CSCI 2041: Curried Functions and Closures

Chris Kauffman

Last Updated:
Tue Dec 11 20:37:50 CST 2018

Logistics

Goals

- Shallow/Deep Equality Wrap-up
- Curried Functions
- Scope and Closures

Next Week

- Mon: Review
- Wed: **Exam 2**
- Fri: Lecture

Midterm Feedback Survey

- Overall Positive
- Results Posted

Assignment 3 `multimanager`

- Manage multiple lists
- Records to track lists/undo
- `option` to deal with editing
- Higher-order funcs for easy bulk operations
- Due Mon 10/22

Exercise: Warm-Up w/ Deep vs. Shallow Equality

```
# let a = "hi";;
# let b = "hi";;
# let c = a;;
# [a=b;  a==b;  a=c;  a==c];;   (* EXPLAIN true/false values *)
  [true; false; true; true]

# let p = (a,a);;
# let u = (a,b);;
# let t = (a,c);;
# let x = p;;
# [p=u;  p==u;  p=t;  p==t;  p=x;  p==x];;   (* EXPLAIN *)
  [true; false; true; false; true; true]
```

Answers: Warm-Up w/ Deep vs. Shallow Equality

```
# let a = "hi";;
# let b = "hi";;
# let c = a;;
# [a=b;  a==b;  a=c;  a==c];;   (* EXPLAIN true/false values *)
  [true; false; true; true]
```

- `a` and `c` are the exact same string, point to the same memory location; they are therefore both deeply and shallowly equal
- `a` and `b` point to different locations so NOT shallowly equal (physically different locations)
- However, `a`/`c` both point to identical strings so deeply equal (structurally equal)

```
# let p = (a,a);;
# let u = (a,b);;
# let t = (a,c);;
# let x = p;;
# [p=u;  p==u;  p=t;  p==t;  p=x;  p==x];;   (* EXPLAIN *)
  [true; false; true; false; true; true]
```

- `p` / `u` / `t` all point to different pairs, therefore not shallowly equal
- All point to structurally identical pair structure of `("hi","hi")` so deeply equal
- `p` / `x` point to the same location: shallow and deeply equal

Aside: The Saga of OCaml's `string` Type

- Older versions of OCaml, `string` was an array of characters, mutable by default

```
# let str = "hello!";;
# str.[0] <- 'y';;
# str.[5] <- 'w';;
# str
- : : string = "yellow"
```

- An exception to the *immutable by default*
- Led to interesting "situations" like

```
# let a = "hi";;
val a : string = "hi"
# let b = a;;
val b : string = "hi"
# a.[0] <- 'f';;
# b;;
- : string = "fi"
```

- Release 4.02, **deprecated** mutating strings
- `string`'s are now totally immutable like Java/Python
- For now string constants each get allocated to separate memory areas but this may change in the future (as it is in Java)
- Type `bytes` was introduced as the mutable alternative

```
# let b = Bytes.of_string "hello!"
# Bytes.set b 0 'y';;
- : unit = ()
# Bytes.set b 5 'w';;
- : unit = ()
# Bytes.to_string b;;
- : string = "yellow"
```

Curried Stuff 1

Source: Lord Byron's Kitchen "Curried Lentils"

The Delicious Kind

Curry (plural curries): an umbrella term referring to a number of dishes originating in the Indian subcontinent.
–Wikipedia "Curry"

Curried Stuff 2

The Function Kind

Haskell Brooks Curry (1900 -1982): An American mathematician and logician... There are three programming languages named after him, Haskell, Brook and Curry, as well as the concept of **currying, a technique used for transforming functions** in mathematics and computer science.
–Wikipedia "Haskell Curry"

Source: Wikipedi "Haskell Curry"

Currying Enables Partial Application

- From `partial_apply.ml`, consider the `add_on` function with type `int -> int -> int`
- Applying it two parameters gives an `int`: a "normal" value
- Applying it to one parameter gives `int -> int`: a function

```
 1  let add_on a b =                    (* standard function of 2 params *)
 2    a + b
 3  ;;                                  (* automatically curried *)
 4  (* val add_on : int -> int -> int *)
 5
 6  let sevenA   = add_on 5 2;;         (* apply to 2 parameters *)
 7  let elevenA  = add_on 5 6;;         (* result: int *)
 8
 9  let add_5A   = add_on 5;;           (* apply to 1 parameter *)
10  let add_9A   = add_on 9;;           (* result: int -> int *)
11
12  let sevenB  = add_5A 2;;            (* add_5 is a function of 1 parameter *)
13  let elevenB = add_5A 6;;            (* apply to 1 param results in int *)
```

- OCaml functions are automatically **curried**
- Allows **partial application** of all normal functions
- This is a somewhat rare feature found in ML and Haskell
- Not default in Lisp, Scheme, Python, Java, C, etc...

Functions as Return Values

- Functions can produce functions as their return value
- The typing for these looks unexpectedly bland due to currying (lol, bland curry)

```
let add_on a b =                    (* standard function of 2 params *)
  a + b
;;                                  (* automatically curried *)
(* val add_on : int -> int -> int *)


let add_on_slam a =                 (* standard function of 1 param *)
  (fun b -> a + b)                  (* returns a function of 1 param *)
;;
(* val add_on_slam : int -> int -> int      same type as add_on *)

let sevenC = add_on_slam 5 2;;      (* apply just as 2 param version *)

let add_5B = add_on_slam 5;;        (* call with single param: curried *)
let sevenD = add_5B 2;;             (* call with additional parameter *)
```

Central Idea of Curried Functions

Standard function declaration syntax and lambda `fun` syntax is internally converted to the following pattern of function

```
(* Prior versions 'add_on' and 'add_on_slam' are internally converted
   to the version 'add_on_dlam' below. *)
let add_on_dlam =              (* bind name 'add_on_dlam' to ... *)
  (fun a ->                    (* a function of 1 param which returns... *)
    (fun b ->                  (* a function of 1 param which returns... *)
      a+b))                    (* an answer through addition *)
;;
(* val add_on_dlam : int -> int -> int   same type as previous versions *)
```

- All functions are 1-param functions internally
- Function **applications** "use up" one parameter[1], may return another function, repeated application yields "normal" values
- Internal application and return mechanisms are simplified
- Curried functions are more flexible

[1] We will see that each function application produces a *closure* which has some variables bound and a pointer to code to execute. Code can fully execute once all free variables are bound.

Most Standard "Operators" are Curried Functions

- Recall infix arithmetic ops are actually functions
- Can retrieve their types by parenthesizing them
- Can partially apply them as they are curried

```
# (+);;
- : int -> int -> int = <fun>
# let add7 = (+) 7;;
val add7 : int -> int = <fun>
# add7 3;;
- : int = 10

# (^);;
- : string -> string -> string = <fun>
# let affix_prefix = (^) "pre-";;
val affix_prefix : string -> string = <fun>
# affix_prefix "ocaml";;
- : string = "pre-ocaml"
# affix_prefix "cognition";;
- : string = "pre-cognition"
```

Exercise: Complete via Currying

Fill in definitions for the below functions using a 1-liner and partial application of standard functions.

```
let print_greeting = ... ;;
(* val print_greeting : string -> unit
   prints a greeting with format "Greetings, XXX: what flavor curry would you like?\n"
   with XXX filled in with a parameter string

   # print_greeting "Elfo";;
   Greetings, Elfo: what flavor curry would you like? *)

let sumlist = ... ;;
(* val sumlist : int list -> int     sum an a list of integers
   # sumlist [9;5;2];;
   - : int = 16 *)

let divall = ... ;;
(* val divall : int -> int list -> int    Divide an integer by all integers in a list
   # divall 100 [2;5];;
   - : int = 10
   # divall 360 [5;6;4];;
   - : int = 3 *)

let kaufenate = ... ;;
(* val kaufenate : string list -> string list
   Prepend the string "Kauf" to a list of strings. Two curry opportunities.
   # kaufenate ["money"; "nix"; "tastic"];;
   - : string list = ["Kaufmoney"; "Kaufnix"; "Kauftastic"] *)
```

Answers: Complete via Currying

See `curried_applications.ml`

```
let print_greeting =
  printf "Greetings, %s: what flavor curry would you like?\n"
;;

let sumlist =
  List.fold_left (+) 0;;
;;

let divall =
  List.fold_left (/)
;;

let kaufenate =
  List.map ((^) "Kauf")
;;
```

Multiple Args vs Tuple Argument

These two functions have different type signatures

```
let add_on a b =                    (* standard function of 2 params *)
  a + b                             (* val add_on : int -> int -> int *)
;;                                  (* automatically curried *)

let add_together (a,b) =            (* standard sytnax, 1 param: a pair *)
  a + b                             (* int * int -> int *)
;;                                  (* can't curry on tuple *)

let eightA = add_together (3,5);;   (* must apply to complete pair *)
```

- Tuple arguments come as a package: can't be curried

Limits of Currying

- Curried functions must apply arguments in order
- Limits flexibility: not useful if later parameter is known but earlier param is **free**
- Can always define a standard function

```
# let affix_suffix str = str ^ "-y";;        (* currying doesn't help *)
val affix_suffix : string -> string = <fun> (* b/c first arg is free *)
# affix_suffix "unix";;
- : string = "unix-y"
# affix_suffix "mone";;
- : string = "mone-y"
```

- Not all "operators" are functions, some are Algebraic type constructors which usually take tuple arguments
- Example: **Cons operator** `::` is **not a function** so is not curried

```
# (::) 1;;
Characters 0-6:
  (::) 1;;
  ^^^^^^
Error: The constructor :: expects 2 argument(s),
       but is applied here to 1 argument(s)
```

Exercise: A Quick Review Puzzle

- Examine the code to the right
- What gets printed?
- Why do certain things get printed? *Relate your answer to lexical scope versus dynamic scope.*

```
 1  let x = "Mario";;
 2  let print_player () =
 3    printf "%s\n" x;
 4  ;;
 5
 6  let x = "Luigi";;
 7  print_player ();;
 8
 9  let e = ref "Bowser";;
10  let print_enemy () =
11    printf "%s\n" !e;
12  ;;
13
14  e := "Magikoopa";;
15  print_enemy ();;
16
17  let e = "Wario";;
18  print_enemy ();;
```

Answers: A Quick Review Puzzle

- **Lexically Scoped**: names refer to the value bound at the time of creation, NOT to the "current" binding
- **Dynamically Scoped**: names refer to the current value bound to the name

OCaml, like most programming languages, is *lexically scoped*.

```
 1  let x = "Mario";;              (* x is bound *)
 2  let print_player () =          (* print_player uses *)
 3    printf "%s\n" x;             (* x, remember its value *)
 4  ;;
 5
 6  let x = "Luigi";;              (* rebind x to new value *)
 7  print_player ();;              (* "Mario" : original x value is retained *)
 8
 9  let print_player2 () =
10    printf "%s\n" x;
11  ;;
12
13  print_player2 ();;
14  print_player1 ();;
15
16  let x = "Princess";;
17
18  let e = ref "Bowser";;         (* ref to string *)
19  let print_enemy () =           (* print_enemy uses value e *)
```

Variable Escape

- A name/value binding is said to **escape** its scope if it is used in some inner scope that outlives the housing scope
- Possible when functions are given as return values
- Rules of lexical scoping dictate that local name/value bindings must be "saved" somehow so that when returned function runs, original values are used
- Applicable to all OCaml functions due to automatic currying of functions

Variable Escape Example

```
 1  let afunc paramX =                (* a function taking a paramter  *)
 2    let localA = "less" in          (* local variables *)
 3    let localB = "greater/eq" in
 4    let retfun paramY =             (* local function to be returned *)
 5      if paramX < paramY then       (* paramX "escapes" into retfun *)
 6        localA                      (* localA "escapes" into retfun *)
 7      else
 8        localB                      (* localB "escapes" into retfun *)
 9    in
10    retfun                          (* return a function *)
11  ;;
12
13  let res  = afunc 10 12;;          (* no need to save params/locals *)
14  (* val res : string = "less" *)
15
16  let gt10 = afunc 10;;             (* save paramX=10, etc somehow *)
17  (* val gt10 : int -> string *)
18  let gt42 = afunc 42;;             (* save paramX=42, etc *)
19  (* val gt42 : int -> string *)
20
21  let localA = "don't care!";;      (* has no effect on evaluation below *)
22
23  let res10_12 = gt10 12;;          (* use paramX=10, evaluate 10 < 12 *)
24  (* "less" *)
25  let res42_12 = gt42 12;;          (* use paramX=42, evaluate 42 < 12 *)
26  (* "greater/eq" *)
```

Frames in the Stack, Frames in the Heap

- Many standard programming languages like C/Java use a **Stack Model** for function execution
  - Function calls push onto the call stack
  - Function return pops off of the stack
- These languages most frequently do not have lambdas combined with functions as return values
- Reason: the Stack Model is insufficient to handle function returns due to variable escape
- Many functional languages like ML and Lisp DO support lambdas and functions as return values which leads to more complex implementation
- In such languages, some/all call frames are allocated in **the heap**, portion of memory managed by garbage collector

Stack Fails with Escaped Variables

```
 1  let afunc paramX =
 2    let localA = "less" in
 3    let localB = "greater/eq" in
 4    let retfun paramY =
 5      if paramX < paramY then
 6        localA
 7      else
 8        localB
 9    in
10    retfun
11  ;;
12
13  let gt10 = afunc 10;;
14  let res  = gt10 18;;
```

On returning from afunc, bindings

```
paramX=10 localA="less" localB="greater"
```

are popped, but are still needed by function bound to gt10.

How can this be resolved?

```
Call stack before afunc returns
| FRAME    | SYMBOL | VALUE          |
|----------+--------+----------------|
| init     | afunc  | <fun>          |
| line:13  | gt10   | ??             |<-+
|          | res    | ??             |  |
|          | ...    |                |  |
|----------+--------+----------------|  |
| afunc    | paramX | 10             |--+
| line:10  | localA | "less"         |
|          | localB | "greater/eq"   |
|          | retfun | <fun>          |
|----------+--------+----------------|

Call stack after afunc returns
| FRAME    | SYMBOL | VALUE          |
|----------+--------+----------------|
| init     | afunc  | <fun>          |
| line:14  | gt10   | <fun>          |
|          | res    | ??             |
|          | ...    |                |
|----------+--------+----------------|
```

Closures

- A **closure**[2] is used to capture state needed for lexically scoped functions to evaluate
- Closures have two parts
  1. **Code**: a pointer to machine instructions to execute
  2. **Environment**: a data structure giving access to all name/value bindings required to execute the code
- OCaml doesn't report whether a `<fun>` involves a closure or not as this is a low-level implementation detail
- A variety of closure implementations exist in practice, vary from one functional language to another and between versions
- Almost all such approaches involve use of the **heap** to allow for eventual garbage collection of closures
- We will consider a very simple, conceptual approach: copy bindings to the heap.

[2] The term "closure" is the source for the name "Clojure", of a modern Lisp implementation with the "j" coming from its connection to the Java platform. Both words are pronounce identically.

Closures in Action 0

Begin with first executable line 13 which calls `afunc`

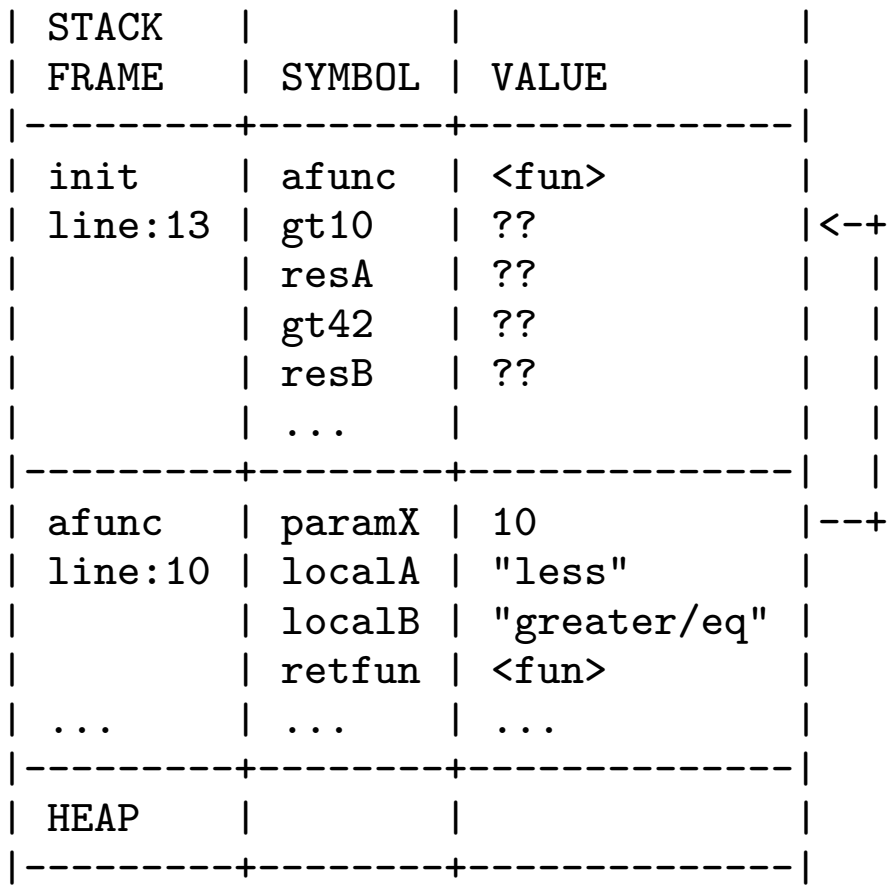
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
>>13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```

STACK FRAME	SYMBOL	VALUE
init	afunc	<fun>
line:13	gt10	??
	resA	??
	gt42	??
	resB	??
	...	
...	...	...
HEAP		

Closures in Action 1

At completion of `afunc`, have several bindings that will escape through returned `retfun`

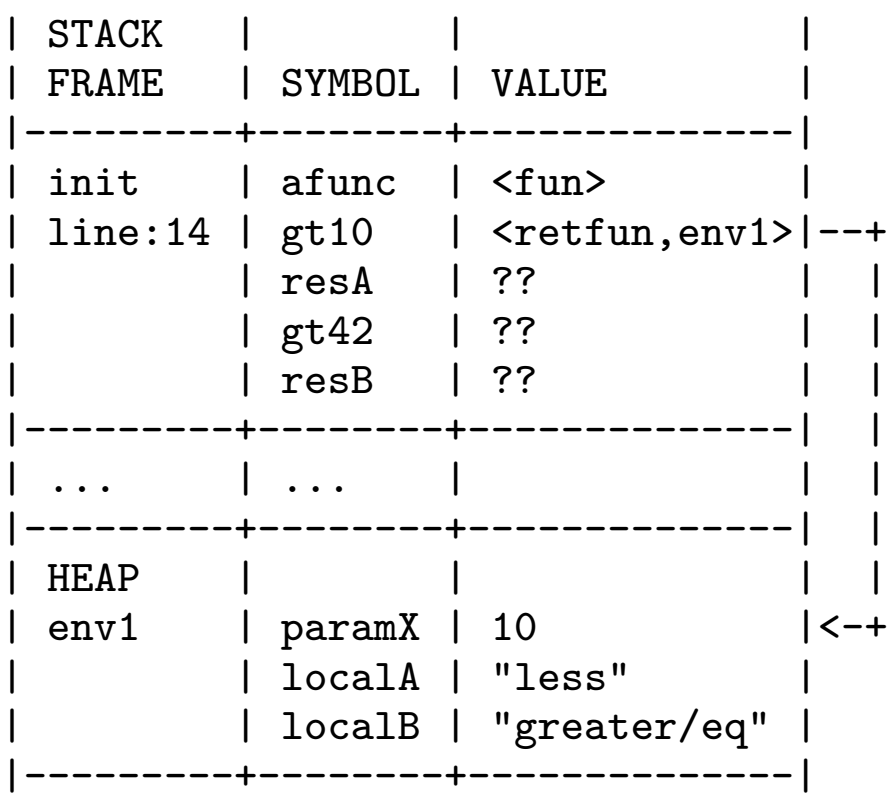
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
>>10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```



Closures in Action 2

A closure is allocated in the heap which preserves the binding environment in which `retfun` existed. This is tracked in the binding for `gt10`.

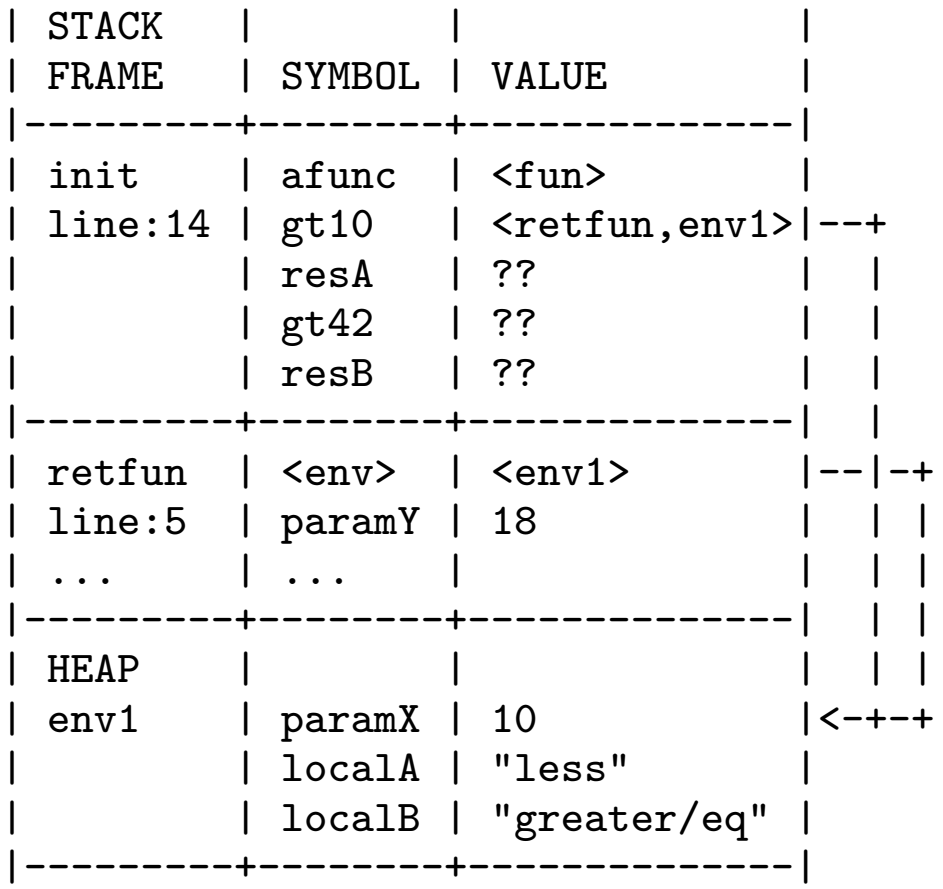
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
>>14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```



Closures in Action 3

Executing `gt10 18` runs code for `retfun` with environment where `paramX=18` and `localA/localB` defined. Compares `10 < 18` and follows `then` branch to line 6.

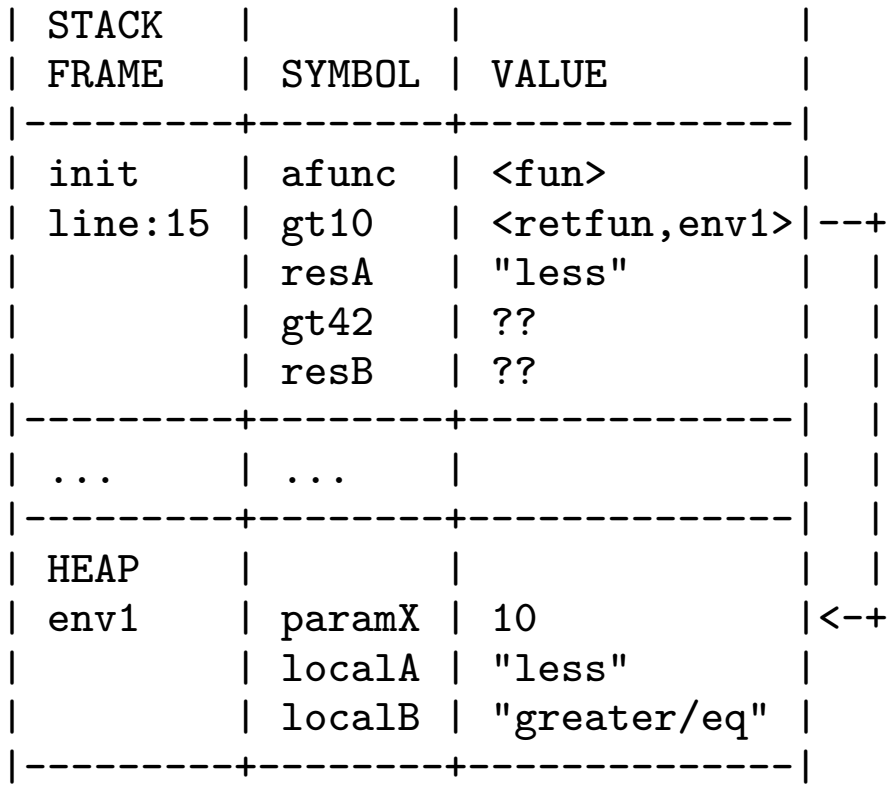
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
>> 5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```



Closures in Action 4

Results bound to `resA` as `"less"`. Next line 15 runs `afunc` again with a different value for `paramX`.

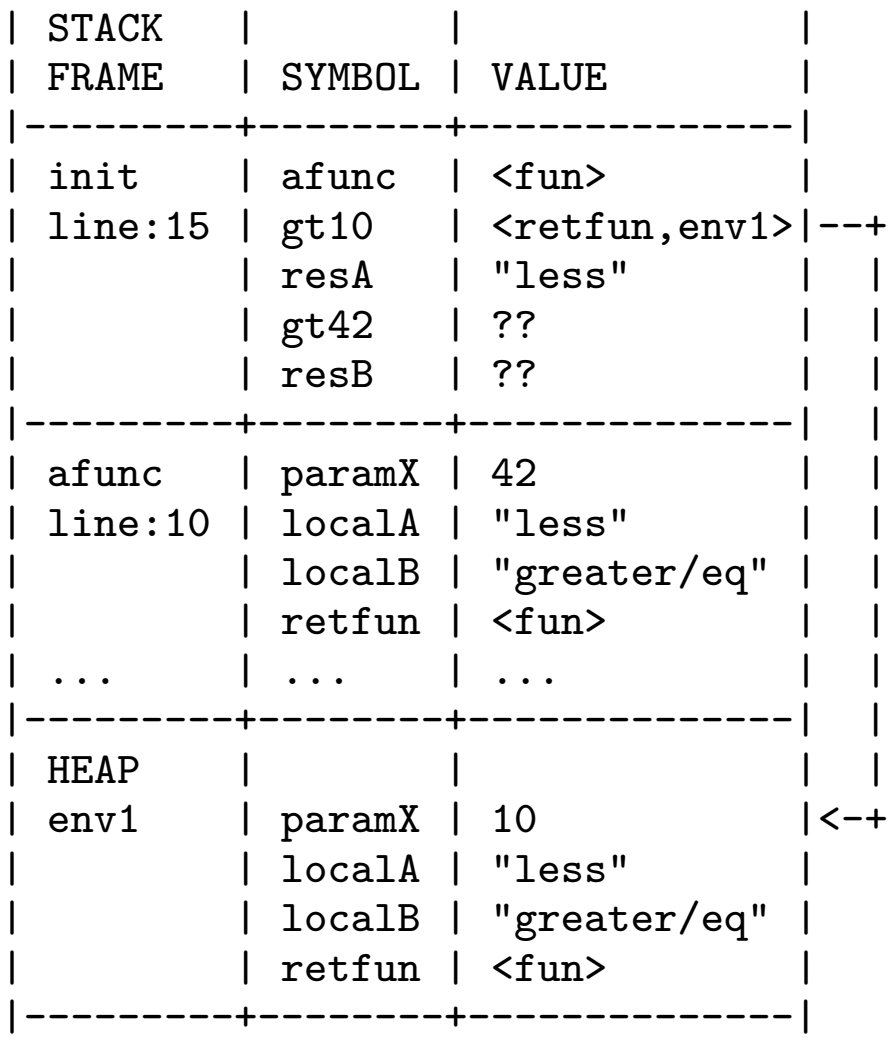
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
>>15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```



Closures in Action 5

afunc again has bindings that escape through retfun so requires a closure to be allocated.

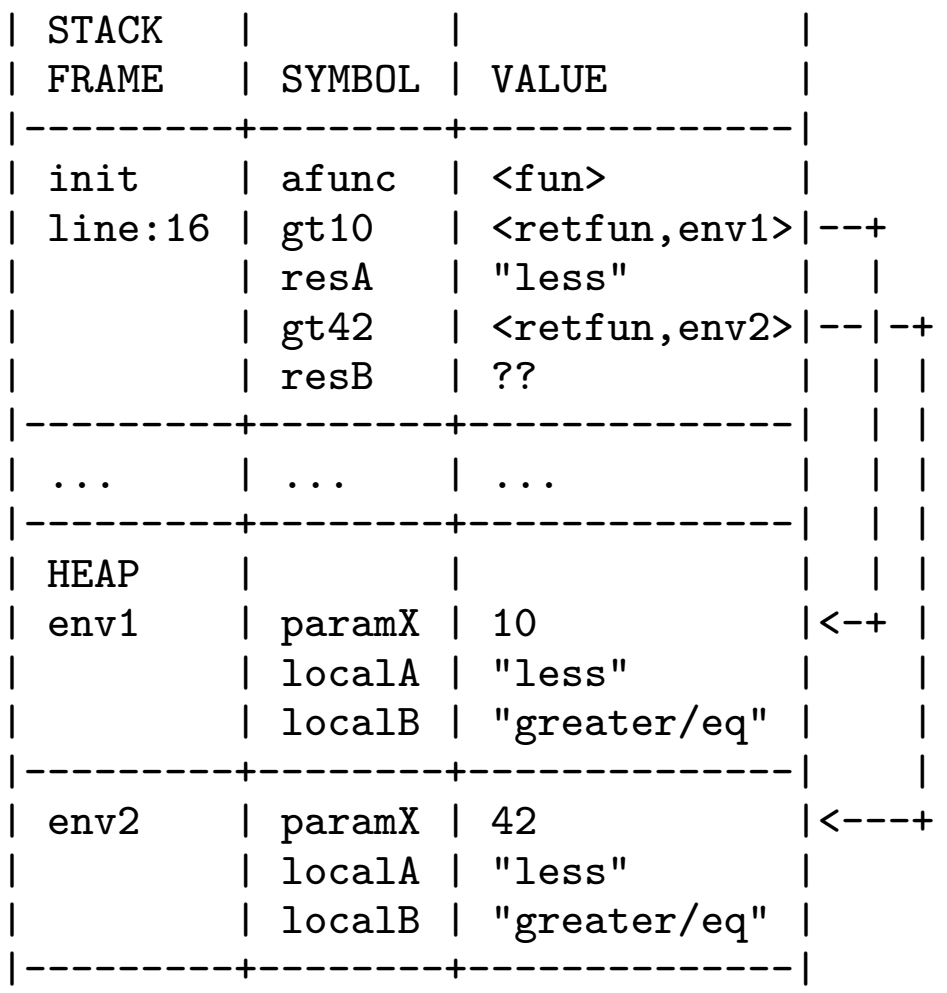
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
>>10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```



Closures in Action 6

gt42 is bound to an closure with an environment where paramX is 42. env2 is distinct from env1.

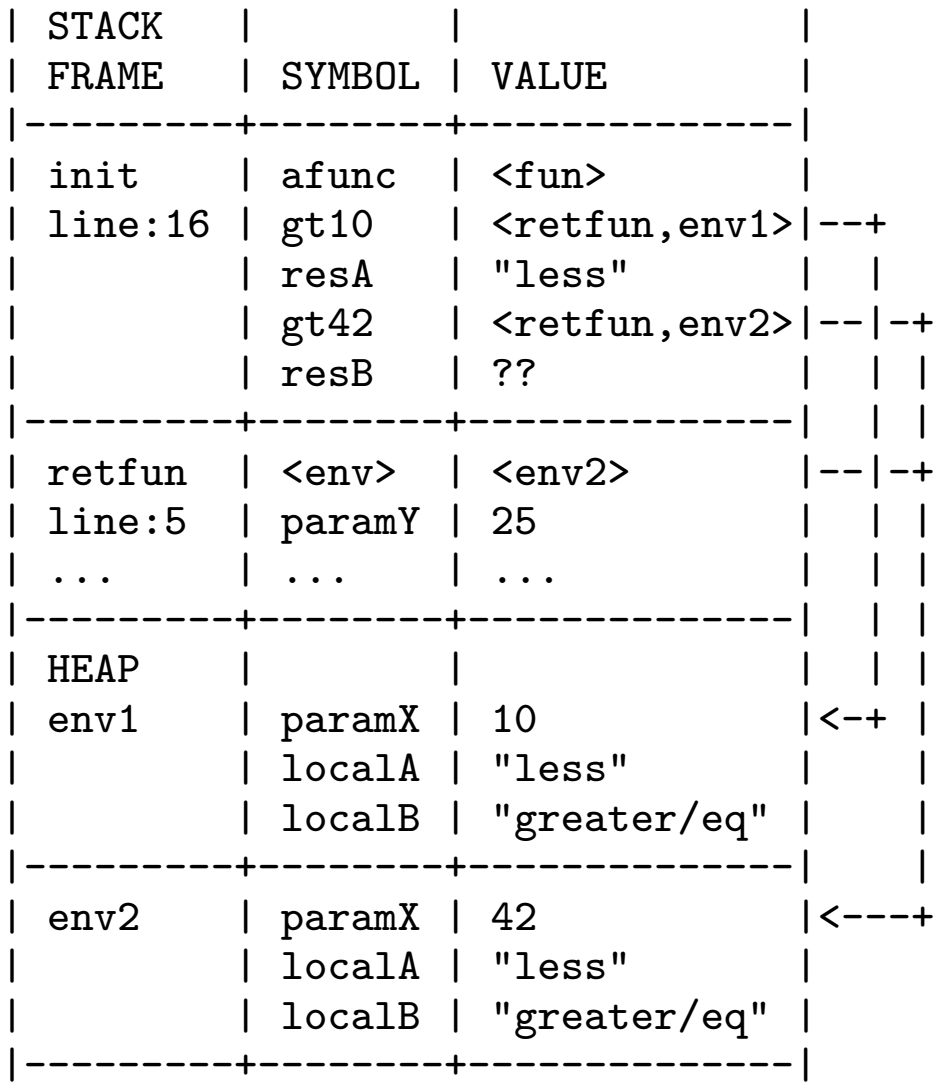
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
>>16 let resB = gt42 25;;
  17
```



Closures in Action 7

Applying `gt42 25` executes `retfun` with `env2` where `paramX=42`. Will follow the `else` branch to line 8.

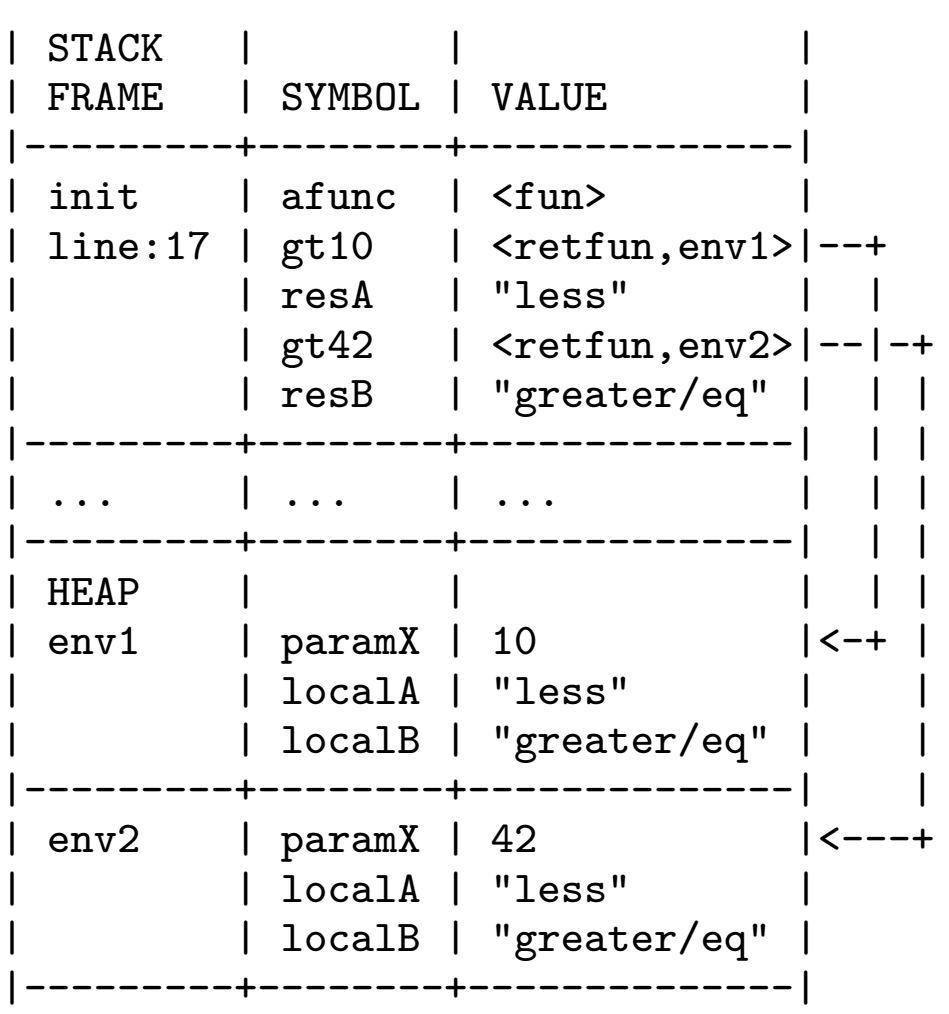
```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
>> 5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
  17
```



Closures in Action 8

Call to `retfun` completes popping off the stack, binding return value "greater/eq" to `resB`.

```
   1 let afunc paramX =
   2   let localA = "less" in
   3   let localB = "greater/eq" in
   4   let retfun paramY =
   5     if paramX < paramY then
   6       localA
   7     else
   8       localB
   9   in
  10   retfun
  11 ;;
  12
  13 let gt10 = afunc 10;;
  14 let resA = gt10 18;;
  15 let gt42 = afunc 42;;
  16 let resB = gt42 25;;
>>17
```

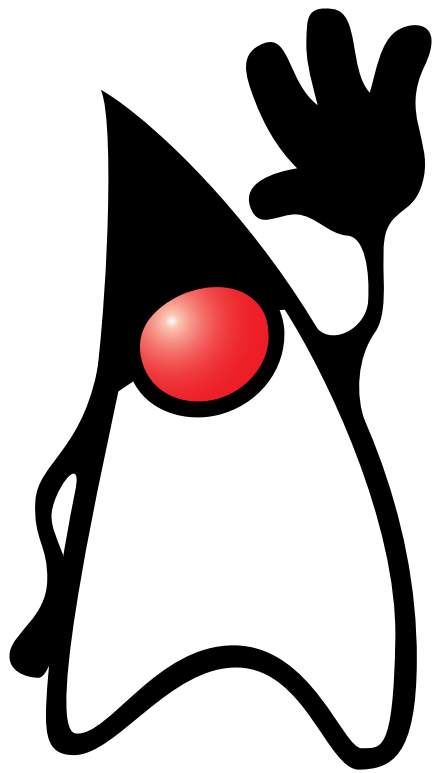


Notes on Closure Demo

- The preceding is a **model** and wrong about some details, but gives a general idea of how closures work
- Modern CPU/Memory is fast at the Stack Model
- Heap allocation is generally slower that Function Stack push/pops as it creates **GC pressure** (garbage collector pressure)
- To drive speed, OCaml's compiler generates stack-based code as much as possible; e.g. whenever module-level functions are **fully applied**
- OCaml compiler may generate several low-level versions of a function for full application, partial application, etc.

Code + Data: This seems familiar...

- Notice that a closure is some code and some data that is *private* along with associated functions on it
- What was that other common programming paradigm that couples data and code...



Closures vs Objects

- Examine the code in `closure_v_object.ml`
- Note how closure create private data accessible only through function calls
- This is pattern is commonly associated with **Object-oriented programming** as well: private data + associated methods
- **Closures and Objects provide equivalent power**
- Languages have varying support for them
  - None: C
  - Closures Only: Standard ML, vanilla Scheme
  - Objects Only: Java
  - Both: OCaml, Clojure, Javascript
- Why both? While equivalent, it is often more convenient to use one or the other
  - Closures to carry out an action which needs context
  - Objects to carry data with controlled operations

A Closure Koan[3]

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing - is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man's closures."

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's objects."

At that moment, Anton became enlightened.

– *Discussion on Scheme between Guy Steel and Anton van Straaten*

[3] *Koan*: a story, dialogue, question, or statement which is used in Zen practice to provoke the "great doubt" and test a student's progress in Zen practice. See Rootless Root for excellent Unix Koans.

Where do first-class functions get used?

Aside from higher-order function patterns, two other use cases are worth explicit mention

- Callback Functions
- Customization Hooks

Callback Functions

- A **callback function** or just **callback** is some action to perform after time has elapsed or a specific event has occurred
- Very common in **GUI Programming** to connect a GUI event like clicking to an action in a callback[4]
- See `gtkcounter.ml` for a simple example with OCaml's LablGtk GUI Library (or Introduction to GTK tutorial)

```
 1    let button =                        (* create a button in the window *)
 2      GButton.button ~label:"Not clicked yet" ~packing:window#add ()
 3    in
 4    let click_count = ref 0 in          (* ref to count clicks *)
 5    let click_callback () =             (* what to do when button is clicked *)
 6      printf "click_callback running\n"; flush_all ();
 7      click_count := !click_count + 1;
 8      let msg = sprintf "Cicked %d times" !click_count in
 9      button#set_label msg
10    in
11    let _ =                             (* connect clicking to the callback *)
12      button#connect#clicked ~callback:click_callback
13    in
```

[4] Java/C++ favor inheritance and objects for callbacks. While equivalent in expressiveness, it is usually much lengthier to write. Contrast `gtkcounter.ml` to a C++ GTK Hello World or Java Swing Hello World

Customization Hooks

- A **customization hook** or just **hook** is code that will run at startup or finish of some event
- C programs can specify exit hooks with the `atexit` library call: it takes a function pointer as a parameter.
- Emacs Hooks apply customizations when different file types are opened; hook code is specified in Emacs Lisp where `lambda` creates an anonymous function

```
;; Part of the .emacs startup code. Hook functions are run when a file of
;; the type mentioned is opened
(add-hook 'c-mode-common-hook                ; c editing customizations
 (lambda ()                                  ; anonymous function which ..
   (setq comment-start "// ")                ; changes comment symbols
   (setq comment-end "")
   (c-set-offset 'cpp-macro 0 nil)           ; adjusts indentation
   (setq c-basic-offset 2)))
(add-hook 'asm-mode-hook                     ; assembly editing customs
 (lambda ()                                  ; anonymous function which
   (setq comment-start "# ")                 ; sets comment syntax to AT&T
   (setq comment-end "")))
(add-hook 'tuareg-mode-hook                  ; ocaml editing customization
 '(lambda ()                                 ; anonymous function which...
    (local-set-key "\M-;" 'comment-dwim)     ; changes key bindings
    (local-set-key "\M-q" 'tuareg-fill-comment)))
```