parameters
 - At invocation, they are matched against actual parameters
- **Pass-by name**
 - Rename each occurrence of formal parameter with its actual parameter --- delay of evaluation
 - Used in Lambda calculus and side-effect free languages
- **Pass-by-value**
 - Replace formal parameter with value of its actual parameter
 - Callee cannot change values of actual parameters
- **Pass-by-reference**
 - Replace formal parameter with address of its actual parameter
 - Callee can change values of actual parameters
 - Different formal parameters may have the same location

Example: What is the final result?

pseudo-code

```
int f (int x)
{
  x := x+1; return x;
};
main() {
  int y = 0;
  print f(y)+y;
}
```

pass-by-ref



Standard ML

```
fun f (x : int ref) =
  ( x := !x+1; !x );
val y = ref 0 : int ref;
f(y) + !y;
```

pass-by-value



```
fun f (z : int) =
  let val x = ref z in
    x := !x+1; !x
  end;
val y = ref 0 : int ref;
f(!y) + !y;
```

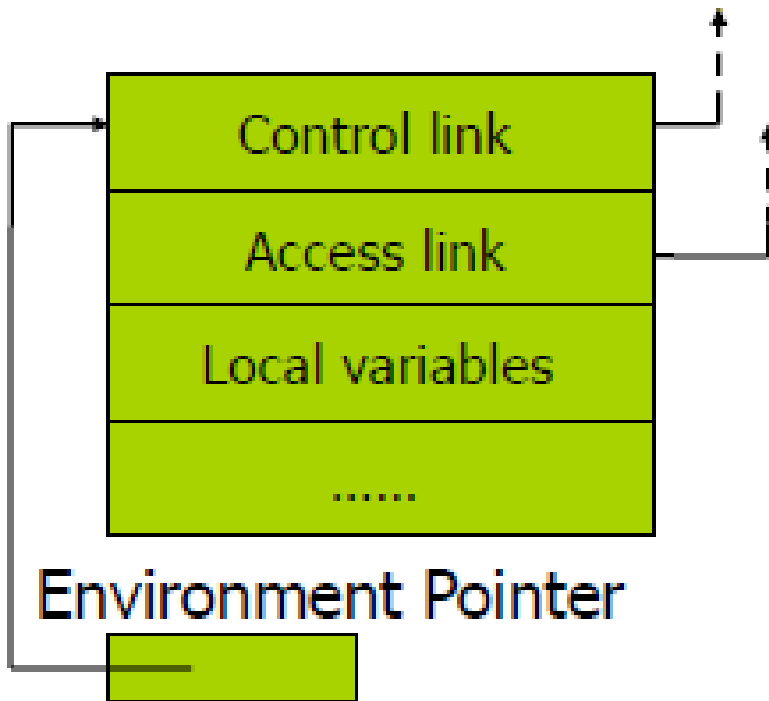
Managing Blocks

- Activation record: memory storage for each block
 - Contains values for local variables in the block
- Managing Activation Records (AR)
 - Allocated on a runtime stack: First-In-Last-Out
 - Before evaluating each block, push its activation record onto runtime stack; after exiting the block, pop its activation record off stack
 - Compilers generate instructions for pushing & popping of activation records (pre-compute their sizes)
- When is a block activated (evaluated)?
 - When evaluating a variable declaration
 - ***first instruction of an inline block***
 - When making a function call
 - ***NOT when evaluating the function declarations***

Finding Variables

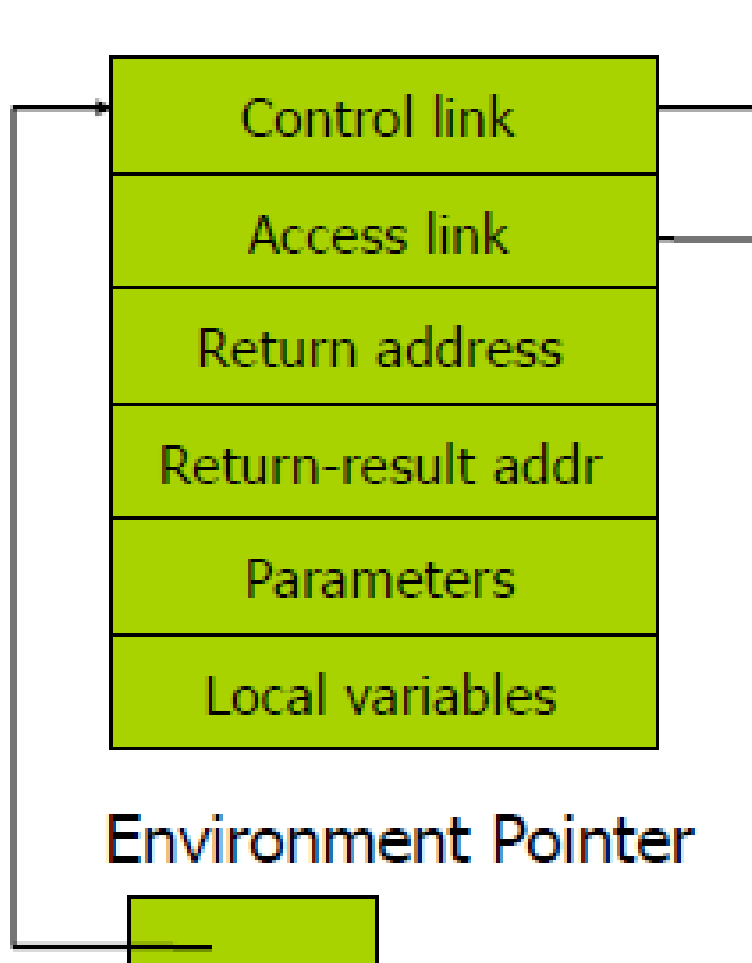
- Goal: find the memory allocated to the variable
 - The activation record that contains the variable
 - ***Determined at runtime***
 - The offset of the variable within the AR
 - ***Determined by the compiler before runtime***
 - Location = activation record pointer + offset
- Dynamically find activation record of introducing block
 - Compile-time: determine the lexical level n of the variable (the nesting level of its block)
 - Runtime: follow access link $m-n$ times, where m is the lexical level of the AR on top of the runtime stack

Activation Record For Inline Blocks



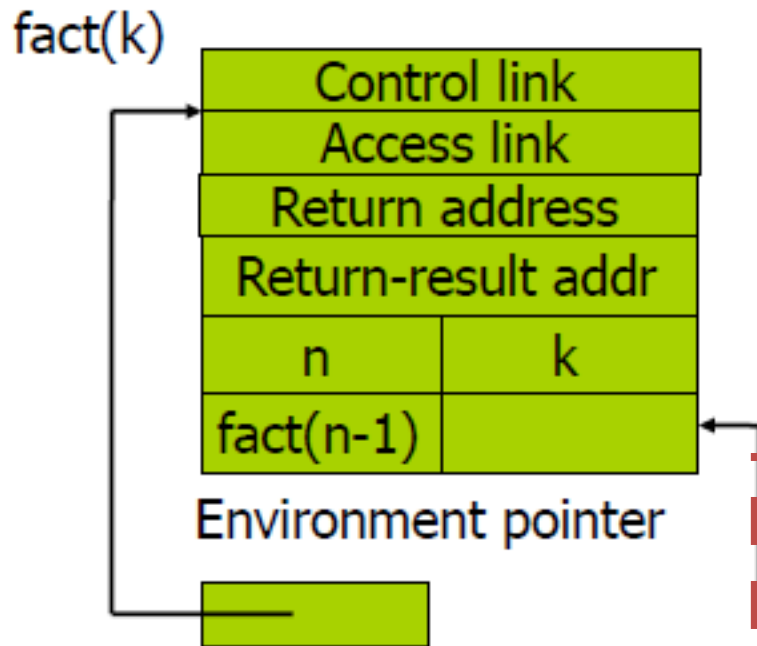
- Control link
 - Point to activation record of previous (calling) block
 - Depend on runtime behavior
 - Support push/pop of ARs
- Access link
 - Point to activation record of immediate enclosing block
 - Depend on static form of program
- Push record on stack
 - Set new control link to env ptr
 - Set env ptr to new record
- Pop record off stack
 - Follow current control link to reset environment pointer

Activation Records For Functions



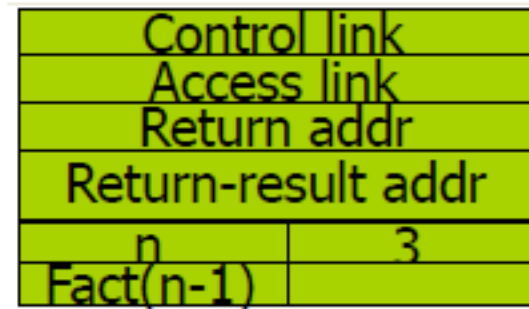
- Return address
 - Where to continue execution after return
 - Pointer to the next instruction following the function call
- Return-result address
 - Where to put return result
 - Pointer to caller's activation record
- Parameters
 - Values for formal parameters
 - Initialized with the actual parameters

Example: Function Calls

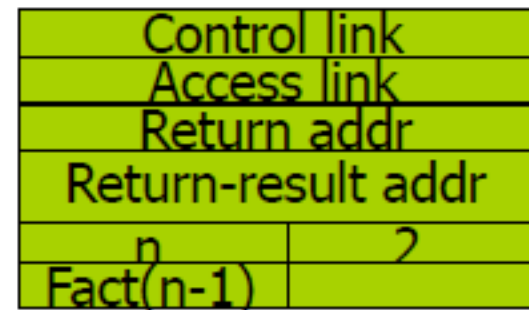


fact(n) = if $n \leq 1$ then 1
 else $n * \text{fact}(n-1)$

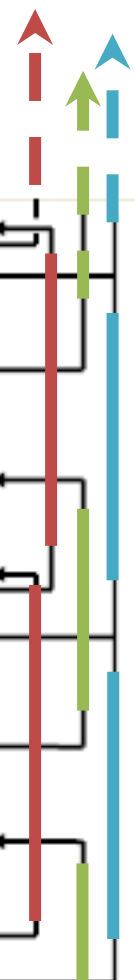
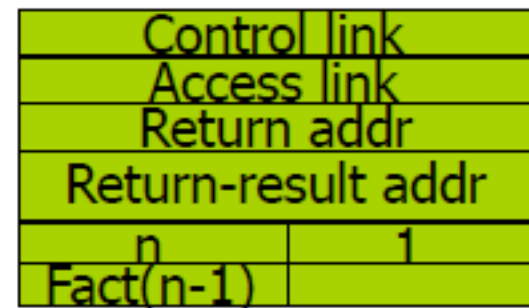
fact(3)



fact(2)



fact(1)



Function Abstraction As Values

```
1  let val x=1;  
2      fun g(z) = x+z;  
3      fun h(z) =  
4          let val x = 2 in  
5              g(z) end  
6      in h(3)  
7  end;
```

- What are values for g,h?
- How to determine their access links?
 - Inlined blocks
Access link = control link
 - Function blocks
Enclosing block of the function definition

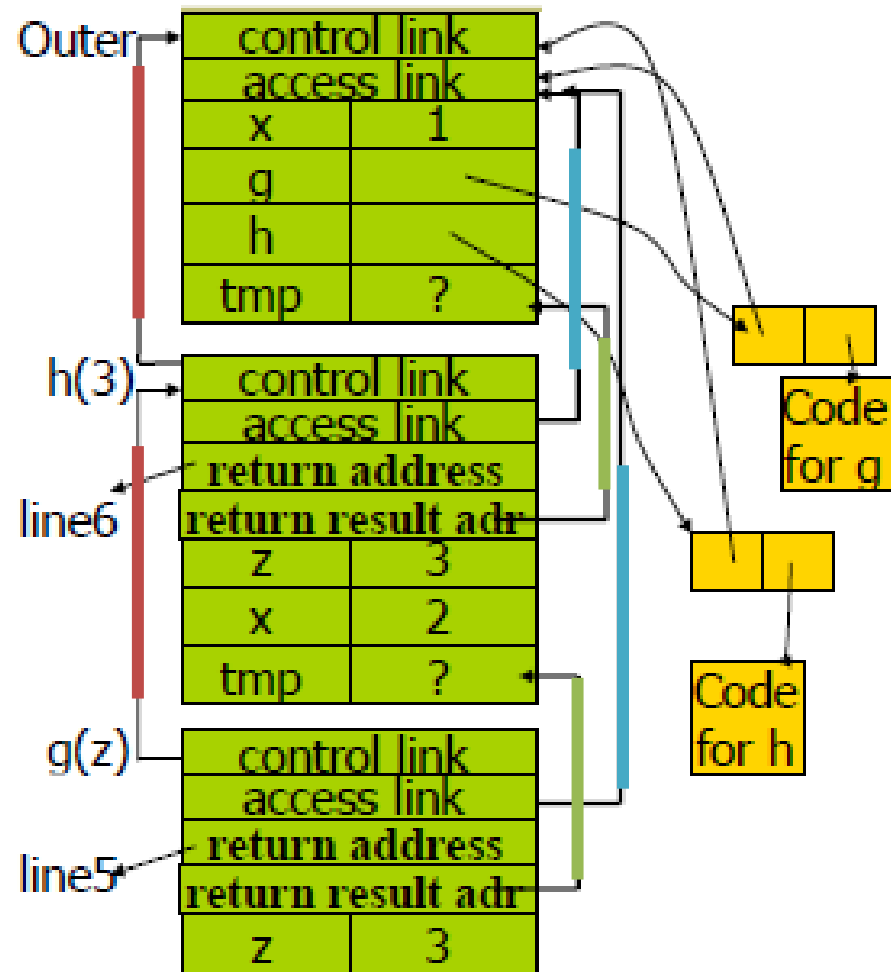
Closures

- A function value is a closure: (env, code)
 - code: a pointer to the function body
 - env: activation record of the enclosing block
- Use closure to maintain a pointer to the static environment of a function body
 - When called, set access link from closure
- When a function is called,
 - Retrieve the closure of the function
 - Push a new activation record onto runtime stack
 - Set return address, return value addr, parameters
 - Set access link to equal to the env pointer in closure
 - Start the next instruction from code pointer in closure

Function Abstraction As Values

```

1  let val x=1;
2    fun g(z) = x+z;
3  fun h(z) =
4    let val x = 2 in
5      g(z) end
6  in h(3)
7  end;
    
```



Return Function as Result


- Language feature: functions that return new functions
 - E.g. `fun compose(f,g) = (fn x => g(f x));`
 - Each function value is a closure = (env, code), where code may contain references to variables in env
 - Code is not “created” dynamically (static compilation)
- Use a closure to save the runtime environment of function
 - Environment: pointer to enclosing activation records
 - But the enclosing activation record may have been popped off the runtime stack
 - Returning functions as results is not allowed in C
 - *Just like returning pointers to local variables*
- Need to extend the standard “stack” implementation
 - Put activation records on heap
 - Invoke garbage collector as needed
 - Not as crazy as it sounds

Tail Call And Tail Recursion

- A function call from g to f is a **tail call**
 - if g returns the result of calling f with no further computation
- Example

tail call

not a tail call



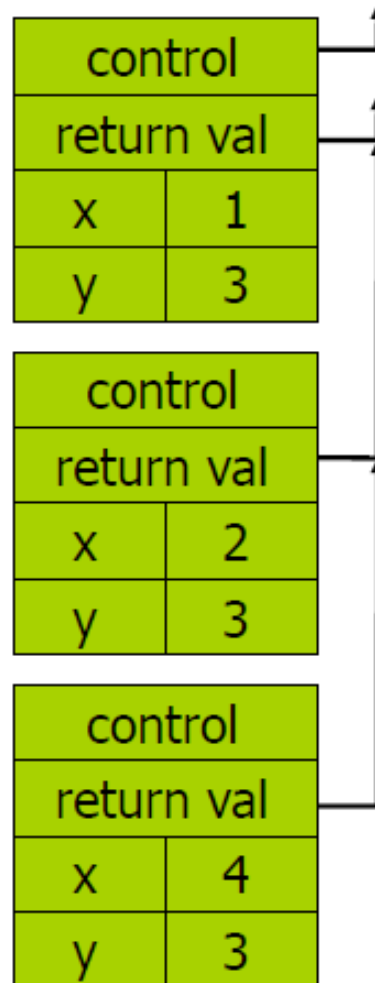
 - fun g(x) = if x>0 then f(x) else f(x)*2
- Optimization
 - Can pop activation record of caller on a tail call
 - Especially useful for a **tail recursive call** (f to f)
 - ***Callee's activation record has exactly same form***
 - ***Callee can reuse activation record of the caller***
 - ***A sequence of tail recursive calls can be translated into a loop***

Example: what is the result?

$f(1,3)$

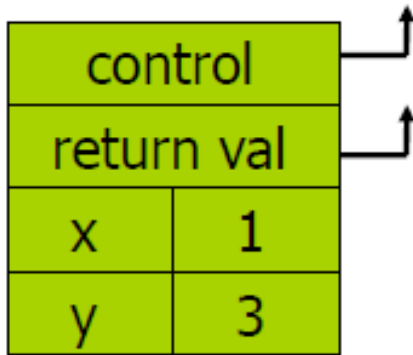
control		↑
return val		
x	1	↑
y	3	

```
fun f(x,y) = if x>y  
  then x  
  else f(2*x, y);  
f(1,3) + 7;
```

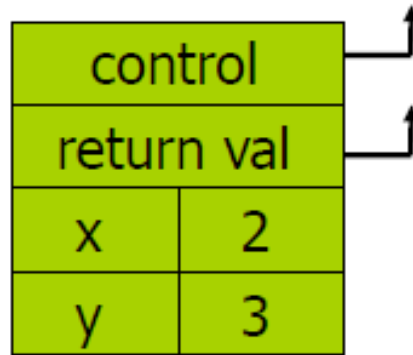


Tail recursion elimination

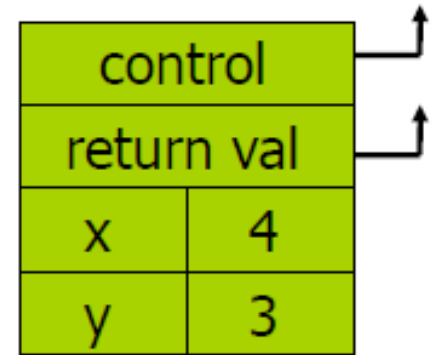
$f(1,3)$



$f(2,3)$



$f(4,3)$



```
fun f(x,y) = if x>y  
  then x  
  else f(2*x, y);  
f(1,3);
```

Optimization: pop followed by push

=> reuse activation record in place

Conclusion: tail recursive function

calls are equivalent to iterative loops

Example: Recursion vs. Loops

- Append lists

```
fun append(nil, ys) = ys  
/ append(x::xs, ys) = x :: append(xs, ys);
```

- Using loop and modification

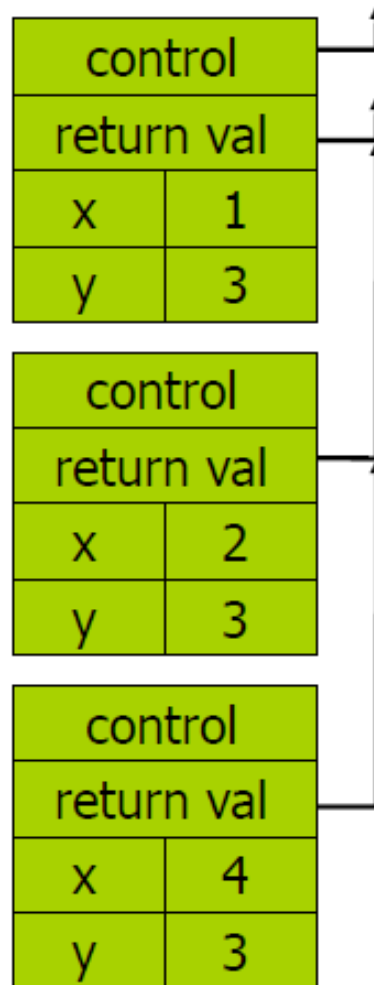
```
fun append(xs,ys) =  
  let val rxs = ref (reverse(xs)); val res = ref ys;  
  in while not (null(!rxs)) do  
    (res := hd(!rxs)::(!res); rxs := tl(!rxs) );  
    !res  
  end;
```

Example: Expressing Tail Recursion Using Loops

$f(1,3)$

control		↑
return val		↑
x	1	
y	3	

```
fun f(x,y) = if x>y
  then x
  else f(2*x, y);
f(1,3) + 7;
```



Expressed in loop:

```
fun f(x,y) =
  let val z = ref x in
  while not (!z > y) do
    z := 2 * !z;
  !z
  end;
```

$f(1,3) + 7;$

Tail recursion and iteration

□ Tail recursive function

```
fun last(x::nil) = x
  | last( x::y) = last(y);
```

□ Iteration

```
fun last(input) =
  let val y= ref input
  in while not(tl(!y)=nil)
    do
      y := tl(!y)
    end;
    hd(!y)
  end
```

□ Step1: what parameters change when making recursive calls?

- create a reference for each changed parameter.
- NOTE: no need to create reference for the return result
 - Tail recursion only returns at the base case

□ Step2: what is the base case of recursion?

- This is the stop condition for the while loop.

□ Step3: what to do before making tail call?

- loop body: prepare for the next tail call

□ Step4: return base case value.

Summary

- Block-structured languages use runtime stack to maintain activation records of blocks
 - Activation records contain parameters, local variables, ...
 - Also pointers to enclosing scope
- Several different parameter passing mechanisms
- Tail calls may be optimized
- Function parameters/results require closures
 - Env pointer of closure used when function is called
 - Runtime stack management may fail if functions are returned as result
 - Closures is not needed if functions are not in nested blocks
 - ***Example: C***



University of Colorado
Colorado Springs



University of Colorado

Boulder | Colorado Springs | Denver | Anschutz Medical Campus