Implementing OCaml in OCaml
Part 2: Substitution and Evaluation

Speaker: David Walker

COS 326

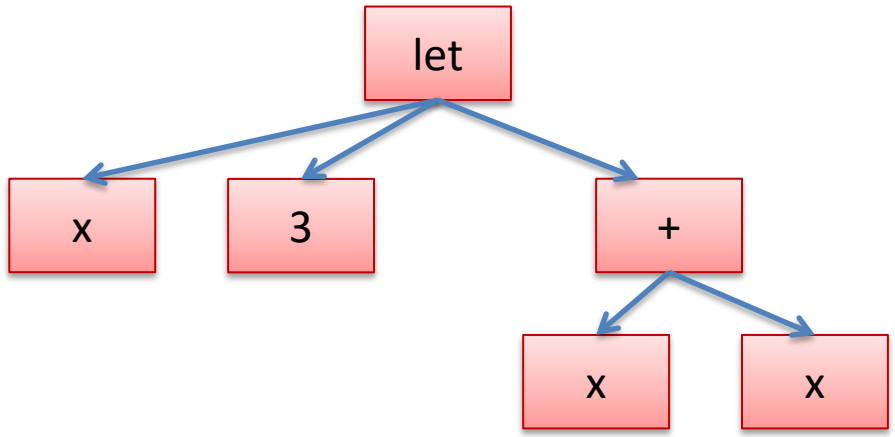
Princeton University

Last Time: Abstract Syntax Trees (ASTs)

foo.ml

```
let x = 3 in
x + x
```

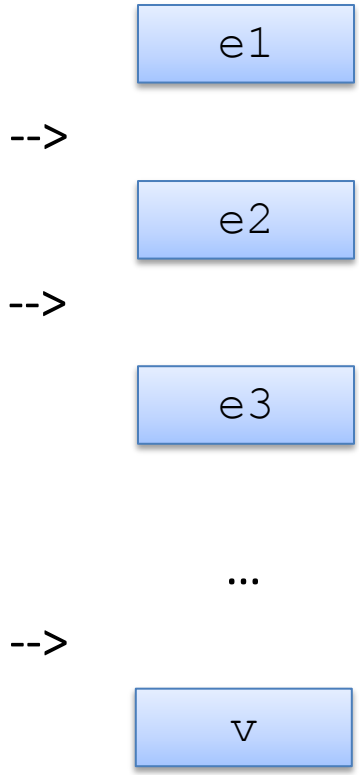
abstract syntax tree



OCaml representation of abstract syntax

```
type var = string
type op = Plus | Minus
type exp =
  | Int of int
  | Op  of exp * op * exp
  | Var of var
  | Let of var * exp * exp
```

Evaluation



Evaluation rewrites the abstract syntax of a program step-by-step until it produces a value.

Evaluation


```
let x = 30 in
let y = 20 + x in
x+y
```

-->

```
let y = 20 + 30 in
30+y
```

-->

```
let y = 50 in
30+y
```

-->

```
30+50
```

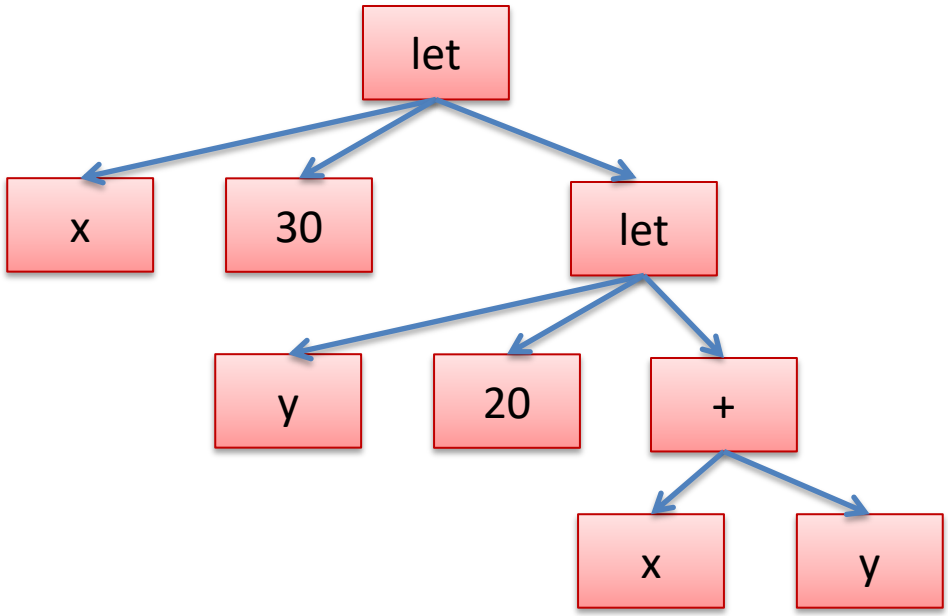
-->

```
80
```

evaluation complete: we have produced a *value*

Evaluation


```
let x = 30 in
let y = 20 in
x+y
```

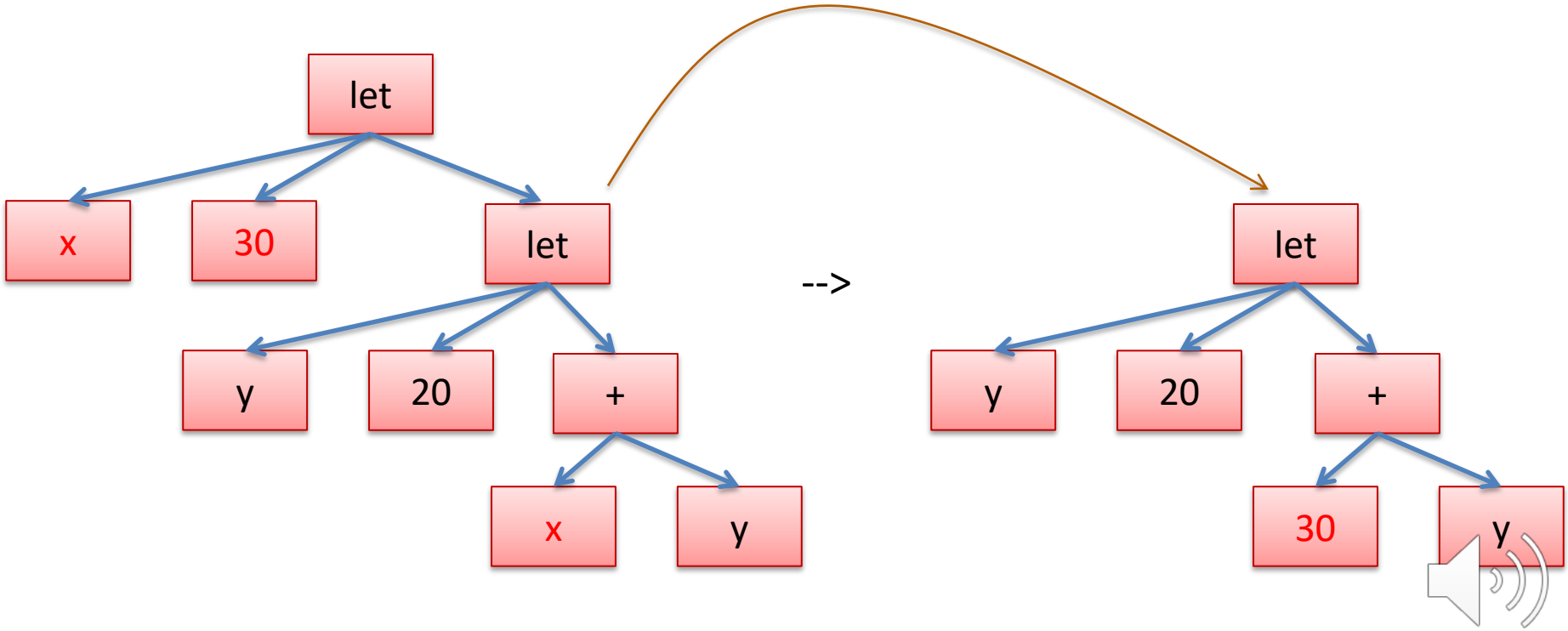


Evaluation via Substitution


```
let x = 30 in
let y = 20 in
x+y
```

-->

```
let y = 20 in
30+y
```



A Useful Auxiliary Function

nested "|" pattern
(can't use variables)

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

A *value* is a successful result of a computation.

Integers (3), strings ("hi"), functions ("fun x -> x + 2") are values.

Operations ("x + 2"), function calls ("f x"), match statements are not values.

Two Other Auxiliary Functions


```
(* eval_op v1 o v2:
   apply o to v1 and v2 *)
eval_op    : value -> op -> value -> value

(* substitute v x e:
   replace free occurrences of x with v in e *)
substitute : value -> variable -> exp -> exp
```

A Simple Evaluator


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value = ...


(* Goal:  evaluate e; return resulting value *)
```

A Simple Evaluator


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
   | Int i ->
   | Op(e1,op,e2) ->



   | Let(x,e1,e2) ->
```

A Simple Evaluator


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) ->



   | Let(x,e1,e2) ->
```

A Simple Evaluator


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) ->
          let v1 = eval e1 in
          let v2 = eval e2 in
          eval_op v1 op v2
   | Let(x,e1,e2) ->
```

A Simple Evaluator

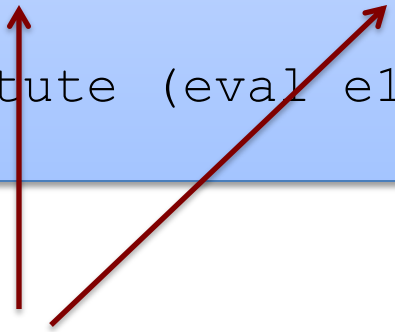

```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) ->
        let v1 = eval e1 in
        let v2 = eval e2 in
        eval_op v1 op v2
  | Let(x,e1,e2) ->
        let v1  = eval e1 in
        let e2' = substitute v1 x e2 in
        eval e2'
```

An Alternative


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) ->
         eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) ->
         eval (substitute (eval e1) x e2)
```

An Alternative


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) ->
        eval_op (eval e1) op (eval e2)
  | Let(x,e1,e2) ->
        eval (substitute (eval e1) x e2)
```



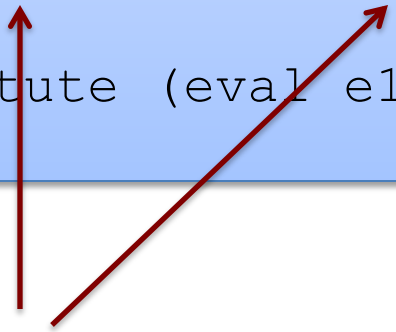
Which gets evaluated first?

Does OCaml use left-to-right eval order or right-to-left?

Always use OCaml let if you want to specify evaluation order.

An Alternative


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) ->
        eval_op (eval e1) op (eval e2)
  | Let(x,e1,e2) ->
        eval (substitute (eval e1) x e2)
```



Since the language we are interpreting is *pure* (no effects), it won't matter which expression gets evaluated first. We'll produce the same answer in either case.

An Alternative


```
is_value   : exp -> bool
eval_op    : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) ->
         eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) ->
         eval (substitute (eval e1) x e2)
```

Quick question:
Do you notice anything else suspicious here about this code?
Anything OCaml might flag?

Oops! We Missed a Case:


```
let eval_op v1 op v2 = …
let substitute v x e = …

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var x -> ???
```

What do we do here? Does our evaluator even run into this case?

Theorem 1: Well-typed programs have no free variables.

Theorem 2: If e has no free variables and e --> e' then e' has no free variables.

We don't need this case if we type check our program!
But it's bad style to leave out cases – you'd get a cloud of error messages.

We Could Use Options


```
let eval_op v1 op v2 = …
let substitute v x e = …

let rec eval (e:exp) : value option =
  match e with
   | Int i -> Some(Int i)
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var x -> None
```

But this isn't quite right – we need to match on the recursive calls to eval to make sure we get Some value!

Exceptions


```
exception UnboundVariable of variable

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var x -> raise (UnboundVariable x)
```

Instead, we can throw an exception.

Exceptions

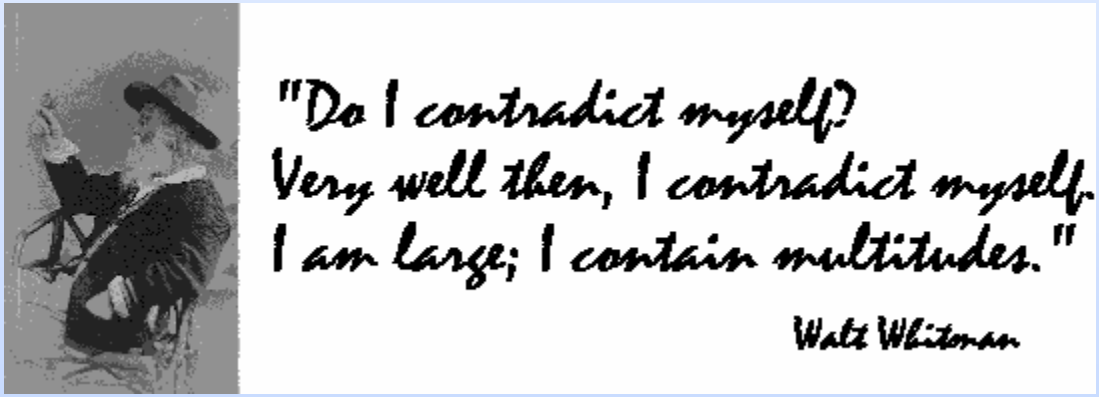

```
exception UnboundVariable of variable

let rec eval (e:exp) : value =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var x -> raise (UnboundVariable x)
```

Note that an exception declaration is a lot like a datatype declaration. Really, we are extending one big datatype (exn) with a new constructor (UnboundVariable).

Later on, we'll see how to catch an exception.

Exception or Option?

In a previous lecture, I railed against Java for all of the null pointer exceptions it raised. Should we use options or exns?



There are some rules; there is some taste involved.

- For errors/circumstances that *will occur*, use options
  - eg: the input might be ill formatted
- For errors that *cannot occur* (unless the program itself has a bug) and for which there are few "entry points" (few places checks needed) use exceptions
  - Java does not follow this rule: objects may be null *everywhere*

AUXILIARY FUNCTIONS

Evaluating the Primitive Operations


```
let eval_op (v1:value) (op:operand) (v2:value) : value =
  match v1, op, v2 with
  | Int i, Plus,  Int j -> Int (i+j)
  | Int i, Minus, Int j -> Int (i-j)
  | Int i, Times, Int j -> Int (i*j)
  | _ ,(Plus | Minus | Times), _ ->
    if is_value v1 && is_value v2 then
      raise TypeError
    else
      raise NotValue
```

Substitution

Want to replace x
(and only x) with v.

```
let substitute (v:exp) (x:variable) (e:exp) : exp =

    ...
```

Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
        match e with
        | Int _ ->
        | Op(e1,op,e2) ->
        | Var y ->              ... use x ...
        | Let (y,e1,e2) ->      ... use x ...



  in
  subst e
```

Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
        match e with
        | Int _ -> e
        | Op(e1,op,e2) ->
        | Var y ->
        | Let (y,e1,e2) ->



  in
  subst e
```

Substitution

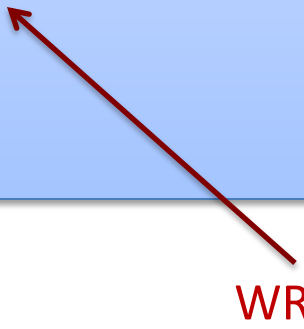
```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
        match e with
        | Int _ -> e
        | Op(e1,op,e2) -> Op(subst e1,op,subst e2)
        | Var y ->
        | Let (y,e1,e2) ->



  in
  subst e
```

Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
         match e with
         | Int _ -> e
         | Op(e1,op,e2) -> Op(subst e1,op,subst e2)
         | Var y -> if x = y then v else e
         | Let (y,e1,e2) ->




  in
  subst e
```

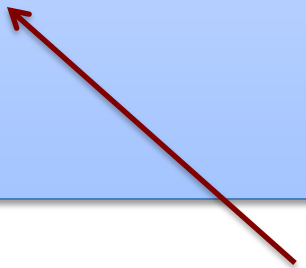
Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
        match e with
        | Int _ -> e
        | Op(e1,op,e2) -> Op(subst e1,op,subst e2)
        | Var y -> if x = y then v else e
        | Let (y,e1,e2) ->
            Let (y,
                 subst e1,
                 subst e2)
  in
  subst e
```



Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
        match e with
        | Int _ -> e
        | Op(e1,op,e2) -> Op(subst e1,op,subst e2)
        | Var y -> if x = y then v else e
        | Let (y,e1,e2) ->
            Let (y,
                 if x = y then e1 else subst e1,
                 if x = y then e2 else subst e2)
  in
  subst e
```



wrong

Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
        match e with
        | Int _ -> e
        | Op(e1,op,e2) -> Op(subst e1,op,subst e2)
        | Var y -> if x = y then v else e
        | Let (y,e1,e2) ->
            Let (y,
                 subst e1,
                 if x = y then e2 else subst e2)
  in
  subst e
```

If x and y are the same variable, then y *shadows* x.

evaluation, must implement our variable *scoping* rules correctly

Exercise

Here's the syntax of our little language:

```
type var = string
type op = Plus | Minus
type exp =
  | Int of int
  | Op  of exp * op * exp
  | Var of var
  | Let of var * exp * exp
```

Extending the abstract syntax of expressions. Extend the evaluation and substitution functions.

- (Easy) Booleans true and false, if statements, and operations like and, or, not
- (Harder) Pairs and patterns "let (x,y) = e1 in e2"