NAMES, BINDING & SCOPES

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

Names:

- Function names, variable names, type names refer to memory addresses at runtime or to abstract type structures at compile time.

Names:

- Function names, variable names, type names refer to memory addresses at runtime or to abstract type structures at compile time.

Binding:

- To clearly define the semantics of the program, we need to clearly identify this association between names and the objects they refer to.
- The compiler/runtime system has to do this automatically.

Names:

- Function names, variable names, type names refer to memory addresses at runtime or to abstract type structures at compile time.

Binding:

- To clearly define the semantics of the program, we need to clearly identify this association between names and the objects they refer to.
- The compiler/runtime system has to do this automatically.

Scopes:

- What are the rules that determine which names are visible in which parts of the program?

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

ROAD MAP

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Name

A mnemonic character string representing something else (an identifier from the parser's point of view)

- `x`, `sin`, `f`, `prog1`, `null?` are names.
- `1`, `2`, `3`, `"test"` are not names.
- `+`, `<=`, … may be names if they are not built-in operations.

Name

A mnemonic character string representing something else (an identifier from the parser's point of view)

- `x`, `sin`, `f`, `prog1`, `null?` are names.
- `1`, `2`, `3`, `"test"` are not names.
- `+`, `<=`, ... may be names if they are not built-in operations.

Binding

An association between two entities, typically between a name and the object it refers to

- Name and memory location (for a variable)
- Name and function
- Name and type

Referencing environment

A complete set of bindings active at a certain point in a program

Referencing environment

A complete set of bindings active at a certain point in a program

Scope of a binding

The region of a program or time interval(s) in the program's execution during which the binding is active

Referencing environment

A complete set of bindings active at a certain point in a program

Scope of a binding

The region of a program or time interval(s) in the program's execution during which the binding is active

Scope

A maximal region of the program where no bindings are destroyed (e.g., a function body)

QUESTIONS ABOUT BINDINGS

- When is the binding established?
- How long does the binding/the bound object exist?
- Where does the bound object live?

Compile time:

- Map high-level language constructs to machine code
- Layout static data in memory

BINDING TIMES

Compile time:

- Map high-level language constructs to machine code
- Layout static data in memory

Link time:

- Resolve references between objects in separately compiled module

BINDING TIMES

Compile time:

- Map high-level language constructs to machine code
- Layout static data in memory

Link time:

- Resolve references between objects in separately compiled module

Load time:

- Assign machine addresses to static data

BINDING TIMES

Compile time:

- Map high-level language constructs to machine code
- Layout static data in memory

Link time:

- Resolve references between objects in separately compiled module

Load time:

- Assign machine addresses to static data

Runtime:

- Bind values to variables
- Allocate dynamic data and assign it to variables
- Allocate local variables on the stack

IMPORTANCE OF BINDING TIMES

Early binding (compile time, link time, load time):

- Faster code
- Typical in compiled languages

Late binding (runtime):

- Greater flexibility
- Typical in interpreted languages

Object lifetime

Period between the creation and destruction of the object

- Time between creation and destruction of a dynamically allocated variable in C++ using `new` and `delete`.

OBJECT AND BINDING LIFETIME

Object lifetime

Period between the creation and destruction of the object

- Time between creation and destruction of a dynamically allocated variable in C++ using `new` and `delete`.

Binding lifetime

Period between the creation and destruction of a binding (name-to-object association)

- Time between invocation and return of a function

Object lifetime

Period between the creation and destruction of the object

- Time between creation and destruction of a dynamically allocated variable in C++ using `new` and `delete`.

Binding lifetime

Period between the creation and destruction of a binding (name-to-object association)

- Time between invocation and return of a function

Two common mistakes:

- Dangling reference: no object for a binding
  (E.g., a pointer refers to an object that has already been deleted)

OBJECT AND BINDING LIFETIME

Object lifetime

Period between the creation and destruction of the object

- Time between creation and destruction of a dynamically allocated variable in C++ using `new` and `delete`.

Binding lifetime

Period between the creation and destruction of a binding (name-to-object association)

- Time between invocation and return of a function

Two common mistakes:

- Dangling reference: no object for a binding
  (E.g., a pointer refers to an object that has already been deleted)
- Memory leak: No binding for an object
  (Prevents the object from being deallocated)

STORAGE ALLOCATION

An object's lifetime is tied to the mechanism used to manage the space where the object resides.

Static object

- Stored at a fixed absolute address
- Lifetime spans the whole execution of the program

Object on stack

- Allocated on the stack in connection with a subroutine call (bound to local variable)
- Lifetime spans period between invocation and return of the subroutine

Object on heap

- Allocated on the heap
- Object created/destroyed at arbitrary times
  - Explicitly by programmer
  - Implicitly by garbage collector

EXAMPLE: OBJECT CREATION & DESTRUCTION IN C++

Object type	Lifetime	Allocation
Local	While function or block is active	Stack
Heap	Arbitrary	Heap
Non-static member	Same as parent object	Same as parent object
Array element	Same as array	Same as array
Local static	From first function/block execution until program exits	Static address space
Global, namespace, class static	While program runs	Static address space
Temporary	During expression evaluation	Register, stack

Note: User-supplied allocation function may change the lifetime of heap objects.

STATIC OBJECTS

- Global variables
- Static variables local to subroutines that retain their value between invocations
- Constant literals
- Tables for runtime support: debugging, type checking, etc.
- Space for subroutines, including local variables, in languages that do not support recursion
  (E.g., early versions of FORTRAN)

ROAD MAP

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

STACK-BASED ALLOCATION (1)

The stack is used to allocate space for subroutines in languages that permit recursion.

Stack frame (activation record)

- Arguments and local variables of the subroutine
- Return value(s) of the subroutine
- Return address

Calling sequence

Maintains the stack:

- Before call, the caller pushes arguments and return address onto the stack.
- After being called (prologue), the callee initializes local variables, etc.
- Before returning (epilogue), the callee cleans up local data.
- After the call returns, the caller retrieves return value(s) and restores the stack to its state before the call.

STACK FRAME (ACTIVATION RECORD)

The runtime management of the stack involves two registers:

- **Frame pointer:** Points to a known location within the current stack frame
- **Stack pointer:** Points to the first unused location on the stack (as a starting position for the next stack frame)

STACK FRAME (ACTIVATION RECORD)

The runtime management of the stack involves two registers:

- **Frame pointer:** Points to a known location within the current stack frame
- **Stack pointer:** Points to the first unused location on the stack (as a starting position for the next stack frame)

The compiler determines the locations of function arguments and local variables relative to the frame pointer.

STACK FRAME (ACTIVATION RECORD)

The runtime management of the stack involves two registers:

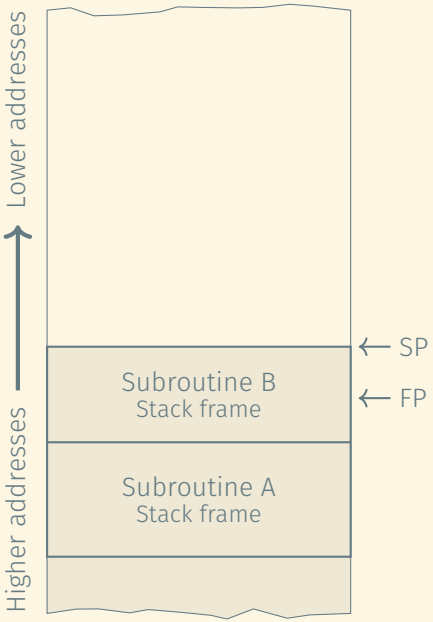
- **Frame pointer:** Points to a known location within the current stack frame
- **Stack pointer:** Points to the first unused location on the stack (as a starting position for the next stack frame)

The compiler determines the locations of function arguments and local variables relative to the frame pointer.

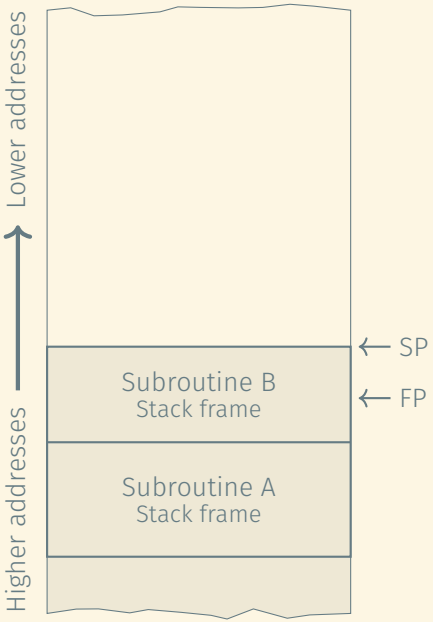
The runtime system manages

- Absolute location of the stack frame in memory (on the stack)
- The size of the stack frame (E.g., when allocating variable-size arrays on the stack)

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

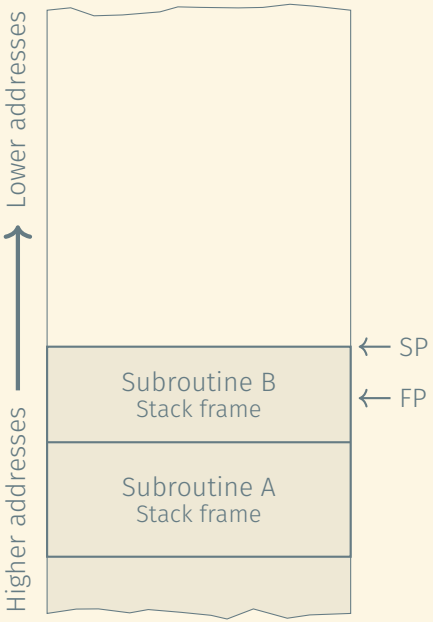


STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



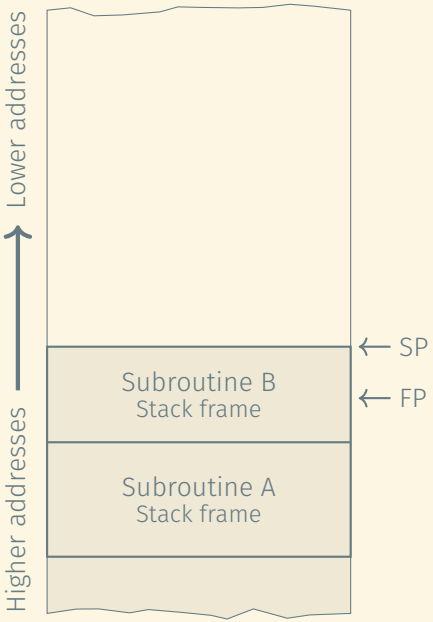
Making the call:

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

Caller:

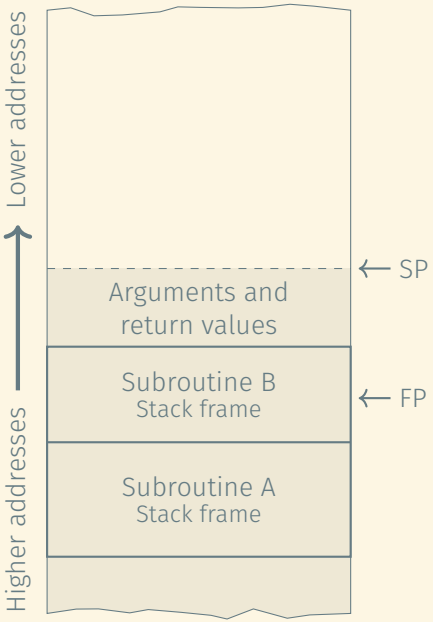


Making the call:

Caller:

- Push arguments on the stack

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

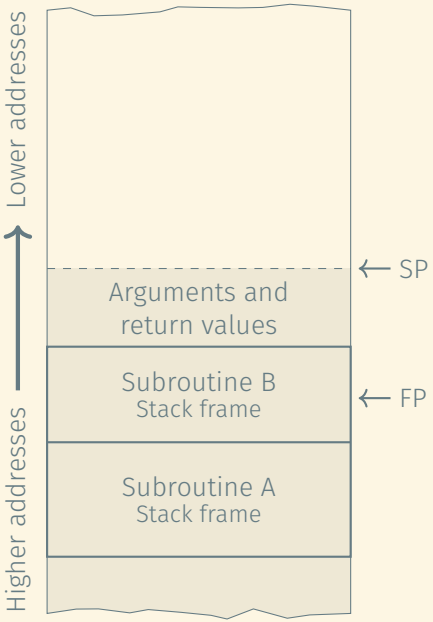


Making the call:

Caller:

- Push arguments on the stack

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

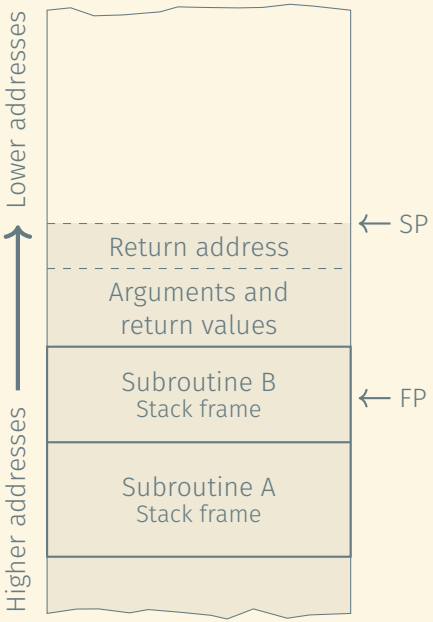


Making the call:

Caller:

- Push arguments on the stack
- Make subroutine call

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

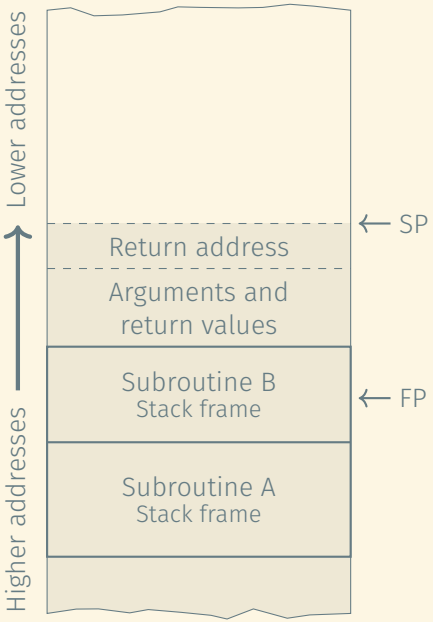


Making the call:

Caller:

- Push arguments on the stack
- Make subroutine call

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



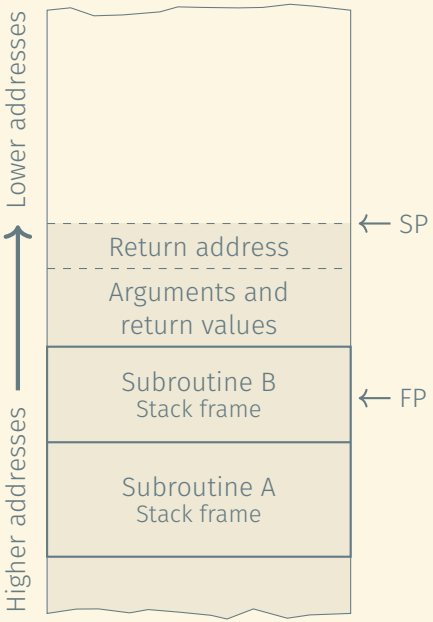
Making the call:

Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

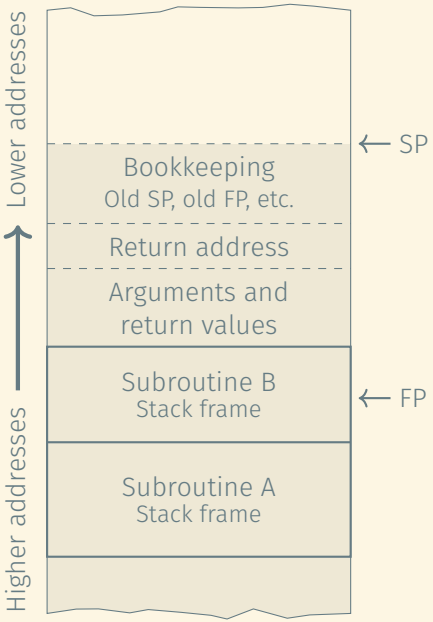
Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

- Push stack pointer and frame pointer on stack

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

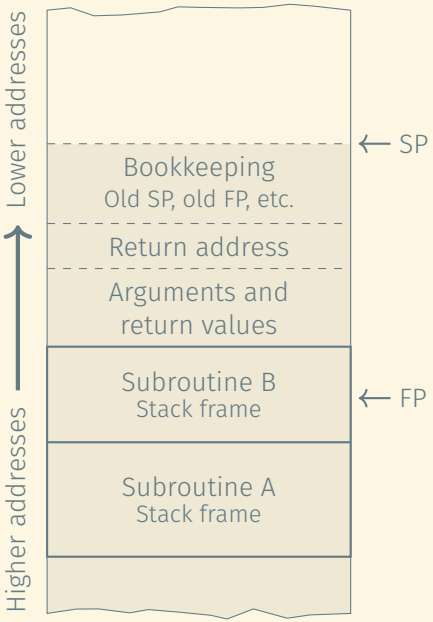
Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

- Push stack pointer and frame pointer on stack

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

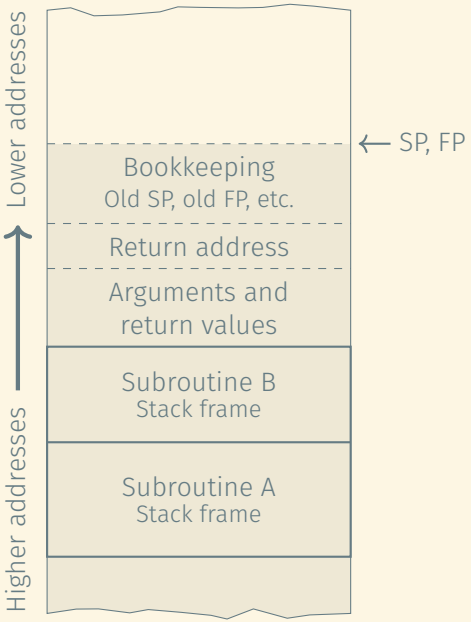
Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

- Push stack pointer and frame pointer on stack
- Update frame pointer

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

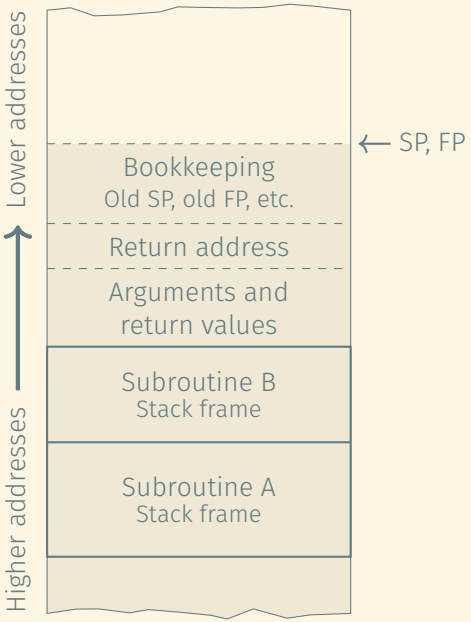
Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

- Push stack pointer and frame pointer on stack
- Update frame pointer

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

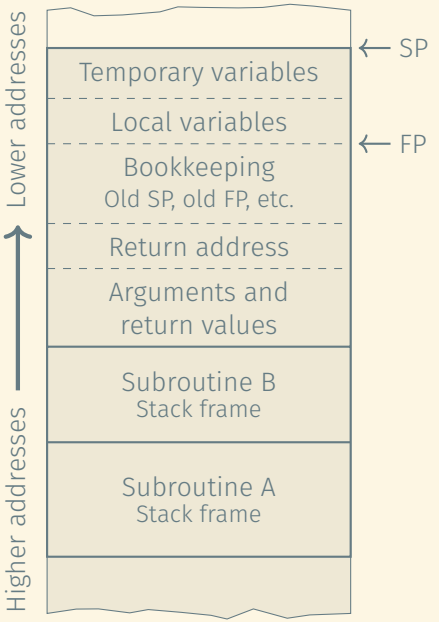
Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

- Push stack pointer and frame pointer on stack
- Update frame pointer
- Allocate local variables and temporary working space, update stack pointer

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Making the call:

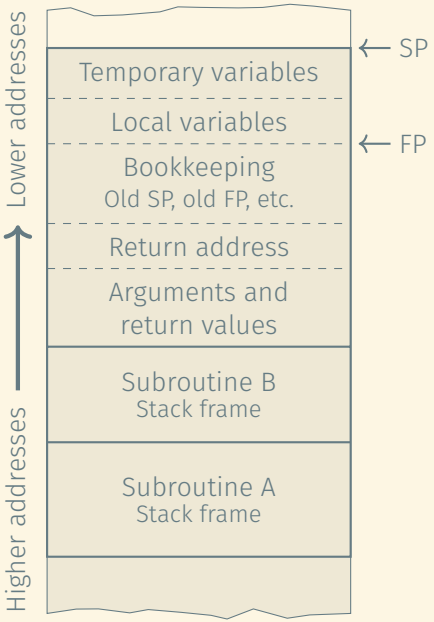
Caller:

- Push arguments on the stack
- Make subroutine call

Callee:

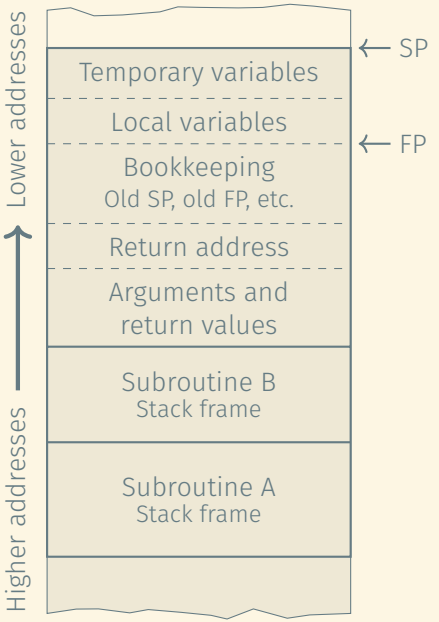
- Push stack pointer and frame pointer on stack
- Update frame pointer
- Allocate local variables and temporary working space, update stack pointer

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Returning from the call:

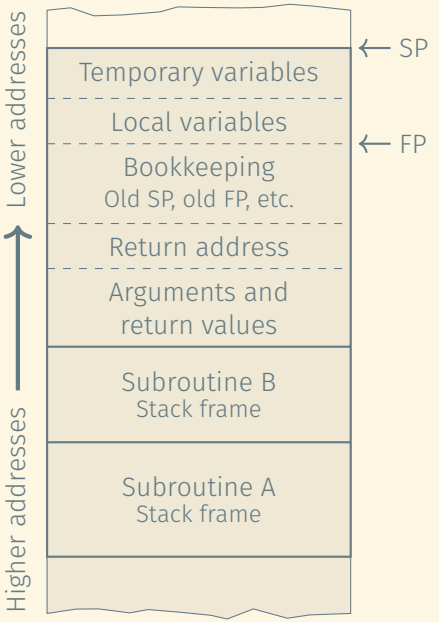
STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Returning from the call:

Callee:

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

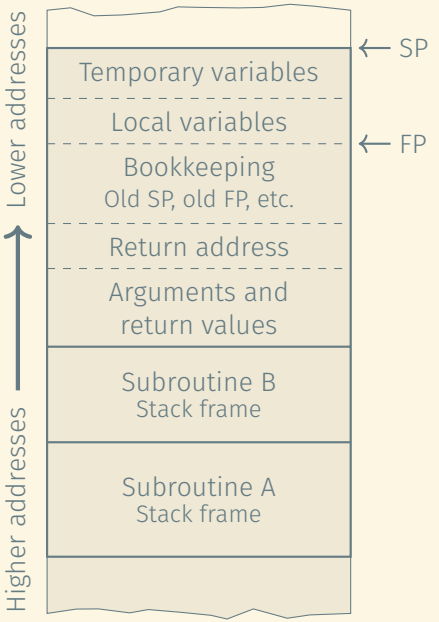


Returning from the call:

Callee:

- Clean up local data as necessary

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

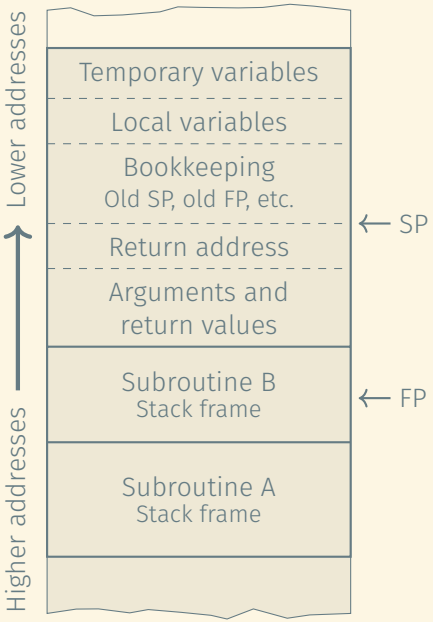


Returning from the call:

Callee:

- Clean up local data as necessary
- Restore stack pointer and frame pointer

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

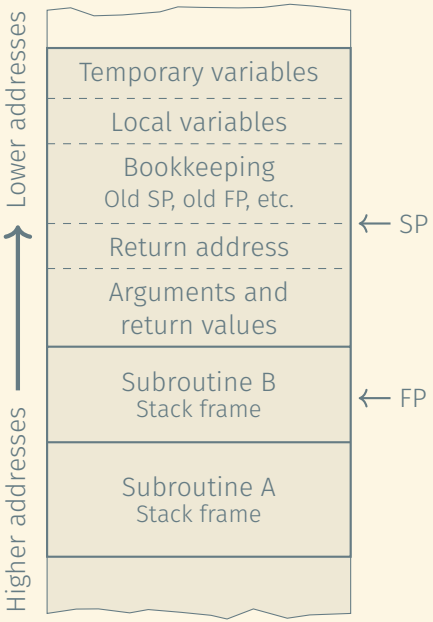


Returning from the call:

Callee:

- Clean up local data as necessary
- Restore stack pointer and frame pointer

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

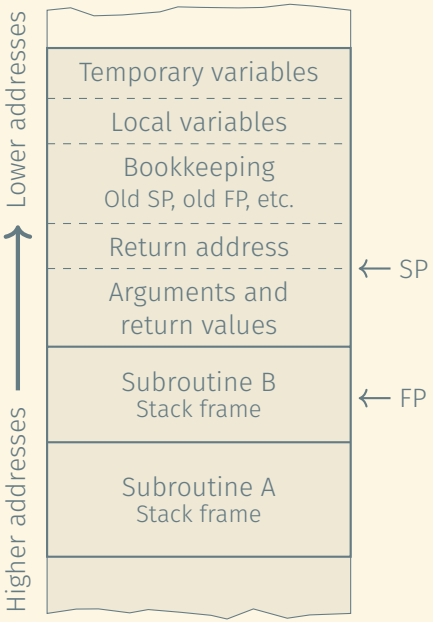


Returning from the call:

Callee:

- Clean up local data as necessary
- Restore stack pointer and frame pointer
- Return

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL

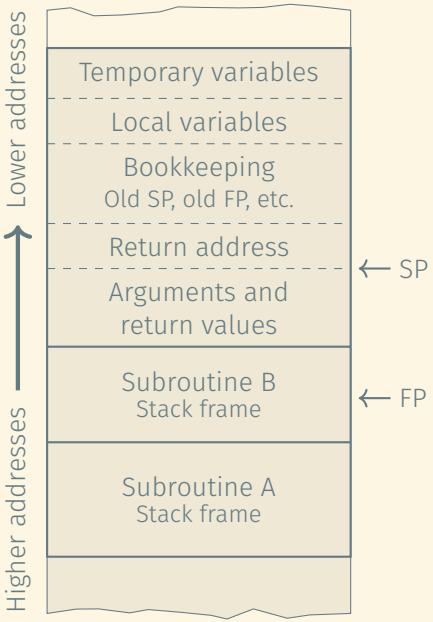


Returning from the call:

Callee:

- Clean up local data as necessary
- Restore stack pointer and frame pointer
- Return

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Returning from the call:

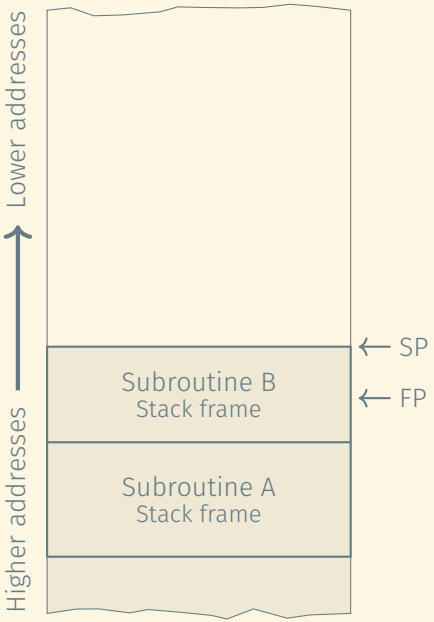
Callee:

- Clean up local data as necessary
- Restore stack pointer and frame pointer
- Return

Caller:

- Retrieve return value from the stack and restore stack pointer to top of previous stack frame

STACK FRAME BEFORE, DURING & AFTER SUBROUTINE CALL



Returning from the call:

Callee:

- Clean up local data as necessary
- Restore stack pointer and frame pointer
- Return

Caller:

- Retrieve return value from the stack and restore stack pointer to top of previous stack frame

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Heap

A region of memory where blocks can be allocated at arbitrary times and in arbitrary order.

Heap

A region of memory where blocks can be allocated at arbitrary times and in arbitrary order.

Heap

A region of memory where blocks can be allocated at arbitrary times and in arbitrary order.

Heap management

Heap

A region of memory where blocks can be allocated at arbitrary times and in arbitrary order.

Heap management

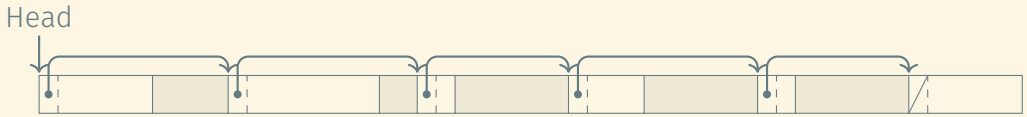
Free list: List of blocks of free memory

Heap

A region of memory where blocks can be allocated at arbitrary times and in arbitrary order.

Heap management

Free list: List of blocks of free memory



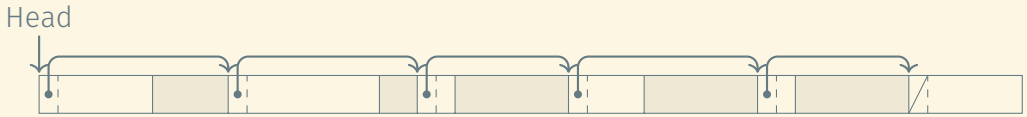
Heap

A region of memory where blocks can be allocated at arbitrary times and in arbitrary order.

Heap management

Free list: List of blocks of free memory

The allocation algorithm searches for a block of adequate size to accommodate the allocation request.



General allocation strategy

- Find a free block that is at least as big as the requested amount of memory.
- Mark requested number of bytes (plus padding) as allocated.
- Return rest of the free block to free list.

General allocation strategy

- Find a free block that is at least as big as the requested amount of memory.
- Mark requested number of bytes (plus padding) as allocated.
- Return rest of the free block to free list.

First fit: Find the first block large enough to accommodate the allocation request.

FIRST-FIT AND BEST-FIT ALLOCATION

General allocation strategy

- Find a free block that is at least as big as the requested amount of memory.
- Mark requested number of bytes (plus padding) as allocated.
- Return rest of the free block to free list.

First fit: Find the first block large enough to accommodate the allocation request.

Best fit: Find the smallest block large enough to accommodate the request.

THE HEAP FRAGMENTATION PROBLEM

Internal fragmentation

- Often only blocks of certain sizes (e.g., 2^k) are allocated.
- This may lead to part of an allocated block being unused.

Internal fragmentation

- Often only blocks of certain sizes (e.g., 2^k) are allocated.
- This may lead to part of an allocated block being unused.

External fragmentation

- Unused space may consist of many small blocks.
- Thus, although the total free space may exceed the allocation request, no block may be large enough to accommodate it.

Internal fragmentation

- Often only blocks of certain sizes (e.g., 2^k) are allocated.
- This may lead to part of an allocated block being unused.

External fragmentation

- Unused space may consist of many small blocks.
- Thus, although the total free space may exceed the allocation request, no block may be large enough to accommodate it.

Neither best-fit nor first-fit is guaranteed to minimize external fragmentation.

The best strategy depends on the size distribution of the allocation requests.

COST OF HEAP ALLOCATION

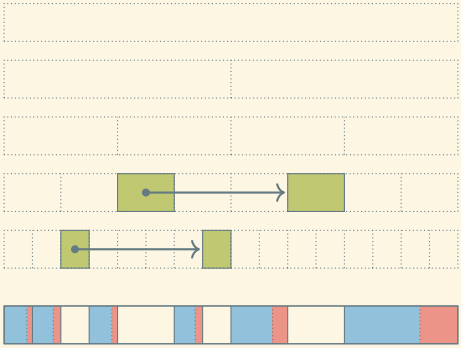
Single free list: Linear cost to find a block to accommodate each request.

COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

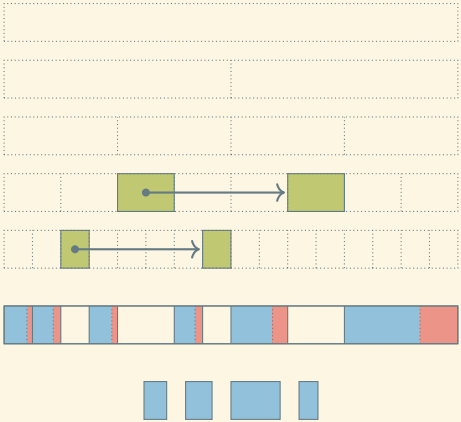


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

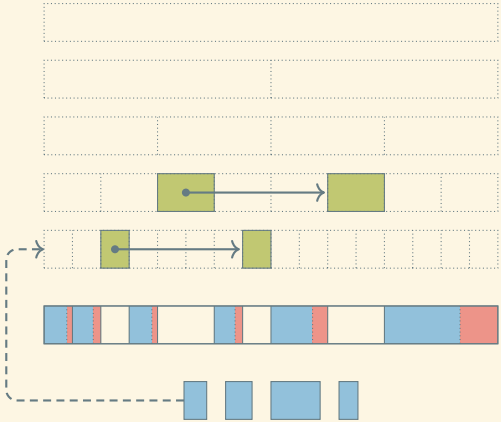
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

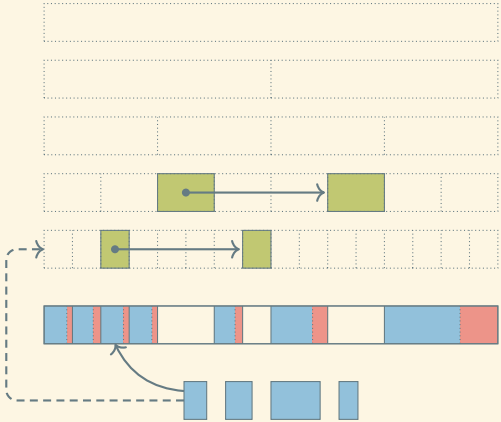


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

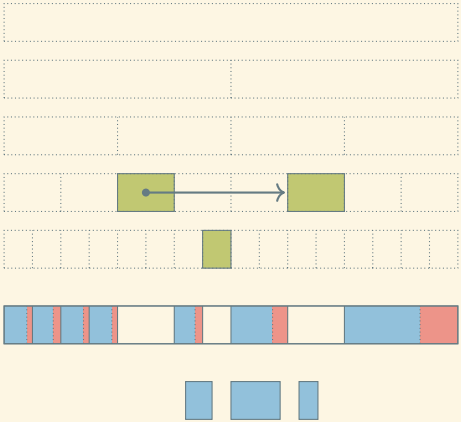
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

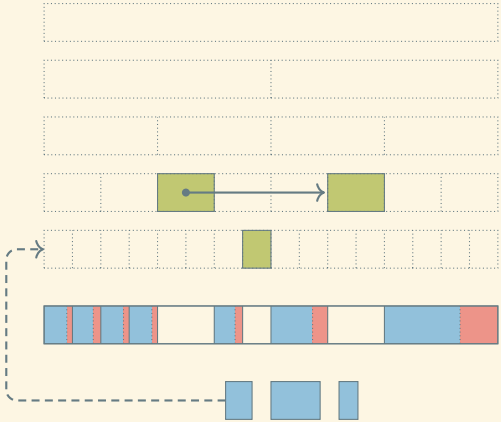


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

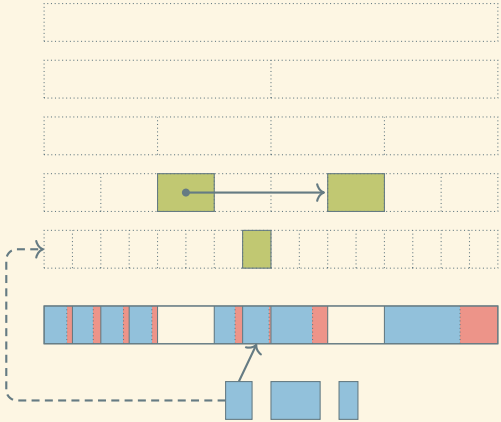
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

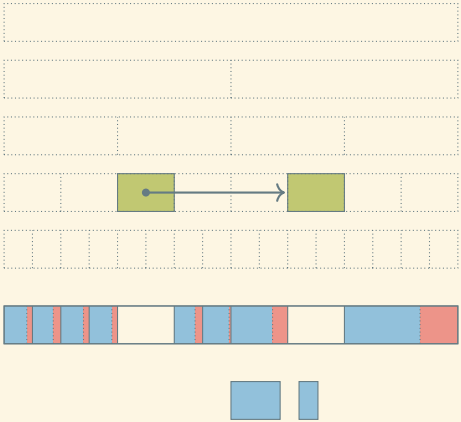


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

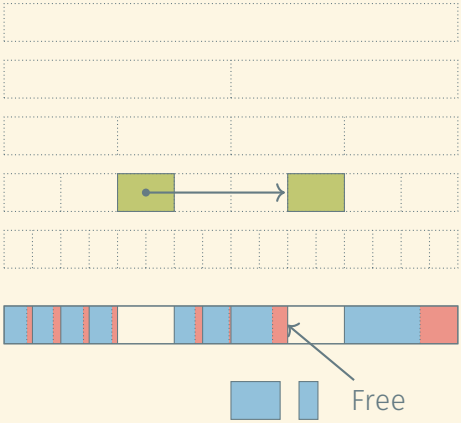
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

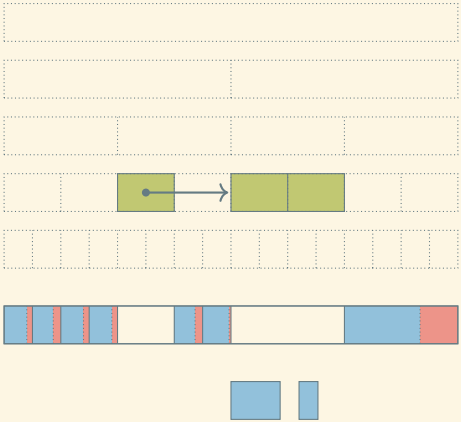


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

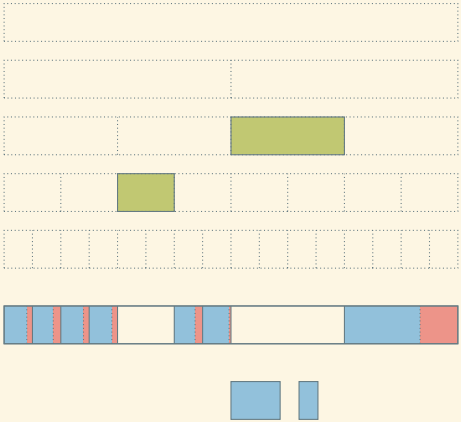


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

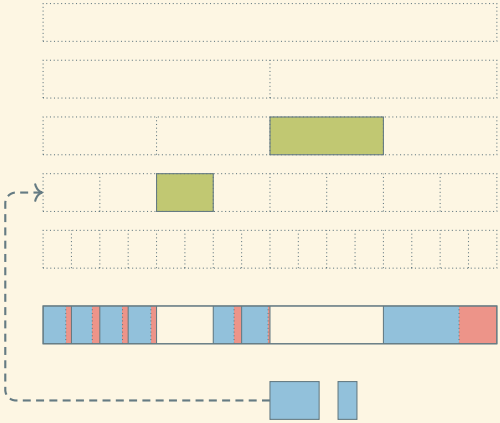


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

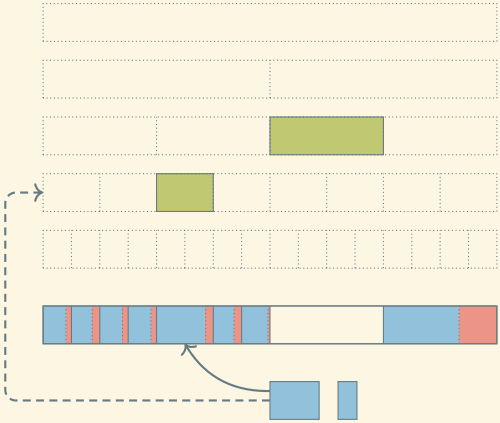


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

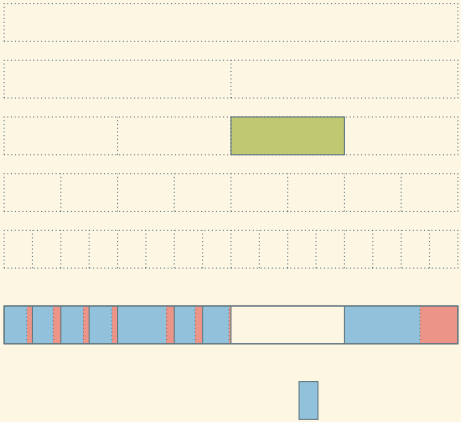
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

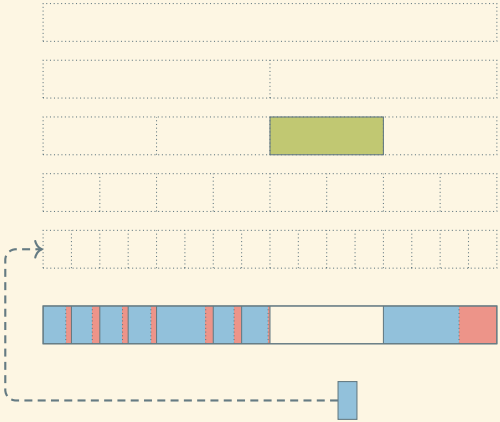


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

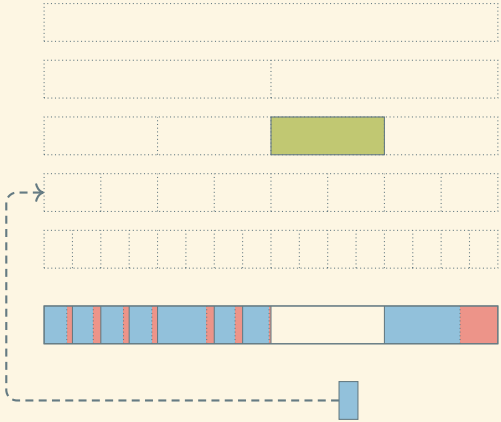
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

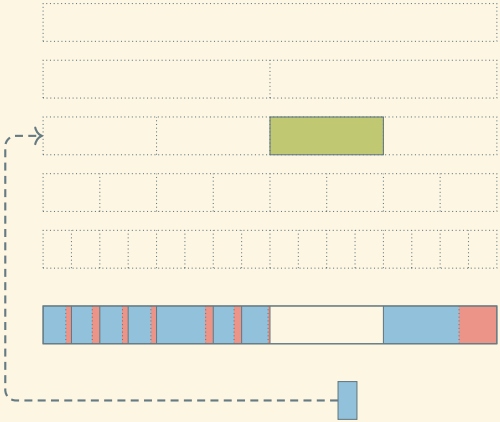
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

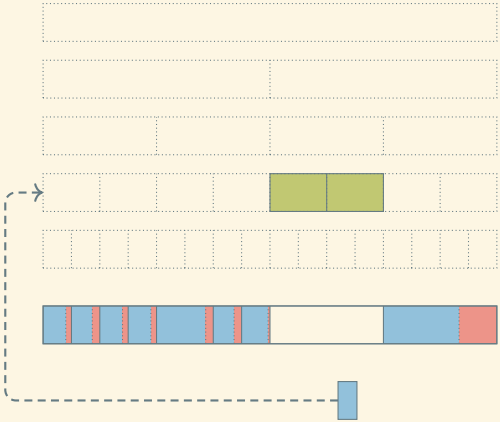
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

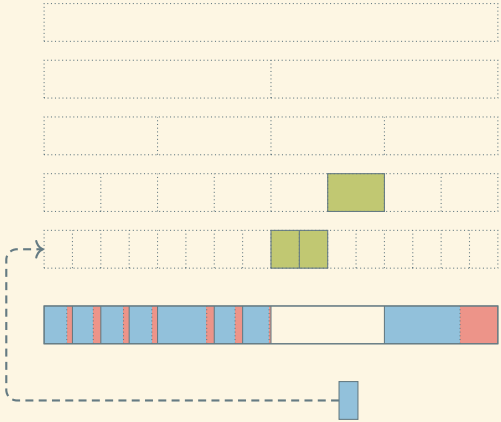
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

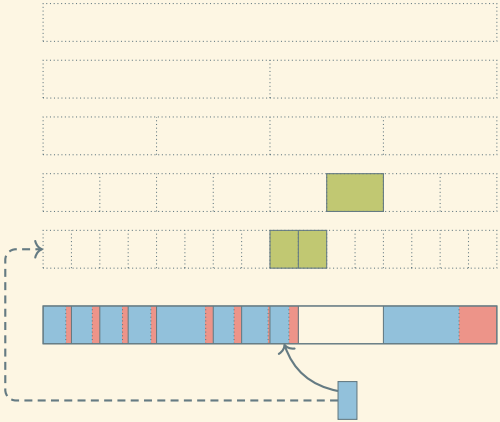
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)

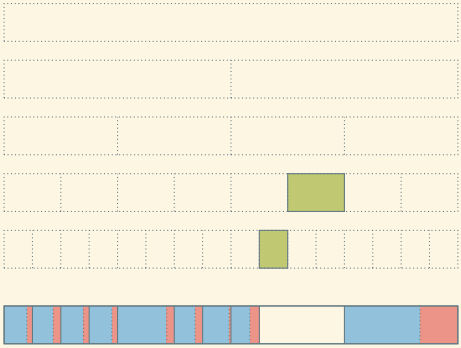


COST OF HEAP ALLOCATION

Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

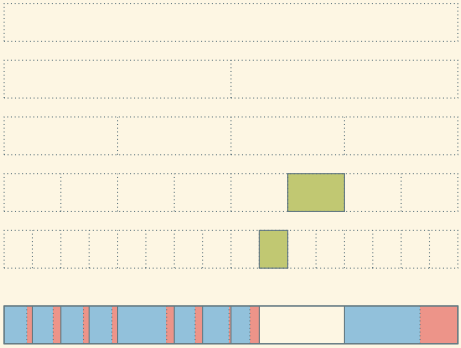
- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Single free list: Linear cost to find a block to accommodate each request.

Buddy system:

- Blocks of size $2^{n_0}, 2^{n_0+1}, 2^{n_0+2}, \ldots$
- Separate free list for each block size
- If block of size 2^k is unavailable, split block of size 2^{k+1}
- If block of size 2^k is deallocated and its buddy is free, merge them
- Worst-case cost: log(memory size)



Fibonacci heap:

- Block sizes are Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Less fragmentation

HEAP DEALLOCATION

Explicit deallocation by programmer:

- Used in Pascal, C, C++, ...
- Efficient
- May lead to bugs that are difficult to find:
  - Dangling pointers/references from deallocating too soon.
  - Memory leaks from not deallocating.

Explicit deallocation by programmer:

- Used in Pascal, C, C++, ...
- Efficient
- May lead to bugs that are difficult to find:
  - Dangling pointers/references from deallocating too soon.
  - Memory leaks from not deallocating.

Automatic deallocation by garbage collector:

- Using in Java, Python, declarative programming languages, ...
- Can add significant runtime overhead
- Safer

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Scope of a binding

The region of a program or time interval(s) in the program's execution during which the binding is active

Scope

Maximal region of the program where no bindings are destroyed (e.g., a function body)

Scope of a binding

The region of a program or time interval(s) in the program's execution during which the binding is active

Scope

Maximal region of the program where no bindings are destroyed (e.g., a function body)

Lexical (static) scoping

- Binding based on nesting of blocks
- Can be determined at compile time

Scope of a binding

The region of a program or time interval(s) in the program's execution during which the binding is active

Scope

Maximal region of the program where no bindings are destroyed (e.g., a function body)

Lexical (static) scoping

- Binding based on nesting of blocks
- Can be determined at compile time

Dynamic scoping

- Binding depends on flow of execution at runtime
- Can only be determined at runtime

Lexical scoping

The binding for a name at the current position is the one encountered in the smallest enclosing lexical unit.

Lexical scoping

The binding for a name at the current position is the one encountered in the smallest enclosing lexical unit.

Lexical units

- Packages, modules, source files
- Classes
- Procedures/functions and nested subroutines
- Blocks
- Records and structures

Lexical scoping

The binding for a name at the current position is the one encountered in the smallest enclosing lexical unit.

Lexical units

- Packages, modules, source files
- Classes
- Procedures/functions and nested subroutines
- Blocks
- Records and structures

Important variant: The current binding for a name is the one encountered in the smallest enclosing lexical unit and preceding the current point in the progam text.

The scope of the red variable is shaded in each example.

```
(... (define x ...) fun1 fun2 ... funk)
```

```
(lambda (x ...) fun1 fun2 ... funk)
```

```
(let ((x exp1) (y exp2) (z exp3)) fun1 fun2 ... funk)
```

```
(let* ((x exp1) (y exp2) (z exp3)) fun1 fun2 ... funk)
```

```
(letrec ((x exp1) (y exp2) (z exp3)) fun1 fun2 ... funk)
```

LEXICAL SCOPING IN SCHEME: EXAMPLE

```
(define (fun a b)
  (let ((x 0)
        (y 0)
        (z 0))

    (set! x (+ a b))

    (let ((a 3)
          (b 4))

      (let ((a 5)
            (b 6))
        (set! y (+ a b)))

      (set! z (+ a b)))

    (list x y z)))
```

LEXICAL SCOPING IN SCHEME: EXAMPLE

```
(define (fun a b)
  (let ((x 0)
        (y 0)
        (z 0))

    (set! x (+ a b))

    (let ((a 3)
          (b 4))

      (let ((a 5)
            (b 6))
        (set! y (+ a b)))

      (set! z (+ a b)))

    (list x y z)))
```

What does `(fun 1 2)` return?

LEXICAL SCOPING IN SCHEME: EXAMPLE

```
(define (fun a b)
  (let ((x 0)
        (y 0)
        (z 0))

    (set! x (+ a b))

    (let ((a 3)
          (b 4))

      (let ((a 5)
            (b 6))
        (set! y (+ a b)))

      (set! z (+ a b)))

    (list x y z)))
```

What does `(fun 1 2)` return?

`'(3 11 7)`

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

What's visible inside P1's body?

What's visible inside P2's body?

What's visible inside P3's body?

What's visible inside P4's body?

What's visible inside P5's body?

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

What's visible inside P1's body?

P1, A1, X_1, P2, P4

What's visible inside P2's body?

What's visible inside P3's body?

What's visible inside P4's body?

What's visible inside P5's body?

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

What's visible inside P1's body?

P1, A1, X_1, P2, P4

What's visible inside P2's body?

P1, A1, X_1, P2, P3, A2

What's visible inside P3's body?

What's visible inside P4's body?

What's visible inside P5's body?

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

What's visible inside P1's body?

P1, A1, X_1, P2, P4

What's visible inside P2's body?

P1, A1, X_1, P2, P3, A2

What's visible inside P3's body?

P1, A1, X_1, P2, P3, A2, A3

What's visible inside P4's body?

What's visible inside P5's body?

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

What's visible inside P1's body?

P1, A1, X_1, P2, P4

What's visible inside P2's body?

P1, A1, X_1, P2, P3, A2

What's visible inside P3's body?

P1, A1, X_1, P2, P3, A2, A3

What's visible inside P4's body?

P1, A1, X_1, P2, P4, A4, F1

What's visible inside P5's body?

LEXICAL SCOPING IN PASCAL

```
procedure P1( A1 : T1 );
  var X : real;
  procedure P2( A2 : T2 );
    procedure P3( A3 : T3 );
    begin
      ...
    end;
  begin
    ...
  end;
  procedure P4( A4 : T4 );
    function F1( A5 : T5 ) : T6;
    var X : integer;
    begin
      ...
    end;
  begin
    ...
  end;
begin
  ...
end;
```

What's visible inside P1's body?

P1, A1, X_1, P2, P4

What's visible inside P2's body?

P1, A1, X_1, P2, P3, A2

What's visible inside P3's body?

P1, A1, X_1, P2, P3, A2, A3

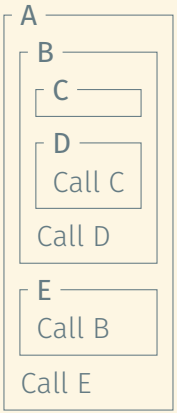
What's visible inside P4's body?

P1, A1, X_1, P2, P4, A4, F1

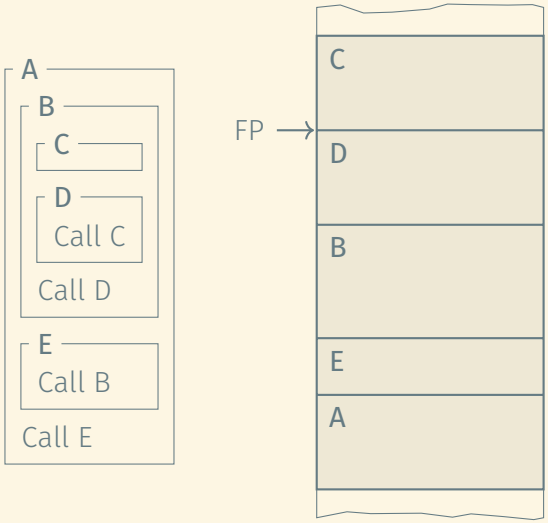
What's visible inside P5's body?

P1, A1, P2, P4, A4, F1, A5, X_2

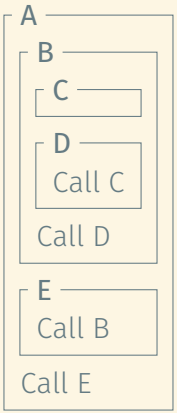
IMPLEMENTATION OF LEXICAL SCOPING: STATIC CHAINS

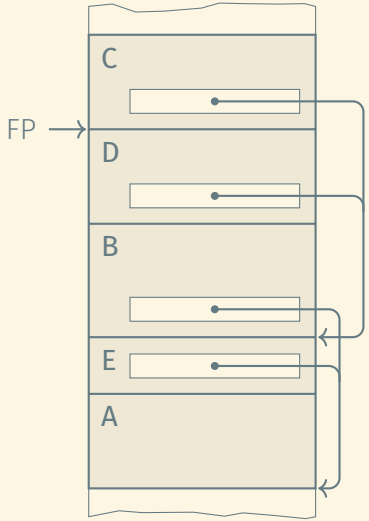


IMPLEMENTATION OF LEXICAL SCOPING: STATIC CHAINS



The stack frame of each invocation has a static link to the stack frame of the most recent invocation of the lexically surrounding subroutine.

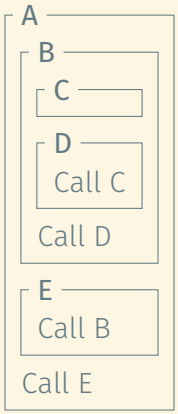


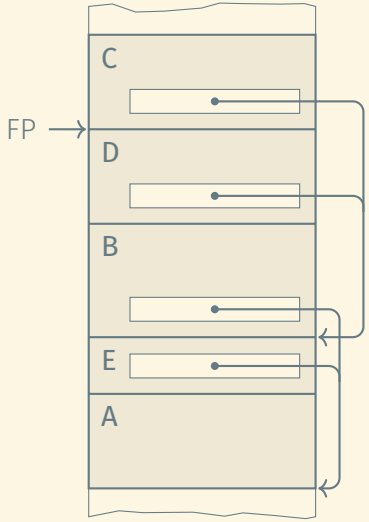


IMPLEMENTATION OF LEXICAL SCOPING: STATIC CHAINS

The stack frame of each invocation has a static link to the stack frame of the most recent invocation of the lexically surrounding subroutine.

To reference a variable in some outer scope, the chain of static links is traversed, followed by adding the offset of the variable relative to the stack frame it is in.



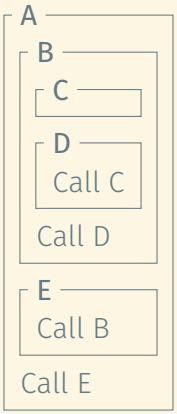


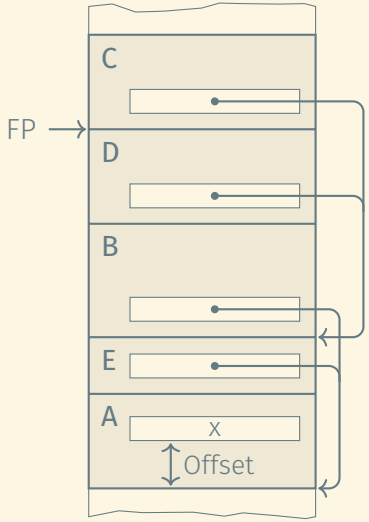
IMPLEMENTATION OF LEXICAL SCOPING: STATIC CHAINS

The stack frame of each invocation has a static link to the stack frame of the most recent invocation of the lexically surrounding subroutine.

To reference a variable in some outer scope, the chain of static links is traversed, followed by adding the offset of the variable relative to the stack frame it is in.

Example: Access variable `x` in procedure `A` from within procedure `C`.





Dynamic binding

The current binding for a given name is the one

- Encountered most recently during execution and
- Not yet destroyed by exiting its scope.

```
# Static scoping
sub f {
  my $a = 1;
  print "f:$a\n";
  &printa();
}

sub printa { print "p:$a\n"; }

$a = 2;
&f();
```

```
# Dynamic scoping
sub f {
  local $a = 1;
  print "f:$a\n";
  &printa();
}

sub printa { print "p:$a\n"; }

$a = 2;
&f();
```

LEXICAL VS DYNAMIC SCOPING: ANOTHER EXAMPLE

```
a : integer -- global declaration

procedure first
  a := 1
end

procedure second
  a : integer -- local declaration
  first()
end

a := 2

second()
print(a)
```

LEXICAL VS DYNAMIC SCOPING: ANOTHER EXAMPLE

```
a : integer -- global declaration

procedure first
  a := 1
end

procedure second
  a : integer -- local declaration
  first()
end

a := 2

second()
print(a)
```

What does this program print

- Under lexical scoping?

LEXICAL VS DYNAMIC SCOPING: ANOTHER EXAMPLE

```
a : integer -- global declaration

procedure first
  a := 1
end

procedure second
  a : integer -- local declaration
  first()
end

a := 2

second()
print(a)
```

What does this program print

- Under lexical scoping?
- Under dynamic scoping?

LEXICAL VS DYNAMIC SCOPING: ANOTHER EXAMPLE

```
a : integer -- global declaration

procedure first
  a := 1
end

procedure second
  a : integer -- local declaration
  first()
end

a := 2

second()
print(a)
```

What does this program print

- Under lexical scoping?
- Under dynamic scoping?

Dynamic scoping is usually a bad idea!

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

ROAD MAP

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

If a subroutine is passed as a parameter, when are the free variables bound?

Shallow binding

Free variables are bound when the subroutine is called.

Deep binding

Free variables are bound when the subroutine is first passed as a parameter.

This is important using both static and dynamic scoping and is known as the funarg problem.

SHALLOW AND DEEP BINDING: EXAMPLE

```
int x = 10;

function f( int a ) {
  x = x + a;
}

function g( function h ) {
  int x = 30;
  h( 100 );
  print( x );
}

function main() {
  g( f );
  print( x );
}
```

SHALLOW AND DEEP BINDING: EXAMPLE

```
int x = 10;

function f( int a ) {
  x = x + a;
}

function g( function h ) {
  int x = 30;
  h( 100 );
  print( x );
}

function main() {
  g( f );
  print( x );
}
```

What is the output of this program using deep binding?

What is the output of this program using shallow binding?

SHALLOW AND DEEP BINDING: EXAMPLE

```
int x = 10;

function f( int a ) {
  x = x + a;
}

function g( function h ) {
  int x = 30;
  h( 100 );
  print( x );
}

function main() {
  g( f );
  print( x );
}
```

What is the output of this program using deep binding?

```
30
110
```

What is the output of this program using shallow binding?

SHALLOW AND DEEP BINDING: EXAMPLE

```
int x = 10;

function f( int a ) {
  x = x + a;
}

function g( function h ) {
  int x = 30;
  h( 100 );
  print( x );
}

function main() {
  g( f );
  print( x );
}
```

What is the output of this program using deep binding?

```
30
110
```

What is the output of this program using shallow binding?

```
130
10
```

EXAMPLE OF THE FUNARG PROBLEM

```
type person = record
  age : integer;
end;

(* age threshold *)
threshold : integer;
people : database;

function older_than(p : person) : boolean
begin
  return p.age >= threshold
end;

procedure print_person(p : person)
begin
(* use line_length to format data *)
end;
```

```
procedure print_selected_records(
  db                        : database;
  predicate, print_routine : procedure
)
  var line_length : integer;
begin
  if device_type(stdout) = terminal
    then line_length := 80;
    else line_length := 132;
  for each record r in db
    if predicate(r)
      then print_routine(r);
end;

threshold := 35;
print_selected_records(
  people, older_than, print_person
);
```

EXAMPLE OF THE FUNARG PROBLEM

The function `older_than` assumes deep binding
(but not necessarily lexical scoping).

The procedure `print_person` assumes shallow binding (and dynamic scoping).

```
people : database;

function older_than(p : person) : boolean
begin
  return p.age >= threshold
end;

procedure print_person(p : person)
begin
(* use line_length to format data *)
end;
```

```
  if device_type(stdout) = terminal
    then line_length := 80;
    else line_length := 132;
  for each record r in db
    if predicate(r)
      then print_routine(r);
end;

threshold := 35;
print_selected_records(
  people, older_than, print_person
);
```

EXAMPLE OF THE FUNARG PROBLEM

This is an example of terrible programming style: the behaviour of both `older_than` and `print_person` depends on variable values not explicitly associated with them.

```
threshold : integer;
people : database;

function older_than(p : person) : boolean
begin
  return p.age >= threshold
end;

procedure print_person(p : person)
begin
(* use line_length to format data *)
end;
```

```
begin
  if device_type(stdout) = terminal
    then line_length := 80;
    else line_length := 132;
  for each record r in db
    if predicate(r)
      then print_routine(r);
end;

threshold := 35;
print_selected_records(
  people, older_than, print_person
);
```

EXAMPLE OF THE FUNARG PROBLEM

The function `older_than` should take the age threshold as an argument.

The procedure `print_person` and its associated settings should be wrapped in a class.

```
people : database;

function older_than(p : person) : boolean
begin
  return p.age >= threshold
end;

procedure print_person(p : person)
begin
(* use line_length to format data *)
end;
```

```
begin
  if device_type(stdout) = terminal
    then line_length := 80;
    else line_length := 132;
  for each record r in db
    if predicate(r)
      then print_routine(r);
end;

threshold := 35;
print_selected_records(
  people, older_than, print_person
);
```

Closure

A pair bundling

- A reference to a subroutine with
- A referencing environment active when the subroutine was defined.

Closures are necessary to implement deep binding in languages that allow functions to be passed around as function arguments and return values:

- When the function is invoked, the referencing environment it refers to may no longer exist and thus needs to be preserved explicitly in the closure.
- This can be used to implement “poor man’s objects”.

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
```

```
> (st1 'pop)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
```

```
> (st1 'pop)
3
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
> (st1 'pop)
3
> (st2 'pop)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
> (st1 'pop)
3
> (st2 'pop)
4
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
> (st1 'pop)
3
> (st2 'pop)
4
> (st1 'pop)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
> (st1 'pop)
3
> (st2 'pop)
4
> (st1 'pop)
2
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
> (st1 'pop)
3
> (st2 'pop)
4
> (st1 'pop)
2
> (st1 'pop)
```

SUBROUTINE CLOSURES: EXAMPLE

```
(define (new-stack)
  (let ((stack '()))
    (lambda (op . args)
      (cond ((eq? op 'push)
             (set! stack (append (reverse args) stack)))
            ((eq? op 'pop)
             (let ((top (car stack)))
               (set! stack (cdr stack))
               top))
            ((eq? op 'empty)
             (null? stack))))))
```

```
> (define st1 (new-stack))
> (define st2 (new-stack))
> (st1 'push 1 2 3)
> (st1 'empty)
#f
> (st2 'empty)
#t
> (st2 'push 4)
> (st1 'pop)
3
> (st2 'pop)
4
> (st1 'pop)
2
> (st1 'pop)
1
```

Frame

- A collection of variable-object bindings
- Can point to a “parent frame” to construct static chains

Frame

- A collection of variable-object bindings
- Can point to a “parent frame” to construct static chains

Referencing environment

A chain of frames represented by pointing to the first frame in the chain

Frame

- A collection of variable-object bindings
- Can point to a "parent frame" to construct static chains

Referencing environment

A chain of frames represented by pointing to the first frame in the chain

The value of a variable x is the value bound to x in the first frame that provides such a binding.

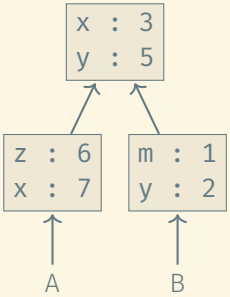
Frame

- A collection of variable-object bindings
- Can point to a "parent frame" to construct static chains

Referencing environment

A chain of frames represented by pointing to the first frame in the chain

The value of a variable `x` is the value bound to `x` in the first frame that provides such a binding.



What are the values of `x` and `y` in the environments `A` and `B`, respectively?

Frame

- A collection of variable-object bindings
- Can point to a "parent frame" to construct static chains

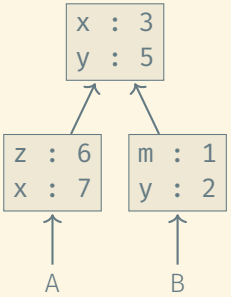
Referencing environment

A chain of frames represented by pointing to the first frame in the chain

Closure

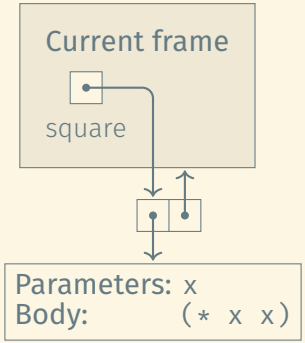
The code of a function paired with a pointer to a referencing environment

The value of a variable `x` is the value bound to `x` in the first frame that provides such a binding.



What are the values of `x` and `y` in the environments `A` and `B`, respectively?

```
(define (square x) (* x x))
```

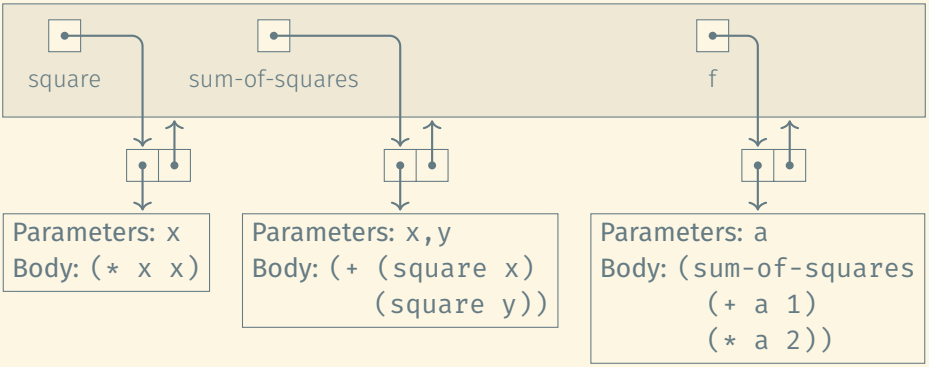


CLOSURES: EXTENDED EXAMPLE (1)

```
(define (square x)           (* x x))
(define (sum-of-squares x y) (+ (square x) (square y)))
(define (f a)                (sum-of-squares (+ a 1) (* a 2)))
```

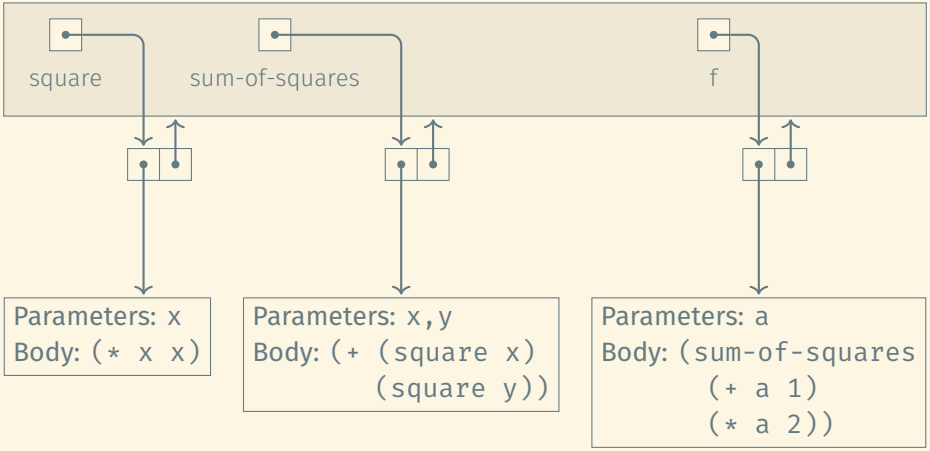
CLOSURES: EXTENDED EXAMPLE (1)

```
(define (square x)            (* x x))
(define (sum-of-squares x y) (+ (square x) (square y)))
(define (f a)                 (sum-of-squares (+ a 1) (* a 2)))
```



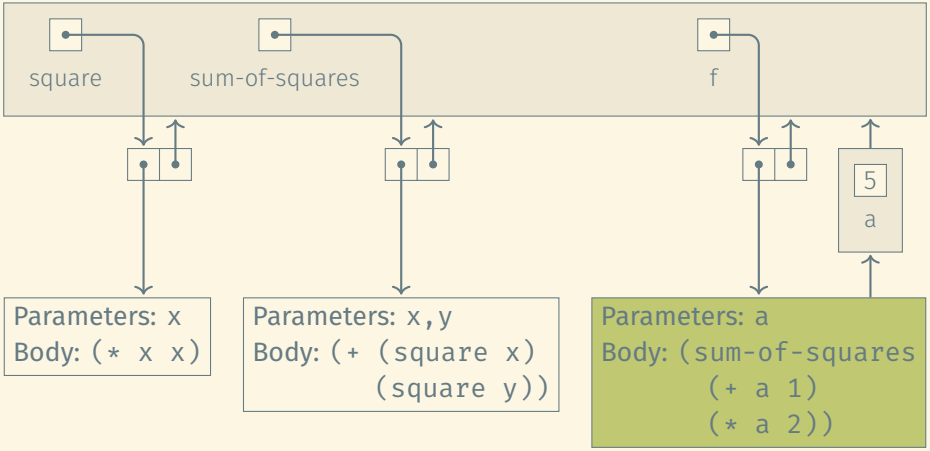
CLOSURES: EXTENDED EXAMPLE (2)

Invocation: `(f 5)`

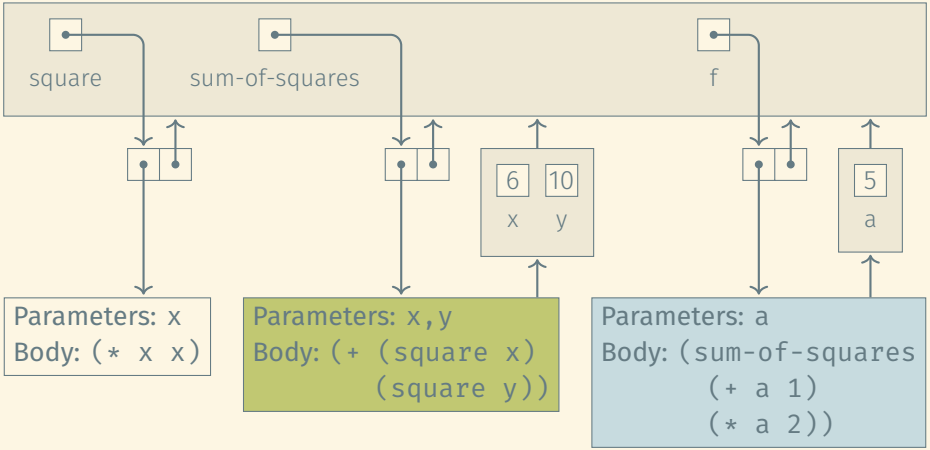


CLOSURES: EXTENDED EXAMPLE (2)

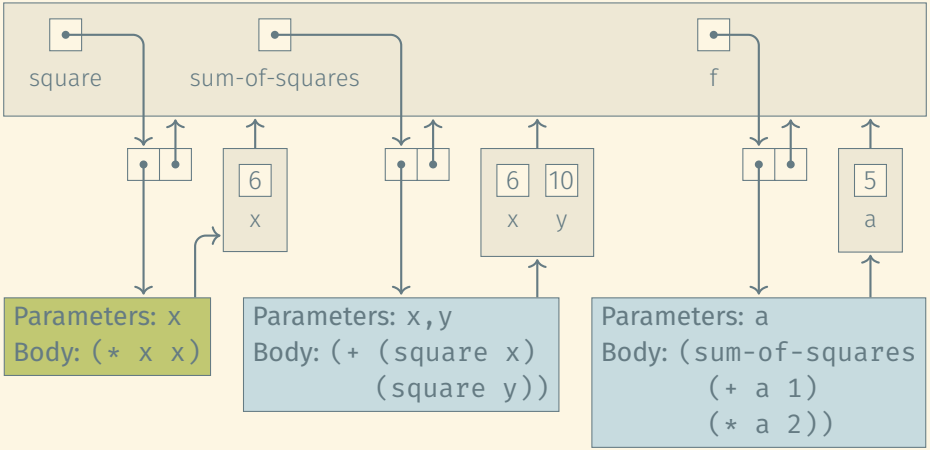
Invocation: `(f 5)`



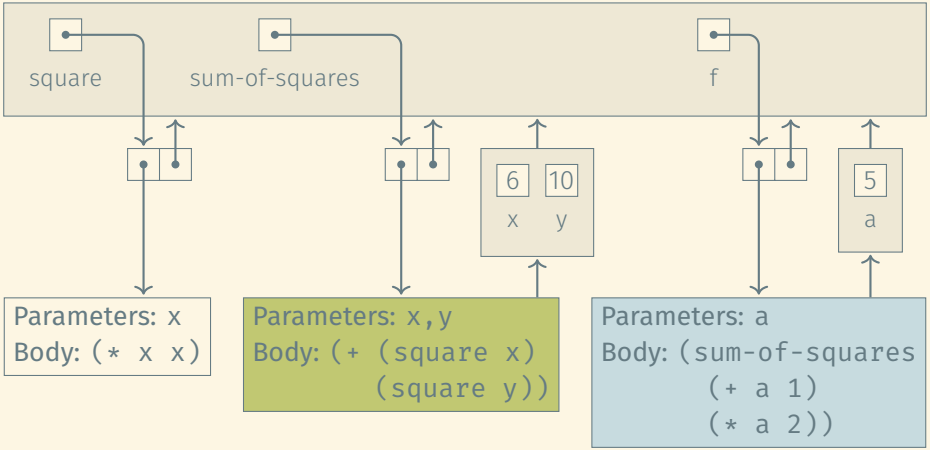
Invocation: `(f 5)`



Invocation: `(f 5)`

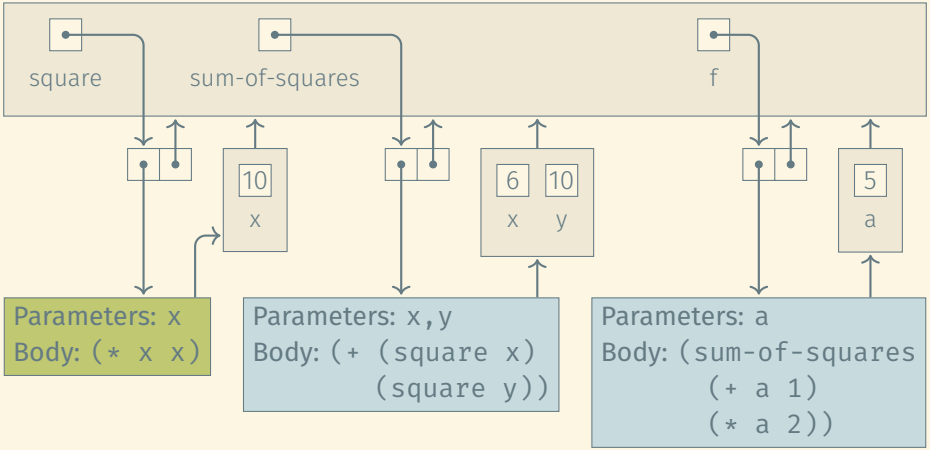


Invocation: `(f 5)`

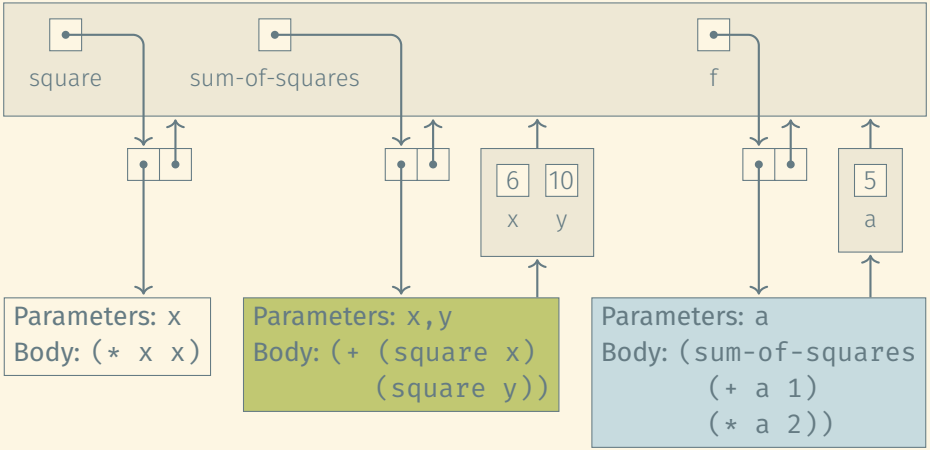


CLOSURES: EXTENDED EXAMPLE (2)

Invocation: `(f 5)`

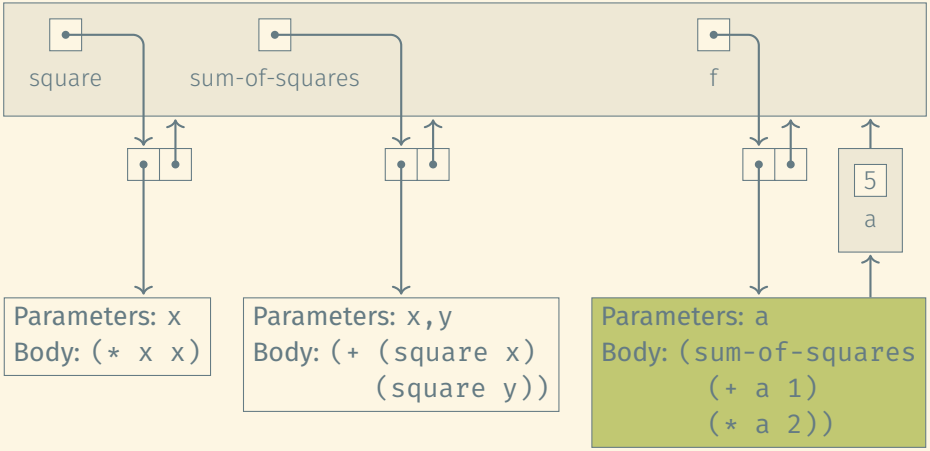


Invocation: `(f 5)`



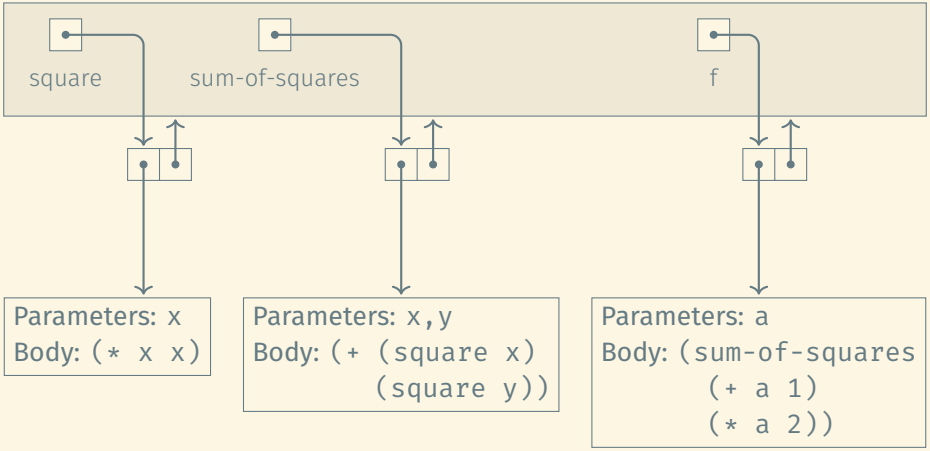
CLOSURES: EXTENDED EXAMPLE (2)

Invocation: `(f 5)`



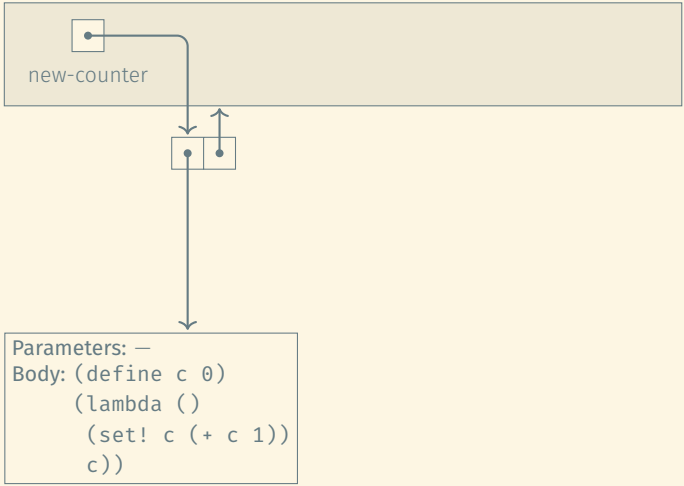
CLOSURES: EXTENDED EXAMPLE (2)

Invocation: `(f 5)`



CLOSURES SIMULATE OBJECTS

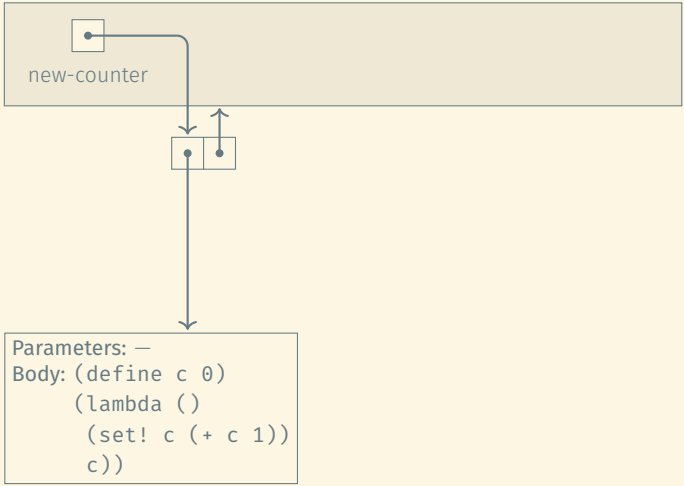
```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

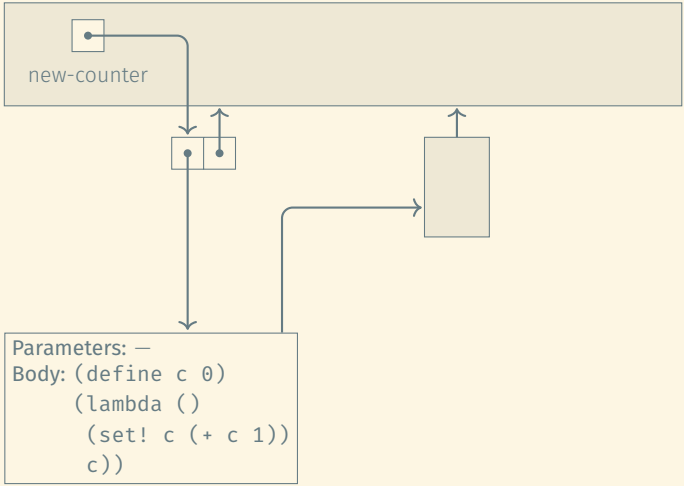
```
> (define cnta
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

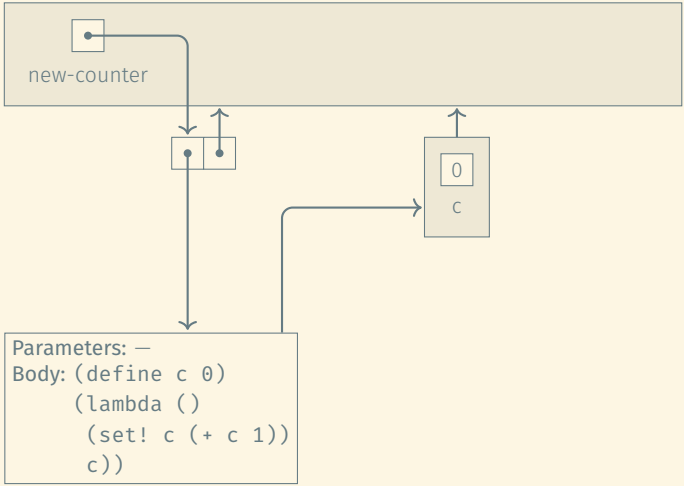
```
> (define cnta
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

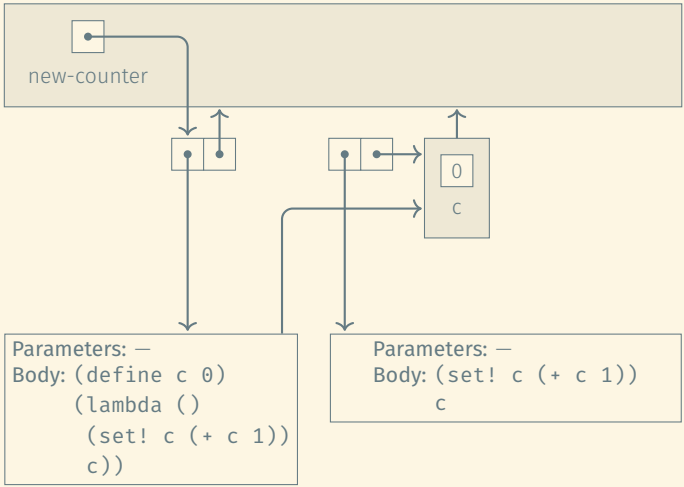
```
> (define cnta
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

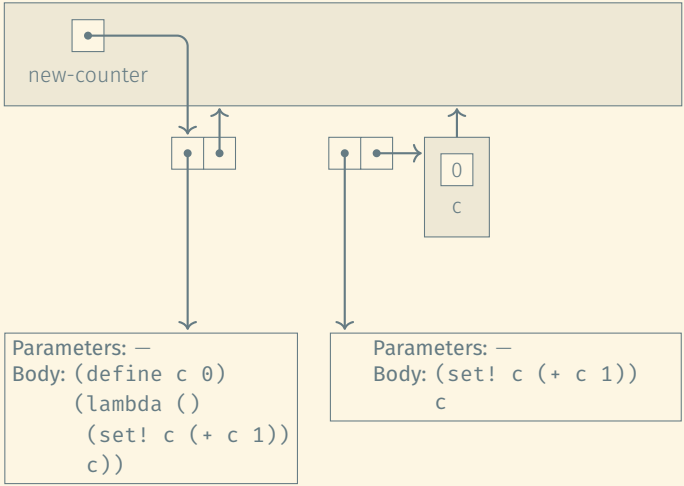
```
> (define cnta
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

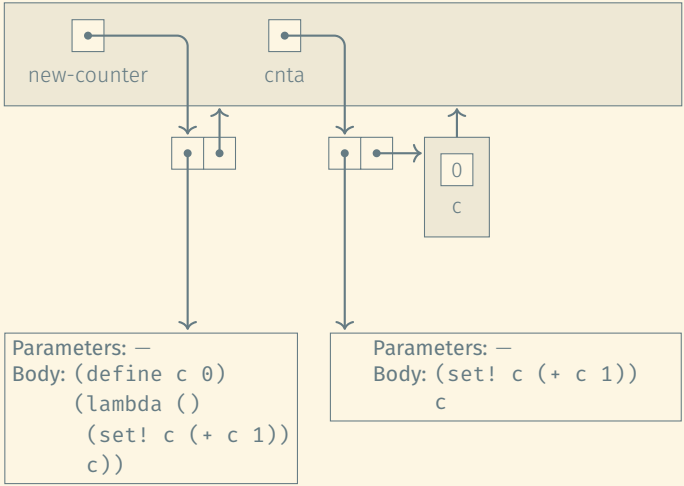
```
> (define cnta
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

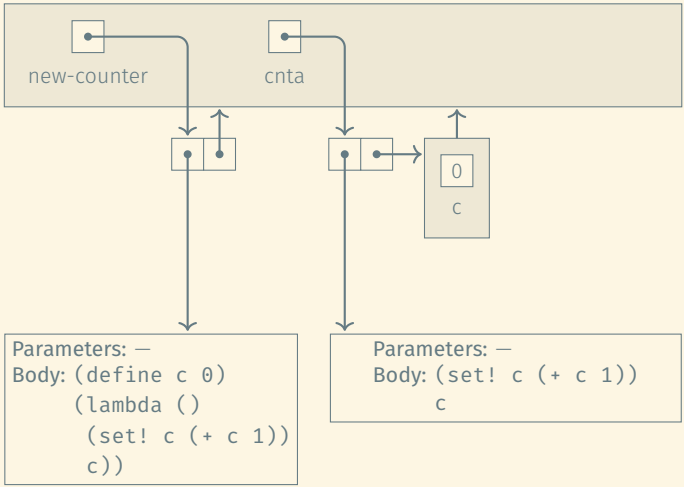
```
> (define cnta
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

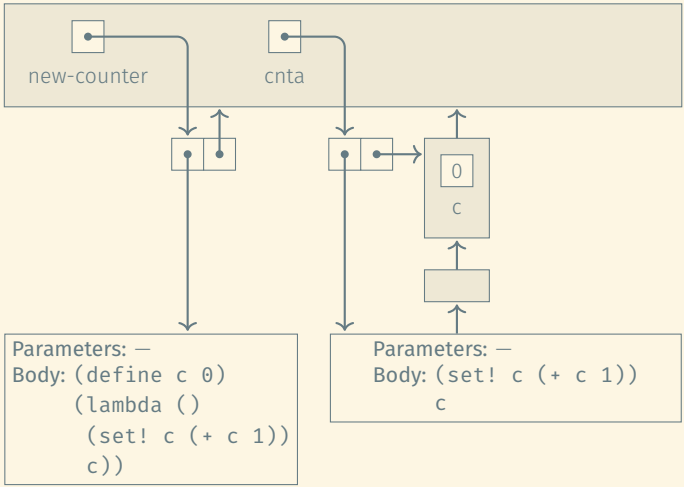
```
> (define cnta
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

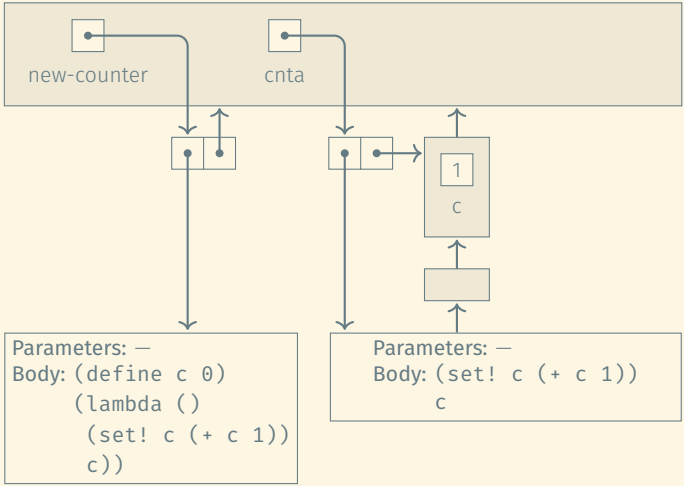
```
> (define cnta
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

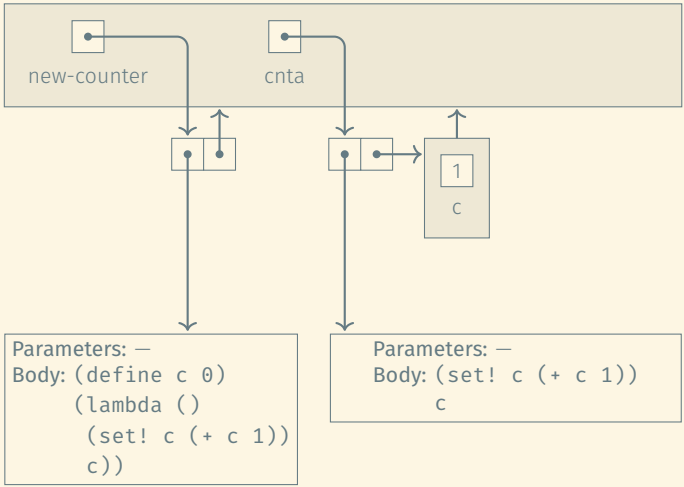
```
> (define cnta
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

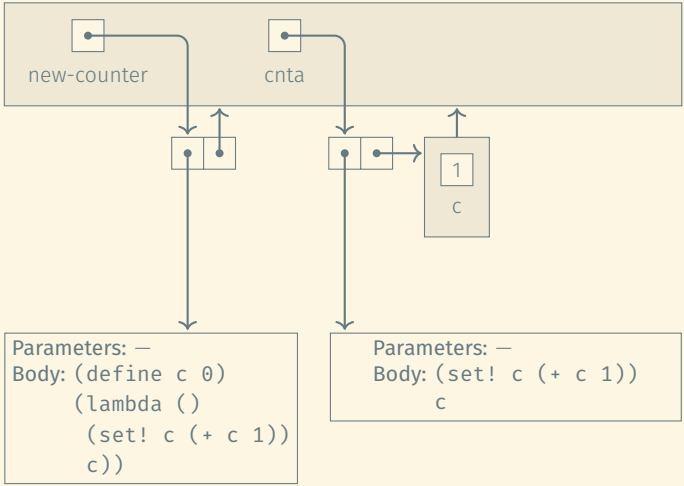
```
> (define cnta
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

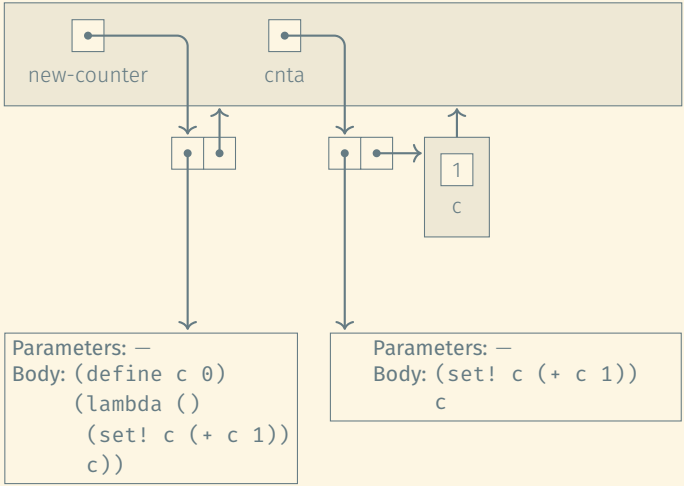
```
> (define cnta
    (new-counter))
> (cnta)
1
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

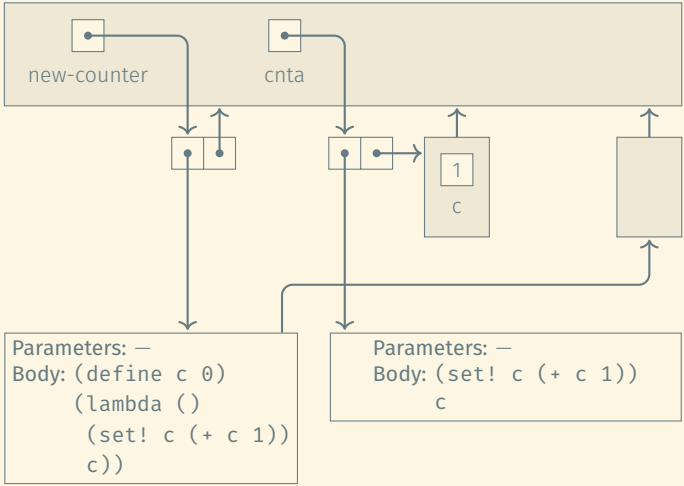
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

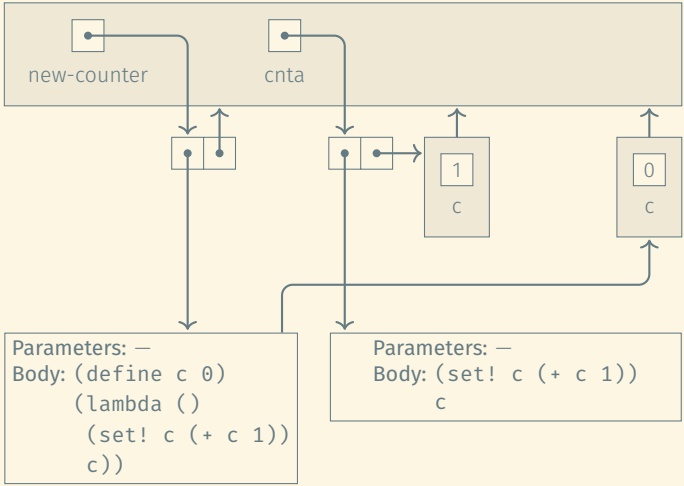
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

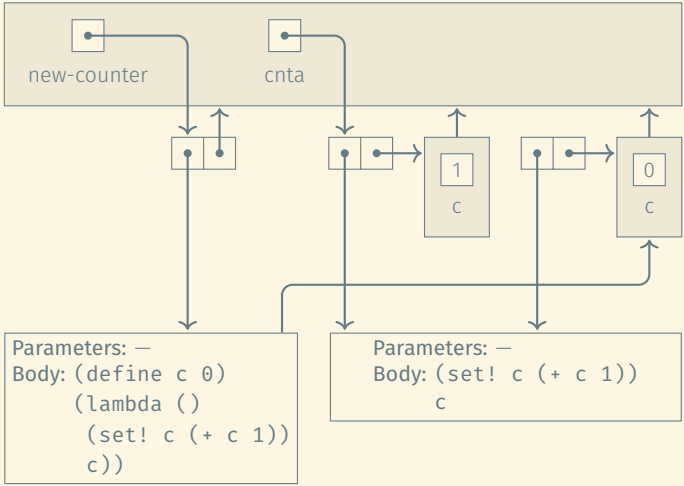
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

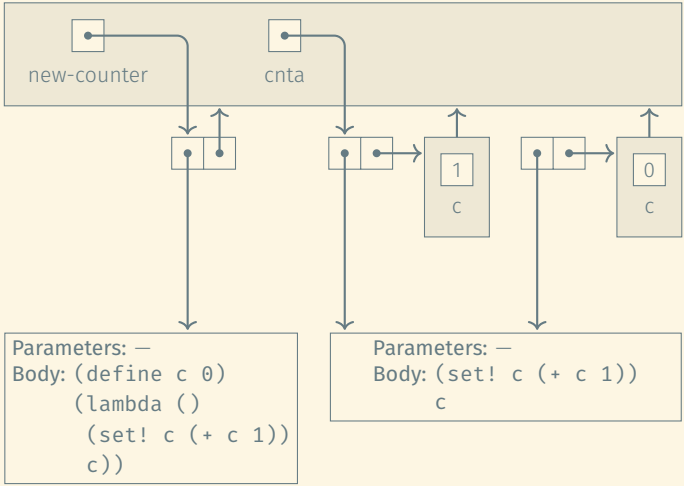
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

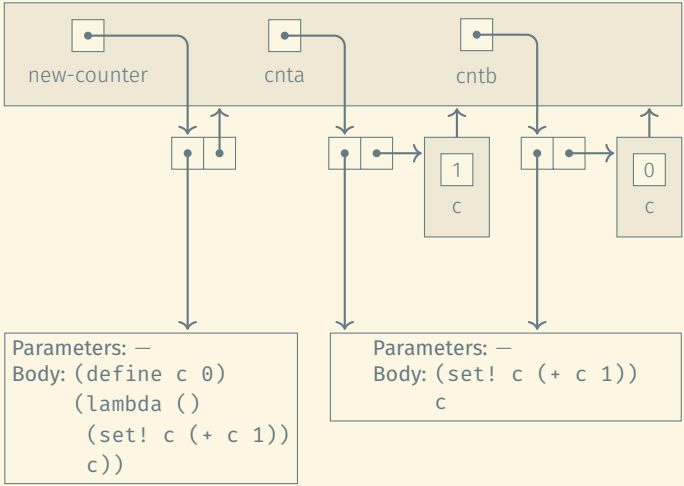
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

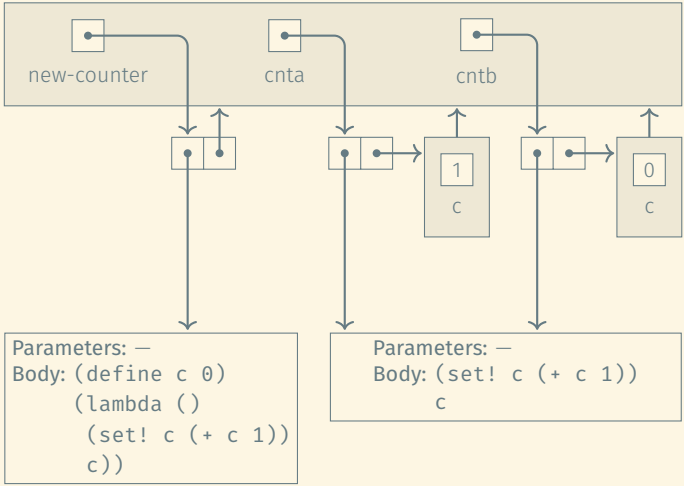
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

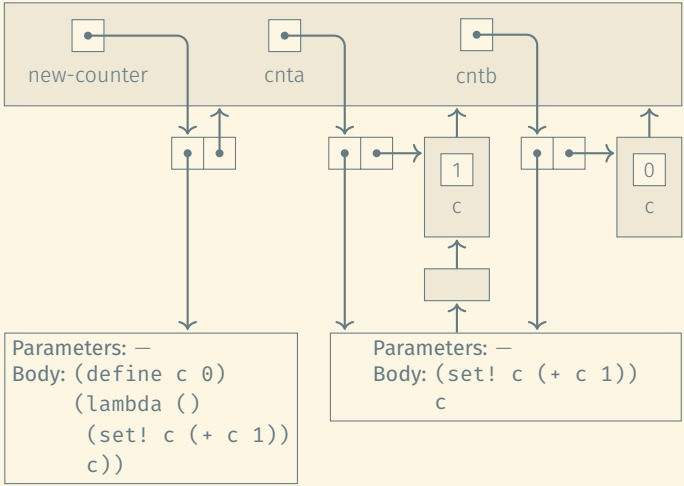
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

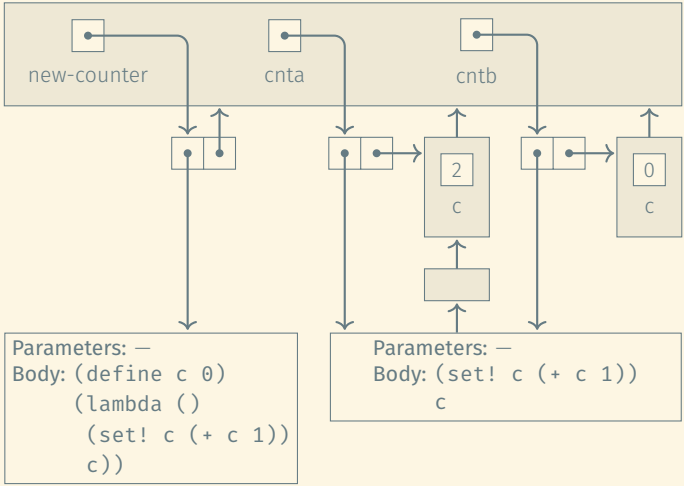
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

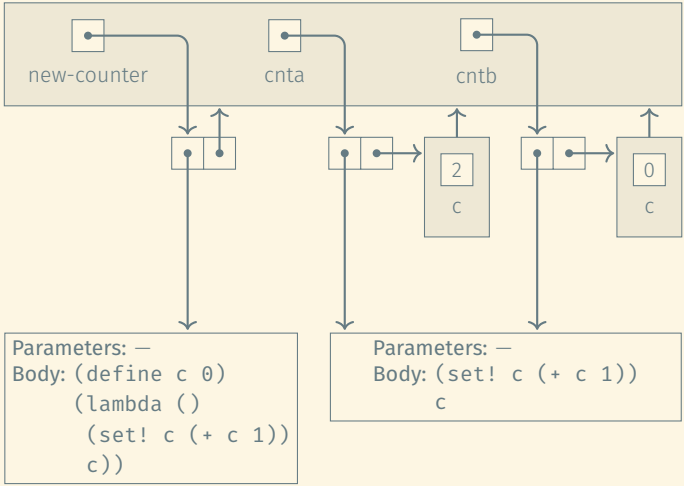
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

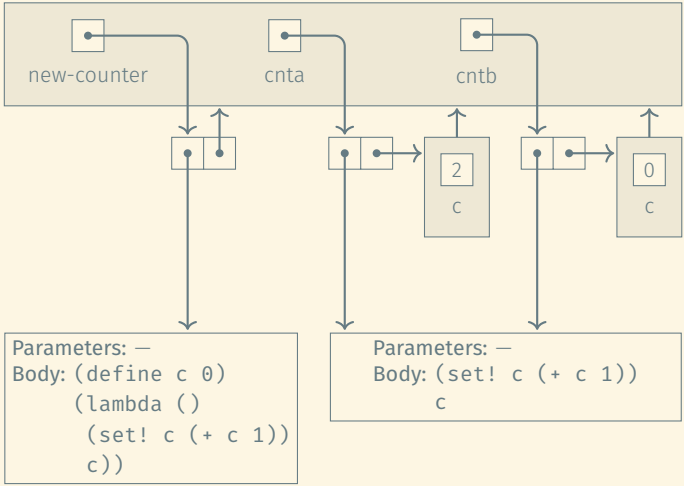
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

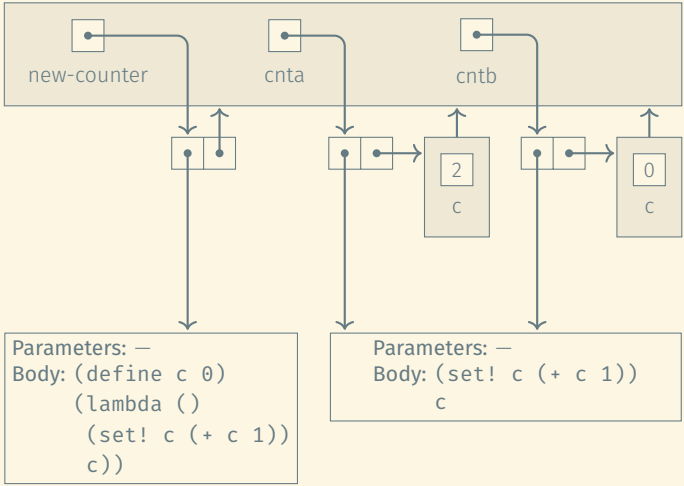
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
2
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

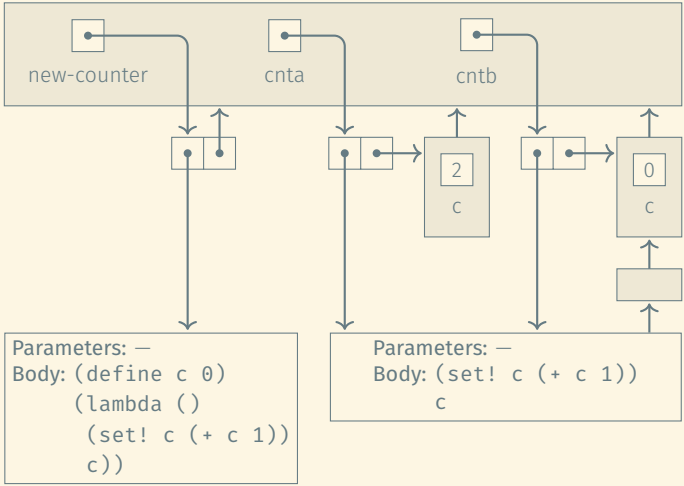
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
2
> (cntb)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

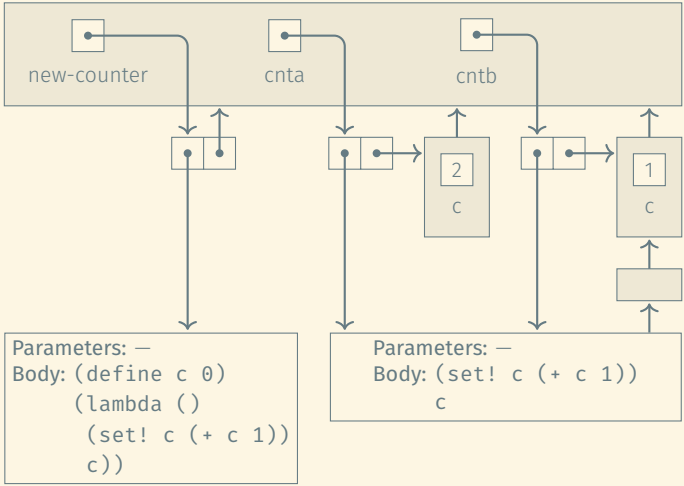
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
2
> (cntb)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

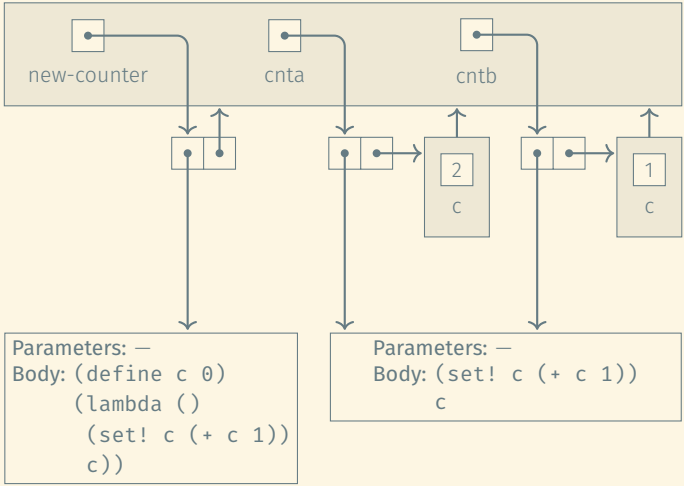
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
2
> (cntb)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

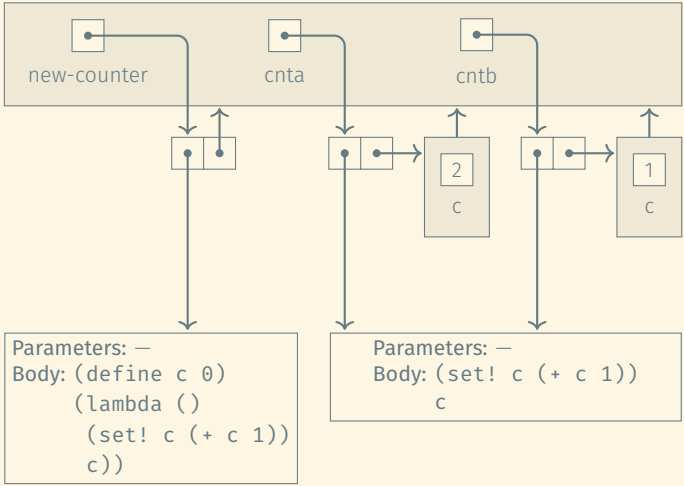
```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
2
> (cntb)
```



CLOSURES SIMULATE OBJECTS

```
(define (new-counter)
  (define c 0)
  (lambda ()
    (set! c (+ c 1))
    c))
```

```
> (define cnta
    (new-counter))
> (cnta)
1
> (define cntb
    (new-counter))
> (cnta)
2
> (cntb)
1
```



Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Motivation

Enable programmers in a team to work independently on different parts of a project.

Motivation

Enable programmers in a team to work independently on different parts of a project.

Requirements

- Modules need to interact with each other through well defined interfaces.
- Internals of modules should be hidden from other modules to avoid unwanted coupling.

- Static variables (in C) provide “private objects” to a single subroutine.

ENCAPSULATION: FROM STATIC VARIABLES TO CLASSES

- Static variables (in C) provide “private objects” to a single subroutine.
- Modules provide the same set of “private objects” to a group of subroutines. (Essentially a single instance of a class.)

ENCAPSULATION: FROM STATIC VARIABLES TO CLASSES

- Static variables (in C) provide “private objects” to a single subroutine.
- Modules provide the same set of “private objects” to a group of subroutines. (Essentially a single instance of a class.)
- Module types (in ML) can be instantiated, effectively acting like classes but without inheritance.

ENCAPSULATION: FROM STATIC VARIABLES TO CLASSES

- Static variables (in C) provide “private objects” to a single subroutine.
- Modules provide the same set of “private objects” to a group of subroutines. (Essentially a single instance of a class.)
- Module types (in ML) can be instantiated, effectively acting like classes but without inheritance.
- Classes add inheritance to module types.

STATIC VARIABLES IN C: EXAMPLE 1

Each invocation produces a different “variable name” starting with the letter `l`:

```
void new_name(char *s, char l) {
  /* This array is automatically filled with
     zeroes when initialized */
  static short int name_nums[52];

  int index = l >= 'a' && l <= 'z'
            ? l - 'a'
            : l - 'A' + 26;
  sprintf(s, "%c%d\0", l, name_nums[index]++);
}
```

STATIC VARIABLES IN C: EXAMPLE 2

Save time by compiling a regular expression only the first time it is used (using the regex library by Henry Spencer):

```
int match_some_dates(char *s) {
  static regexp *date = NULL;
  if (date == NULL) {
    date = regcomp("[0-9][0-9]? (Jan|Feb) 200[4-9]");
  }
  return (regexec(date, s) == 0);
}
```

MODULES IN MODULA-2: EXAMPLE

```
MODULE stack;

IMPORT element, stack_size;

EXPORT push, pop;

TYPE stack_index = [1..stack_size];

VAR s   : ARRAY stack_index OF element;
    top : stack_index;

PROCEDURE push(elem : element);
BEGIN ... END;

PROCEDURE pop() element;
BEGIN ... END;

BEGIN
  top := 1;
END stack.
```

```
MODULE main;

TYPE element = ...;

CONST stack_size = ...;

FROM stack IMPORT push, pop;

VAR x, y : element;

BEGIN
  ...
  push(x);
  ...
  y := pop;

END main.
```

Visibility is specified using explicit `IMPORT` and `EXPORT` statements.

This is an example of closed scopes (as opposed to open scopes where bindings from “outside” are freely passed into the scope).

Closed scopes force programmers to clearly document the interface.

C has no support for modules.

Java, C#, Perl, Python, Ada, and Haskell provide selectively open scopes.

Separate compilation units

Examples: C, C++

- Include files simulate EXPORT lists.
- No protection against name clashes between “modules”.

Separate compilation units

Examples: C, C++

- Include files simulate EXPORT lists.
- No protection against name clashes between “modules”.

Namespaces

Example: C++

Help address the issue with name clashes in different compilation units in C++.

Separate compilation units

Examples: C, C++

- Include files simulate EXPORT lists.
- No protection against name clashes between “modules”.

Packages

Examples: Java, Perl, Ada

Namespaces

Example: C++

Help address the issue with name clashes in different compilation units in C++.

CONSTRUCTS RELATED TO MODULES

Separate compilation units

Examples: C, C++

- Include files simulate EXPORT lists.
- No protection against name clashes between “modules”.

Namespaces

Example: C++

Help address the issue with name clashes in different compilation units in C++.

Packages

Examples: Java, Perl, Ada

Clusters

Example: Clu

A MODULE DEFICIENCY

The stack module presented earlier cannot be used to provide multiple stacks to an application that requires them.

A MODULE DEFICIENCY

The stack module presented earlier cannot be used to provide multiple stacks to an application that requires them.

Possible solutions:

The stack module presented earlier cannot be used to provide multiple stacks to an application that requires them.

Possible solutions:

- Not really a solution: Duplicate the code over multiple modules with the same name.

The stack module presented earlier cannot be used to provide multiple stacks to an application that requires them.

Possible solutions:

- Not really a solution: Duplicate the code over multiple modules with the same name.
- A module that provides explicit means to create, manage, and destroy multiple stacks. (Requires the stack as an argument to each stack function.)

The stack module presented earlier cannot be used to provide multiple stacks to an application that requires them.

Possible solutions:

- Not really a solution: Duplicate the code over multiple modules with the same name.
- A module that provides explicit means to create, manage, and destroy multiple stacks. (Requires the stack as an argument to each stack function.)
- Module types (e.g., Simula, Euclid, ML) are modules that can be instantiated.

A MODULE DEFICIENCY

The stack module presented earlier cannot be used to provide multiple stacks to an application that requires them.

Possible solutions:

- Not really a solution: Duplicate the code over multiple modules with the same name.
- A module that provides explicit means to create, manage, and destroy multiple stacks. (Requires the stack as an argument to each stack function.)
- Module types (e.g., Simula, Euclid, ML) are modules that can be instantiated.
- Go all the way to classes.

Every instance of a module type or class has a separate copy of the module type's or class's variables.

Every instance of a module type or class has a separate copy of the module type's or class's variables.

Classes

Module types + inheritance

Every instance of a module type or class has a separate copy of the module type's or class's variables.

Classes

Module types + inheritance

```
public class Stack {
  private int stack_size;
  private element[] s;
  private int top = 0;

  public void push(element x) { ... }

  public element pop() { ... }
}
```

```
...
stack A, B;
element x, y;
...
A.push(x);
...
y = B.pop();
...
```

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Language concepts:

- Names
- Binding
- Scoping
- Modules and classes
- Aliasing and overloading

Implementation:

- Stack frames
- Heap management
- Static chains
- Closures

Aliasing

- More than one name is bound to the same object (reference, pointers, ...).
- Makes compiler optimization difficult.

Examples:

```
int a, b, *p, *q;
a = *p; *q = 3; b = *p;
```

```
double sum, sum_of_squares;
void accumulate(double &x) {
  sum += x;
  sum_of_squares += x * x;
}
accumulate(sum);
```

Overloading

The same name is bound to more than one object.

Example:

```
int    operator +(const int    &a, const int    &b);
string operator +(const string &a, const string &b);
```

PROBLEMS WIH ALIASING

Aliases make code more confusing and make the resulting bugs hard to find.

Optimization of code becomes difficult if not impossible.

Aliases make code more confusing and make the resulting bugs hard to find.

Optimization of code becomes difficult if not impossible.

`restrict` keyword in C99:

- Used by the programmer to tell the compiler that a given pointer is the only means used to update the memory location it references.
- Allows the compiler to perform optimizations that would be impossible in the presence of aliasing.
- The resulting optimization may lead to even more obscure bugs if the programmer doesn't keep their promise and introduces aliases for this pointer.

Most languages have some form of overloading (e.g., arithmetic operators).

We normally do not think about this type of overloading, as we simply think about “doing math with numbers” and the right thing happens.

The amount of overloading is fixed because the programmer cannot add new “versions” of the same function or operator.

USER-DEFINED OVERLOADING

Allows the programmer to define new functions with the same name as an existing built-in or previously user-defined function.

USER-DEFINED OVERLOADING

Allows the programmer to define new functions with the same name as an existing built-in or previously user-defined function.

Supported in many modern object-oriented languages:

- C++, C#: `A.operator +(B)`
- Ada: `"+"(A, B)`
- FORTRAN90: Interface construct to associate e.g., “+” with some binary function

USER-DEFINED OVERLOADING

Allows the programmer to define new functions with the same name as an existing built-in or previously user-defined function.

Supported in many modern object-oriented languages:

- C++, C#: `A.operator +(B)`
- Ada: `"+"(A, B)`
- FORTRAN90: Interface construct to associate e.g., "+" with some binary function

Requires a dispatch mechanism that allows the compiler to choose the correct implementation of the function based on the arguments that are provided:

Should I use the `int`, `float` or `string` version of "+"?

PROS AND CONS OF USER-DEFINED OVERLOADING

Pro:

User-defined overloading makes a language more powerful and expressive with a potential to increase clarity (e.g., arithmetic operators for complex numbers).

PROS AND CONS OF USER-DEFINED OVERLOADING

Pro:

User-defined overloading makes a language more powerful and expressive with a potential to increase clarity (e.g., arithmetic operators for complex numbers).

Con:

It has the potential for tremendous confusion if the behaviour associated with an overloaded function or operator does not match what one would intuitively expect based on its name.

Coercion

The compiler automatically converts an object into an object of another type when required.

Coercion

The compiler automatically converts an object into an object of another type when required.

Examples:

- Coercion from `int` to `float` when one operand of an arithmetic expression is a `float`.

Coercion

The compiler automatically converts an object into an object of another type when required.

Examples:

- Coercion from `int` to `float` when one operand of an arithmetic expression is a `float`.
- In Java, `"" + o` automatically calls `o.toString()`.

Coercion

The compiler automatically converts an object into an object of another type when required.

Examples:

- Coercion from `int` to `float` when one operand of an arithmetic expression is a `float`.
- In Java, `"" + o` automatically calls `o.toString()`.
- In C++, `Foo o = 5;` implicitly calls the constructor `Foo::Foo(int)` unless it is marked as `explicit`.

Generics and polymorphism

- Single body of code
- Behaviour is customized

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

Generics and polymorphism

- Single body of code
- Behaviour is customized

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

Templates (C++)

Separate copies of the code generated by compiler for each type

```
std::vector<int> int_vec;
std::vector<char> char_vec;
```

SUMMARY

- Bindings associate names with the objects they refer to.
- Scoping rules (lexical, dynamic) determine what different names in different places in a program or at different times during the program execution.
- Object life time is linked to where objects are stored.
- Stack frames and closures are used to manage space for local variables.
- The heap stores explicitly allocated objects.
- Static chains are used to manage lexical scoping when using stack frames to manage local variables.
- Modules and classes help break large projects into parts that interact via well-defined interfaces.