# Implementing OCaml in OCaml Part 3: More Features, More Fun!

# Speaker: David Walker COS 326 Princeton University

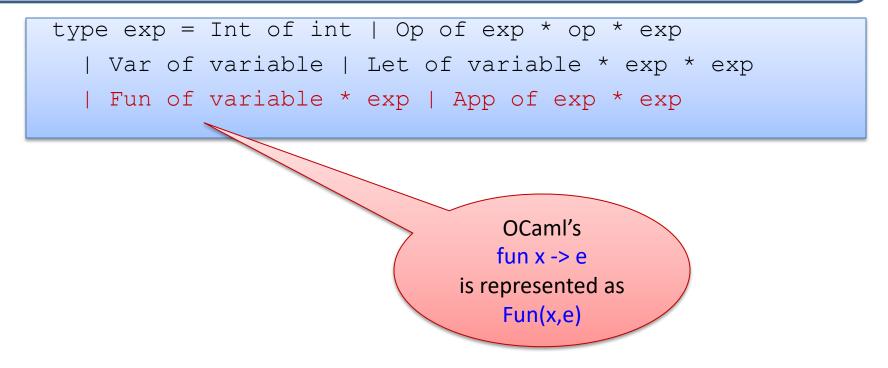


slides copyright 2020 David Walker and Andrew Appel permission granted to reuse these slides for non-commercial educational purposes

type exp = Int of int | Op of exp \* op \* exp

| Var of variable | Let of variable \* exp \* exp

| Fun of variable \* exp | App of exp \* exp



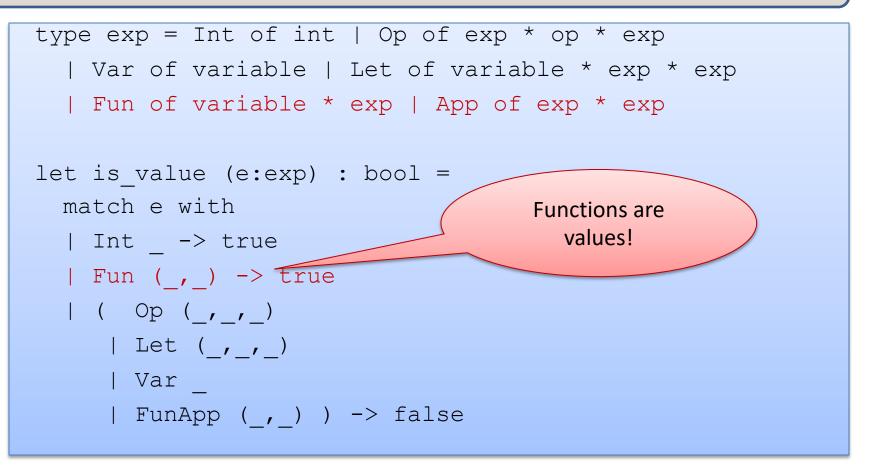


type exp = Int of int | Op of exp \* op \* exp

| Var of variable | Let of variable \* exp \* exp

| Fun of variable \* exp | App of exp \* exp

A function "application" (ie: function call) fact 3 is implemented as App (Var "fact", Int 3)



Easy Exam Question: What value does the OCaml interpreter produce when it evaluates the expression (fun  $x \rightarrow 3$ )?

Answer: the value produced is (fun x -> 3)

```
type exp = Int of int | Op of exp * op * exp
  | Var of variable | Let of variable * exp * exp
  | Fun of variable * exp | App of exp * exp
let is value (e:exp) : bool =
 match e with
  | Int -> true
  | Fun ( , ) -> true
  | ( Op (_,_,)
     | Let ( , , )
     | Var
     | App (_,_) ) -> false
                                        Function Apps are
                                           not values.
```

```
let rec eval (e:exp) : exp =
 match e with
  | Int i -> Int i
  | Op(e1,op,e2) -> eval op (eval e1) op (eval e2)
  | Let(x,e1,e2) \rightarrow eval (substitute (eval e1) x e2)
  | Var x -> raise (UnboundVariable x)
  | Fun (x,e) \rightarrow Fun (x,e)
  | App (e1,e2) ->
      (match eval e1, eval e2 with
       | Fun (x,e), v2 -> eval (substitute v2 x e)
       -> raise TypeError)
```



# Simplifying a little

```
let rec eval (e:exp) : exp =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) -> eval op (eval e1) op (eval e2)
  | Let(x,e1,e2) \rightarrow eval (substitute (eval e1) x e2)
  | Var x -> raise (UnboundVariable x)
  | Fun (x,e) \rightarrow Fun (x,e)
  | App (e1,e2) ->
       (match eval e1 with
        | Fun (x, e) \rightarrow eval (substitute (eval e2) x e)
        -> raise TypeError)
                   We don't really need
```

to pattern-match on e2. Just evaluate here

# Simplifying a little

```
let rec eval (e:exp) : exp =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) -> eval op (eval e1) op (eval e2)
  | Let(x,e1,e2) \rightarrow eval (substitute (eval e1) x e2)
  | Var x -> raise (UnboundVariable x)
  | Fun (x,e) \rightarrow Fun (x,e)
  | App (ef,e1) ->
       (match eval ef with
        | Fun (x, e^2) \rightarrow eval (substitute (eval e1) x e2)
        | -> raise TypeError)
                                          This looks like
                                         the case for let!
```

# Let and Lambda

In general:

let 
$$x = e1$$
 in  $e2 == (fun x -> e2) e1$ 



#### So we could write:

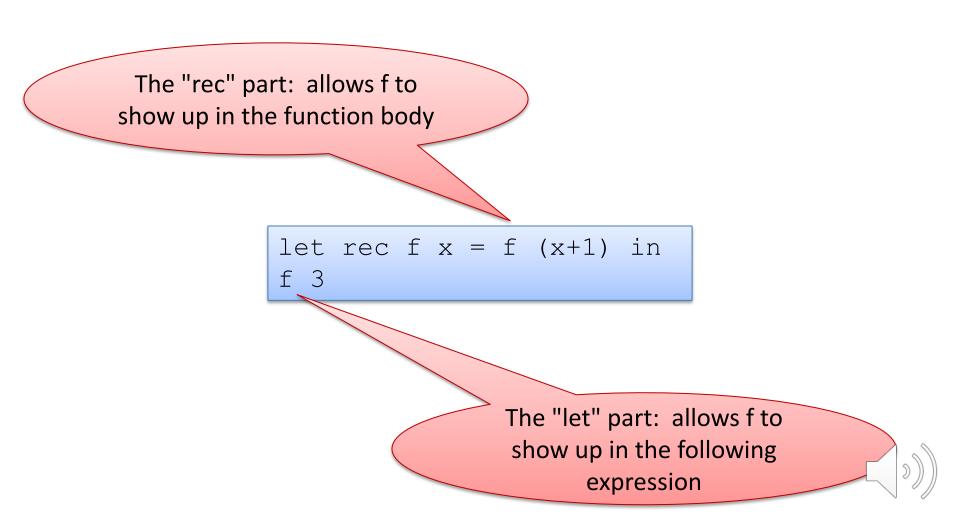
```
let rec eval (e:exp) : exp =
 match e with
  | Int i -> Int i
  | Op(e1,op,e2) -> eval op (eval e1) op (eval e2)
  | Let(x,e1,e2) -> eval (App (Fun (x,e2), e1))
  | Var x -> raise (UnboundVariable x)
  | Fun (x,e) \rightarrow Fun (x,e)
  | App (ef,e2) ->
      (match eval ef with
       | Fun (x, e1) \rightarrow eval (substitute (eval e1) x e2)
       -> raise TypeError)
```

In programming-languages speak: "Let is *syntactic sugar* for a function App"

Syntactic sugar: A new feature defined by a simple, local transformation.

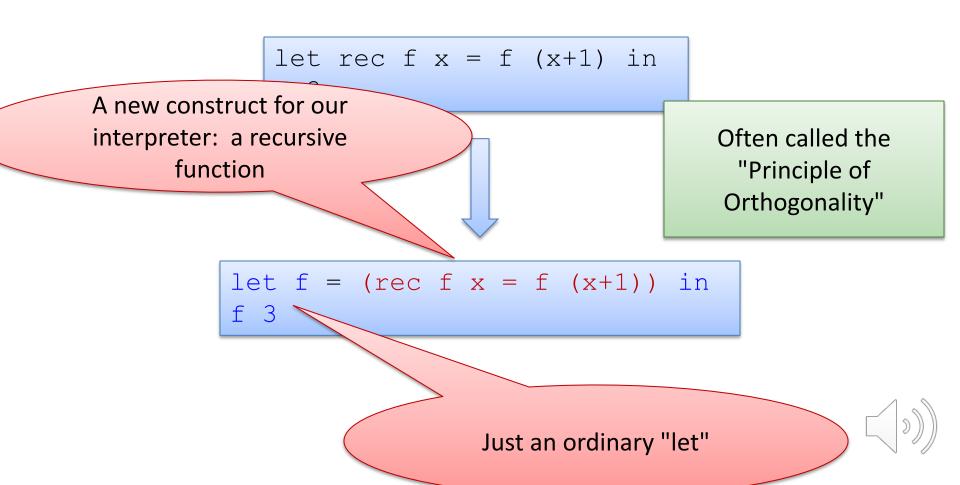
### **Recursive Function Definitions in OCaml**

A "let rec" definition does two independent things

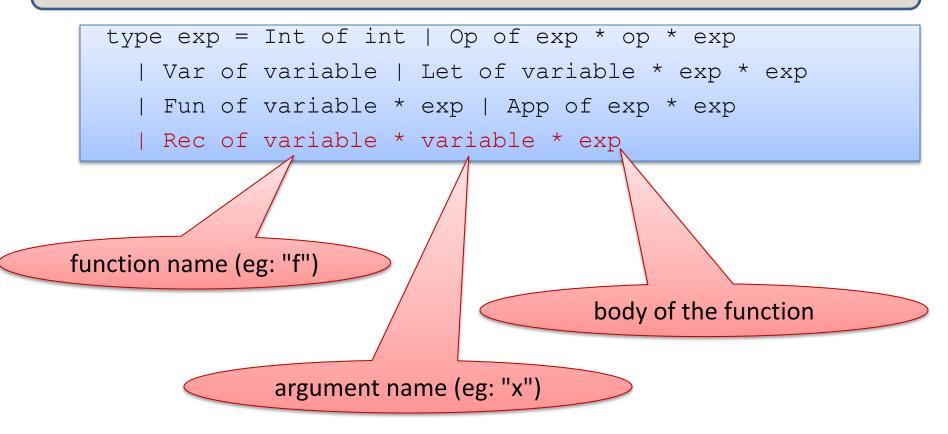


#### **Recursive Function Definitions in OCaml**

In our interpreter, we are going to split those things apart into two different constructs

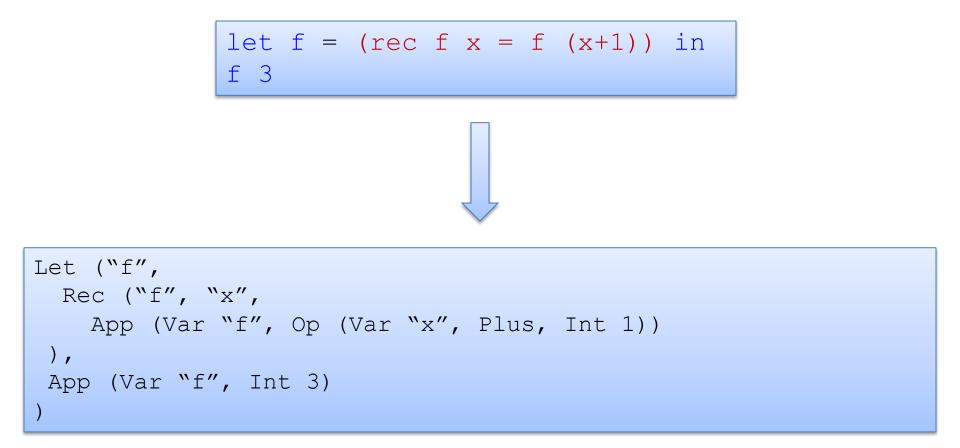


## **Recursive definitions**





#### **Recursive Function Definitions in OCaml**



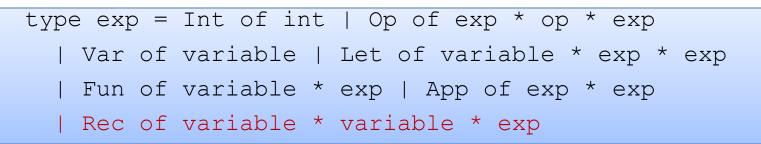


### **Recursive Function Definitions in OCaml**

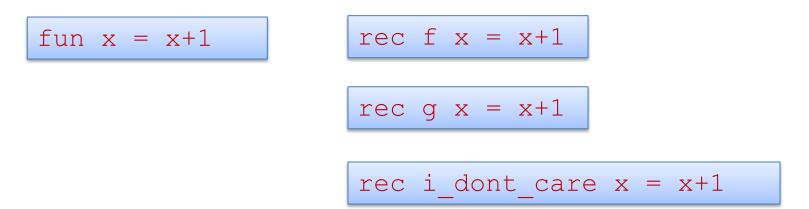
To avoid confusion, let's rename the variable used in the following expression (but not the function body).

Ap

#### **Recursive Values**



Notice that the following values are the same:



So now that we have the "Rec" form in our syntax, we could delete the "Fun" form as it is unnecessary and can be encoded:



#### **Recursive definitions**

```
type exp = Int of int | Op of exp * op * exp
| Var of variable | Let of variable * exp * exp
| Fun of variable * exp | App of exp * exp
| Rec of variable * variable * exp
```

```
let is_value (e:exp) : bool =
  match e with
    | Int _ -> true
    | Fun (_,_) -> true
    | Rec of (_,_,_) -> true
    | (Op (_,_,_) | Let (_,_,_) |
        Var _ | App (_,_) ) -> false
```

# Interlude: Notation for Substitution

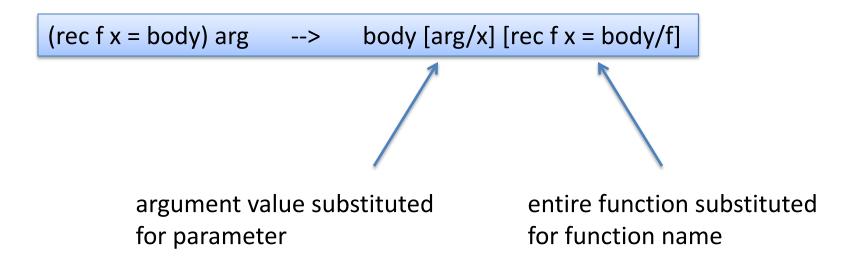
"Substitute value v for variable x in expression e:" e[v/x]

#### Examples of substitution:

(x + y) [7/y]	is	(x + 7)
(let x =30 in let y=40 in x + y) [7/y]	is	(let x =30 in let y=40 in x + y)
(let y = y in let y = y in y + y) [7/y]	is	(let y = 7 in let y = y in y + y)



Basic evaluation rule for recursive functions:





#### **Evaluating Recursive Functions**

Start out with a let bound to a recursive function:

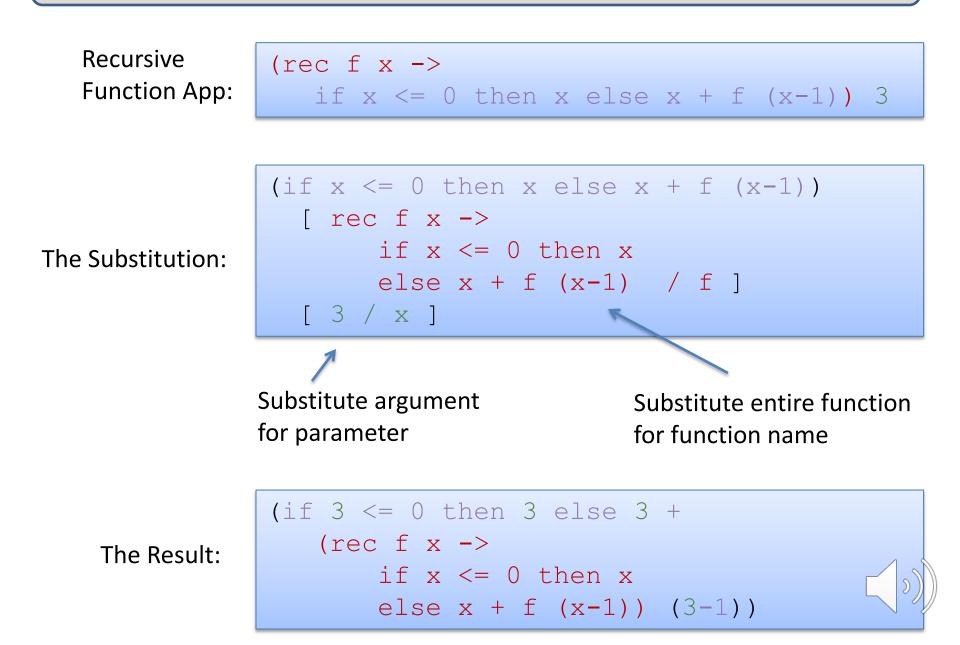
The Substitution:

g 3 [rec f x -> if x <= 0 then x else x + f (x-1) / g]

The Result:

(rec f x ->
 if x <= 0 then x else x + f (x-1)) 3</pre>

# **Evaluating Recursive Functions**



#### **Evaluating Recursive Functions**

```
let rec eval (e:exp) : exp =
  match e with
   Int i -> Int i
   Op(e1,op,e2) \rightarrow eval op (eval e1) op (eval e2)
   Let (x, e1, e2) \rightarrow eval (substitute (eval e1) x e2)
   Var x -> raise (UnboundVariable x)
   Fun (x,e) \rightarrow Fun (x,e)
   App (e1, e2) \rightarrow
       (match eval e1 with
                                        pattern as x
        | Fun (x,e) \rightarrow
            let v = eval e2 in
                                        match the pattern
            substitute e x v
                                        and binds x to value
          (\text{Rec }(f,x,e)) as f val ->
            let v = eval e2 in
            let body = substitute f val f
                            (substitute v x e) in
            eval body
        -> raise TypeError)
```

#### **More Evaluation**

```
(rec fact n = if n \le 1 then 1 else n * fact(n-1)) 3
-->
if 3 < 1 then 1 else
  3 * (rec fact n = if ... then ... else ...) (3-1)
-->
3 * (rec fact n = if ...) (3-1)
-->
3 * (rec fact n = if ...) 2
-->
3 * (if 2 \le 1 then 1 else 2 * (rec fact n = ...)(2-1))
-->
3 * (2 * (rec fact n = ...) (2-1))
-->
3 * (2 * (rec fact n = ...) (1))
-->
3 * 2 * (if 1 <= 1 then 1 else 1 * (rec fact ...) (1-1))
-->
3 * 2 * 1
```

# Exercise 1

(a) What is the result of the following substitution? In your answer, rename variables so you have as many unique variable names as possible.

$$(let g = rec f (x) = let g = fun x -> g (f x) in 0 in g (fun g -> g)) [(fun g -> g + 1)/g]$$

(b) What are the free variables of the following expression?

let  $g = rec f(x) = let g = fun x \rightarrow g(f x) in 0 in g(fun g \rightarrow g)$ 

(c) What are the free variables of your answer to (a)? More generally, how are the free variables of the expression e and the expression (e[v/x]) related?

# Exercise 2

26

Try extending the language and its evaluation system with:

- booleans (true, false, and, or, not, if)
- pairs (with pair creation and field extraction operations)