Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Functions

# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16

First definition syntactic sugar for second
Using a nameless function

# (fun x -> x * 3) 5;;  (* An application *)
- : int = 15
# ((fun y -> y +. 2.0), (fun z -> z * 3));;
(* As data *)
- : (float -> float) * (int -> int) = (<fun>, <fun>)

Note: in fun v -> exp(v), scope of variable is only the body exp(v)
Values fixed at declaration time

```ocaml
# let x = 12;;
val x : int = 12

# let plus_x y = y + x;;
val plus_x : int -> int = <fun>

# plus_x 3;;
```

What is the result?
Values fixed at declaration time

# let x = 12;;
val x : int = 12

# let plus_x y = y + x;;
val plus_x : int -> int = <fun>

# plus_x 3;;
- : int = 15
Values fixed at declaration time

# let x = 7;;  (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;

What is the result this time?
Values fixed at declaration time

# let x = 7;; (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;

What is the result this time?
Values fixed at declaration time

```ocaml
# let x = 7;;  (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;
- : int = 15
```
Observation: Functions are first-class values in this language.

Question: What value does the environment record for a function variable?

Answer: a closure
A *closure* is a pair of an environment and an association of a formal parameter (the input variables)\(^*\) with an expression (the function body), written:

\[
\text{f} \rightarrow < (v_1, \ldots, v_n) \rightarrow \text{exp}, \rho_f >
\]

Where \(\rho_f\) is the environment in effect when \(f\) is defined (if \(f\) is a simple function)

\(^*\) Will come back to the “formal parameter”
Closure for plus_x

- When plus_x was defined, had environment:
  \[ \rho_{\text{plus}_x} = \{ \ldots, x \rightarrow 12, \ldots \} \]

- Recall: \texttt{let plus}_x \texttt{ y = y + x}
  
is really \texttt{let plus}_x = \texttt{fun y -> y + x}

- Closure for \texttt{fun y -> y + x}:
  \[ \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle \]

- Environment just after plus_x defined:
  \[ \{ \text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle \} + \rho_{\text{plus}_x} \]
Now it’s your turn

You should be able to complete ACT1
Functions with more than one argument

```ocaml
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let t = add_three 6 3 2;;
val t : int = 11
# let add_three =
  fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second
# let add_three x y z = x + y + z;;

val add_three : int -> int -> int -> int = <fun>

- What is the value of add_three?
- Let $\rho_{\text{add\_three}}$ be the environment before the declaration
- Remember:

  let add_three =
    fun x -> (fun y -> (fun z -> x + y + z));;

Value: $<x \rightarrow \text{fun } y \rightarrow (\text{fun } z \rightarrow x + y + z)$, $\rho_{\text{add\_three}}$ >
Partial application of functions

```ocaml
let add_three x y z = x + y + z;;

# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```
Partial application of functions

```
let add_three x y z = x + y + z;;
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

- Partial application also called `sectioning`
Functions as arguments

```ocaml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>

# let g = thrice plus_two;;
val g : int -> int = <fun>

# g 4;;
- : int = 10

# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```
Tuples as Values

// \( \rho_7 = \{ c \rightarrow 4, \text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5 \} \)

# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

// \( \rho_8 = \{ s \rightarrow (5, "hi", 3.2), \ c \rightarrow 4, \ \text{test} \rightarrow 3.7, \ a \rightarrow 1, \ b \rightarrow 5 \} \)
Pattern Matching with Tuples

\[
\rho_8 = \{ s \rightarrow (5, "hi", 3.2), \\
c \rightarrow 4, \ test \rightarrow 3.7, \\
a \rightarrow 1, \ b \rightarrow 5 \}\]

# let \( (a,b,c) = s; \) (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2

# let \( x = 2, \ 9.3; \) (* tuples don't require parens in Ocaml *)
val x : int * float = (2, 9.3)
Nested Tuples

(*Tuples can be nested *)

let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)

(*Patterns can be nested *)
let (p,(st,_),_) = d;; (* _ matches all, binds nothing *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
Functions on tuples

```ocaml
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```
# let triple_to_pair triple =

match triple with
  (0, x, y) -> (x, y)
| (x, 0, y) -> (x, y)
| (x, y, _) -> (x, y);

val triple_to_pair : int * int * int -> int * int = <fun>

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause
Closure for plus_pair

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before \texttt{plus\_pair} defined

- Closure for \texttt{plus\_pair}:

\[
<(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}}>
\]

- Environment just after \texttt{plus\_pair} defined:

\[
\{\text{plus\_pair} \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}}>\} + \rho_{\text{plus\_pair}}
\]
Save the Environment!

- A closure is a pair of an environment and an association of a pattern (e.g. \((v_1,\ldots,v_n)\) giving the input variables) with an expression (the function body), written:

\[
< (v_1,\ldots,v_n) \rightarrow \text{exp}, \rho >
\]

- Where \(\rho\) is the environment in effect when the function is defined (for a simple function)