

Implementing an Interpreter

text file containing program
as a sequence of characters

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

Parsing

data structure representing program

```
Let ("x",  
    Num 3,  
    Binop(Plus, Var "x", Var "x"))
```

data structure representing
result of evaluation

```
Num 6
```

Evaluation

Pretty
Printing

```
6
```

text file/stdout
containing with formatted output

the **data type**
and **evaluator**
tell us a lot
about **program**
semantics

Making These Ideas Precise

We can define a datatype for simple OCaml expressions:

```
type variable = string

type op = Plus | Minus | Times | ...

type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of variable
  | Let_e of variable * exp * exp

type value = exp
```

Making These Ideas Precise

We can define a datatype for simple OCaml expressions:

```
type variable = string
type op = Plus | Minus | Times | ...
type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of variable
  | Let_e of variable * exp * exp
type value = exp

let e1 = Int_e 3
let e2 = Int_e 17
let e3 = Op_e (e1, Plus, e2)
```

← represents "3 + 17"

Making These Ideas Precise

We can represent the OCaml program:

```
let x = 30 in
let y =
  (let z = 3 in
   z*4)
in
y+y;;
```

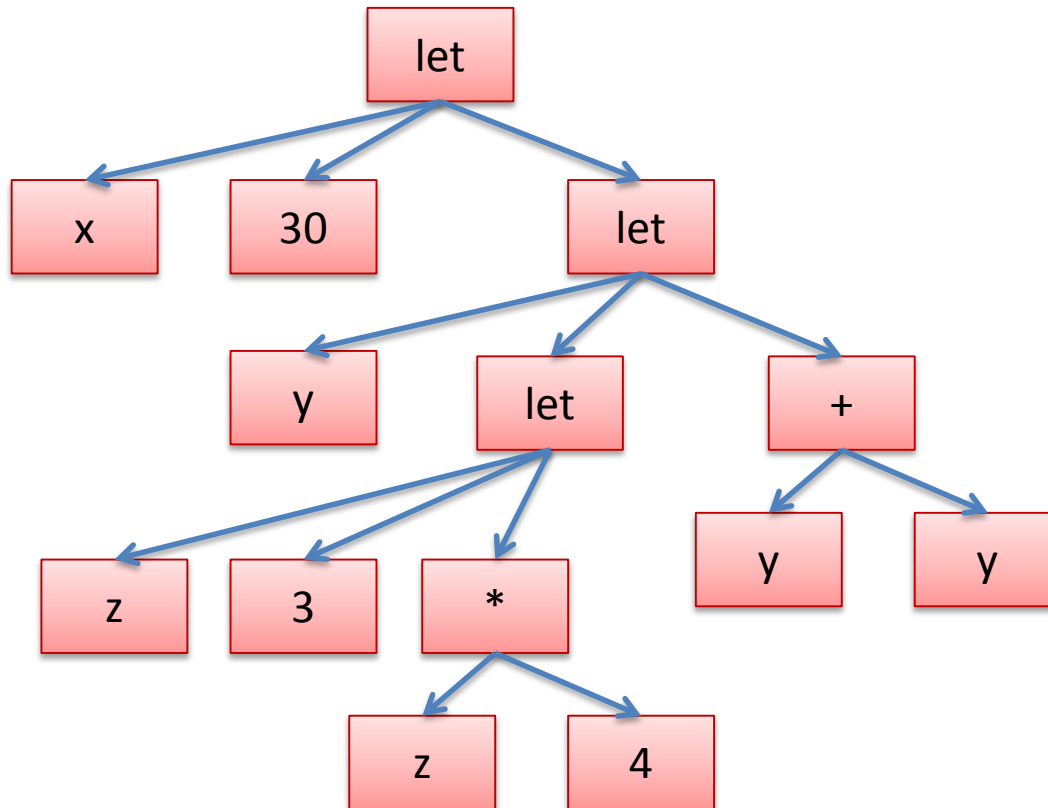
as an exp value:

```
Let_e("x", Int_e 30,
      Let_e("y",
            Let_e("z", Int_e 3,
                  Op_e(Var_e "z", Times, Int_e 4)),
            Op_e(Var_e "y", Plus, Var_e "y"))
```

Making These Ideas Precise

Notice how this reflects the “tree”:

```
Let_e("x", Int_e 30,  
      Let_e("y", Let_e("z", Int_e 3,  
                      Op_e(Var_e "z", Times, Int_e 4)),  
      Op_e(Var_e "y", Plus, Var_e "y"))
```



Binding occurrences versus applied occurrences

```
type exp =  
  | Int_e of int  
  | Op_e of exp * op * exp  
  | Var_e of variable  
  | Let_e of variable * exp * exp
```

This is an **applied** occurrence of a variable

Binding occurrences versus applied occurrences

```
type exp =  
  | Int_e of int  
  | Op_e of exp * op * exp  
  | Var_e of variable  
  | Let_e of variable * exp * exp
```

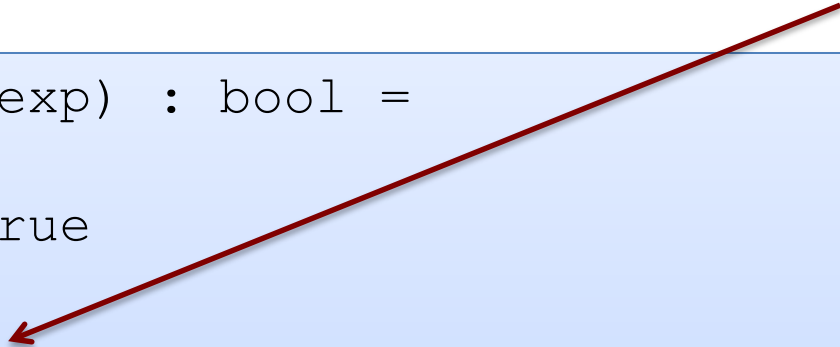
This is an **applied** occurrence of a variable

This is a **binding** occurrence of a variable

Some Useful Auxiliary Functions

nested “|” pattern
(can't use variables)

```
let is_value (e:exp) : bool =  
  match e with  
  | Int_e _ -> true  
  | ( Op_e _  
    | Let_e _  
    | Var_e _ ) -> false
```



```
eval_op      : value -> op -> value -> exp
```

```
(* 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) : exp =  
  match e with  
  | Int_e i ->  
  | Op_e(e1,op,e2) ->  
  | Let_e(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) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e (e1, op, e2) ->
  | Let_e (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) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e (e1,op,e2) ->
      let v1 = eval e1 in
      let v2 = eval e2 in
      eval_op v1 op v2
  | Let_e (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) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) ->
      let v1 = eval e1 in
      let v2 = eval e2 in
      eval_op v1 op v2
  | Let_e(x,e1,e2) ->
      let v1 = eval e1 in
      let e2' = substitute v1 x e2 in
      eval e2'
```

Simpler but Dangerous

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

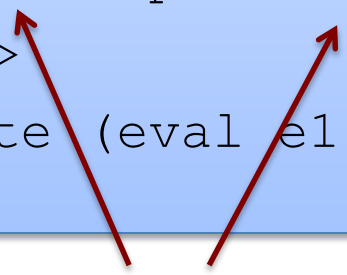
```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) ->  
    eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) ->  
    eval (substitute (eval e1) x e2)
```

Why?

Simpler but Dangerous

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

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e (e1, op, e2) ->
    eval_op (eval e1) op (eval e2)
  | Let_e (x, e1, e2) ->
    eval (substitute (eval e1) x e2)
```



Which gets evaluated first?

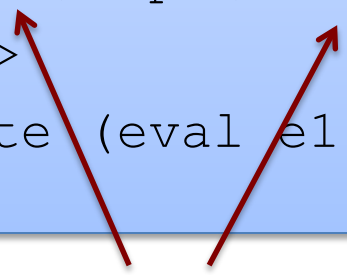
Does OCaml use left-to-right eval order or right-to-left?

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

Simpler but Dangerous

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

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e (e1, op, e2) ->
    eval_op (eval e1) op (eval e2)
  | Let_e (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.

Limitations of metacircular interpreters

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

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e (e1, op, e2) ->
      let v1 = eval e1 in
      let v2 = eval e2 in
      eval_op v1 op v2
  | Let_e (x, e1, e2) ->
      let v1 = eval e1 in
      let e2' = substitute v1 e2 in
      eval e2'
```

Which gets evaluated first,
(eval e1) or (eval e2)?
Seems obvious, right?
But that's because we
assume OCaml has call-by-value
evaluation! If it were
call-by-name, then this
ordering of lets would
not guarantee order
of evaluation.

*Moral: using a language to define its
own semantics has limitations.*

Back to the eval function...

```
let eval_op v1 op v2 = ...  
let substitute v x e = ...  
  
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
```

(same as the one a couple of slides ago)

Oops! We Missed a Case:

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

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> ???
```

We should never encounter a variable – they should have been substituted with a value! (This is a type-error.)

We could leave out the case for variables if we ***type check before evaluating.***
(*which we should definitely do!*)

But that will create a mess of Ocaml warnings – bad style. (Bad for debugging.)

We Could Use Options:

```
let eval_op v1 op v2 = ...  
let substitute v x e = ...  
  
let rec eval (e:exp) : exp option =  
  match e with  
    | Int_e i -> Some (Int_e i)  
    | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
    | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
    | Var_e 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) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)
```

Instead, we can throw an exception.

Exceptions

```
exception UnboundVariable of variable
```

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e 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.

AUXILIARY FUNCTIONS

Evaluating the Primitive Operations

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

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x) ;;
```

Substitution

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

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

```
...
```

```
;;
```


Substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =  
  let rec subst (e:exp) : exp =  
    match e with  
      | Int_e _ ->  
      | Op_e (e1,op,e2) ->  
      | Var_e y ->          ... use x ...  
      | Let_e (y,e1,e2) ->  ... use x ...  
  
  in  
  subst e  
  
;;
```

Substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =  
  let rec subst (e:exp) : exp =  
    match e with  
      | Int_e _ -> e  
      | Op_e (e1,op,e2) ->  
      | Var_e y ->  
      | Let_e (y,e1,e2) ->  
  
  in  
  subst e  
  
;;
```

Substitution

```
let substitute (v:value) (x:variable) (e:exp) : exp =  
  let rec subst (e:exp) : exp =  
    match e with  
      | Int_e _ -> e  
      | Op_e (e1,op,e2) -> Op_e (subst e1,op,subst e2)  
      | Var_e y ->  
      | Let_e (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 _ -> e  
      | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)  
      | Var_e y -> if x = y then v else e  
      | Let_e (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 _ -> e  
    | Op_e (e1,op,e2) -> Op_e(subst e1,op,subst e2)  
    | Var_e y -> if x = y then v else e  
    | Let_e (y,e1,e2) ->  
      Let_e (y,  
            subst e1,  
            subst e2)  
  
  in  
  subst e  
  
;;
```

wrong



Substitution

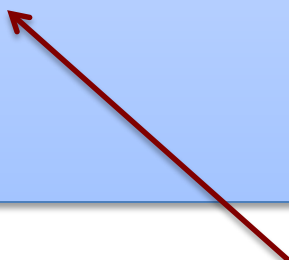
```
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
    | Int_e _ -> e
    | Op_e (e1,op,e2) -> Op_e(subst e1,op,subst e2)
    | Var_e y -> if x = y then v else e
    | Let_e (y,e1,e2) ->
      Let_e (y,
        if x = y then e1 else subst e1,
        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 _ -> e  
      | Op_e (e1,op,e2) -> Op_e(subst e1,op,subst e2)  
      | Var_e y -> if x = y then v else e  
      | Let_e (y,e1,e2) ->  
          Let_e (y,  
                subst e1,  
                if x = y then e2 else 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 _ -> e  
      | Op_e (e1,op,e2) -> Op_e(subst e1,op,subst e2)  
      | Var_e y -> if x = y then v else e  
      | Let_e (y,e1,e2) ->  
          Let_e (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.

Substitution

```
let substitute (v:exp) (x:variable) (e:exp) : exp =  
  let rec subst (e:exp) : exp =  
    match e with  
    | Int_e _ -> e  
    | Op_e (e1,op,e2) -> Op_e (subst e1,op,subst e2)  
    | Var_e y -> if x = y then v else e  
    | Let_e (y,e1,e2) ->  
      Let_e (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.

SCALING UP THE LANGUAGE

(MORE FEATURES, MORE FUN)

Scaling up the Language

```
type exp = Int_e of int | Op_e of exp * op * exp  
  | Var_e of variable | Let_e of variable * exp * exp  
  | Fun_e of variable * exp | FunCall_e of exp * exp ;;
```

Scaling up the Language

```
type exp = Int_e of int | Op_e of exp * op * exp  
  | Var_e of variable | Let_e of variable * exp * exp  
  | Fun_e of variable * exp | FunCall_e of exp * exp ;;
```

OCaml's
`fun x -> e`
is represented as
`Fun_e(x,e)`

Scaling up the Language

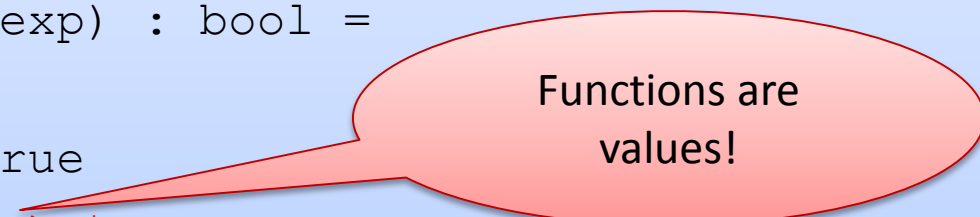
```
type exp = Int_e of int | Op_e of exp * op * exp  
  | Var_e of variable | Let_e of variable * exp * exp  
  | Fun_e of variable * exp | FunCall_e of exp * exp ;;
```

A function call
`fact 3`
is implemented as
`FunCall_e (Var_e "fact", Int_e 3)`

Scaling up the Language:

```
type exp = Int_e of int | Op_e of exp * op * exp
  | Var_e of variable | Let_e of variable * exp * exp
  | Fun_e of variable * exp | FunCall_e of exp * exp;;
```

```
let is_value (e:exp) : bool =
  match e with
  | Int_e _ -> true
  | Fun_e (_,_) -> true
  | ( Op_e (_,_,_)
    | Let_e (_,_,_)
    | Var_e _
    | FunCall_e (_,_) ) -> false ;;
```



Functions are values!

Easy exam question:

What value does the OCaml interpreter produce when you enter `(fun x -> 3)` in to the prompt?

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

Scaling up the Language:

```
type exp = Int_e of int | Op_e of exp * op * exp  
  | Var_e of variable | Let_e of variable * exp * exp  
  | Fun_e of variable * exp | FunCall_e of exp * exp;;
```

```
let is_value (e:exp) : bool =  
  match e with  
  | Int_e _ -> true  
  | Fun_e (_,_) -> true  
  | ( Op_e (_,_,_)  
    | Let_e (_,_,_)  
    | Var_e _  
    | FunCall_e (_,_) ) -> false ;;
```

Function calls are
not values.

Scaling up the Language:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1, eval e2 with  
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)  
     | _ -> raise TypeError)
```


Scaling up the Language:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1, eval e2 with  
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)  
     | _ -> raise TypeError)
```

values (including functions) always evaluate to themselves.

Scaling up the Language:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1, eval e2 with  
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)  
     | _ -> raise TypeError)
```

To evaluate a function call, we first evaluate both e1 and e2 to values.

Scaling up the Language

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1, eval e2 with  
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)  
     | _ -> raise TypeError)
```

e1 had better evaluate to a function value, else we have a type error.

Scaling up the Language

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1, eval e2 with  
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)  
     | _ -> raise TypeError)
```

Then we substitute e2's value (v2) for x in e and evaluate the resulting expression.

Simplifying a little

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1  
     | Fun_e (x,e) -> 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_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (ef,e1) ->  
    (match eval ef with  
     | Fun_e (x,e2) -> eval (substitute (eval e1) x e2)  
     | _ -> raise TypeError)
```

This looks like
the case for let!

Let and Lambda

```
let x = 1 in x+41
```

```
-->
```

```
1+41
```

```
-->
```

```
42
```

```
(fun x -> x+41) 1
```

```
-->
```

```
1+41
```

```
-->
```

```
42
```

In general:

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

So we could write:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (FunCall (Fun_e (x,e2), e1))  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (ef,e2) ->  
    (match eval ef with  
     | Fun_e (x,e1) -> eval (substitute (eval e1) x e2)  
     | _ -> raise TypeError)
```

In programming-languages speak: “Let is *syntactic sugar* for a function call”

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

Recursive definitions

```
type exp = Int_e of int | Op_e of exp * op * exp  
  | Var_e of variable | Let_e of variable * exp * exp |  
  | Fun_e of variable * exp | FunCall_e of exp * exp  
  | Rec_e of variable * variable * exp ;;
```

```
let rec f x = f (x+1) in f 3
```

(rewrite)



```
let f = rec f x -> f (x+1)) in  
f 3
```

(alpha-convert)



```
let g = rec f x -> f (x+1)) in  
g 3
```

(implement)



```
Let_e ("g",  
  Rec_e ("f", "x",  
    FunCall_e (Var_e "f", Op_e (Var_e "x", Plus, Int_e 1))  
  ),  
  FunCall (Var_e "g", Int_e 3)  
)
```

Recursive definitions

```
type exp = Int_e of int | Op_e of exp * op * exp  
  | Var_e of variable | Let_e of variable * exp * exp |  
  | Fun_e of variable * exp | FunCall_e of exp * exp  
  | Rec_e of variable * variable * exp ;;
```

```
let is_value (e:exp) : bool =  
  match e with  
  | Int_e _ -> true  
  | Fun_e (_,_) -> true  
  | Rec_e of (_,_,_) -> true  
  | (Op_e (_,_,_) | Let_e (_,_,_) |  
    Var_e _ | FunCall_e (_,_) ) -> 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 $(x + 7)$

$(\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y) [7/y]$ is $(\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y)$

$(\text{let } y = y \text{ in let } y = y \text{ in } y + y) [7/y]$ is $(\text{let } y = 7 \text{ in let } y = y \text{ in } y + y)$

Evaluating Recursive Functions

Basic evaluation rule for recursive functions:

$(\text{rec } f \ x = \text{body}) \ \text{arg} \ \rightarrow \ \text{body} \ [\text{arg}/x] \ [\text{rec } f \ x = \text{body}/f]$

argument value substituted
for parameter

entire function substituted
for function name

Evaluating Recursive Functions

Start out with
a let bound to
a recursive function:

```
let g =  
  rec f x ->  
    if x <= 0 then x  
    else x + f (x-1)  
in g 3
```

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
```


Evaluating Recursive Functions

Recursive
Function Call:

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

The Substitution:

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



Substitute argument
for parameter

Substitute entire function
for function name

The Result:

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

Evaluating Recursive Functions

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)  
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)  
  | Var_e x -> raise (UnboundVariable x)  
  | Fun_e (x,e) -> Fun_e (x,e)  
  | FunCall_e (e1,e2) ->  
    (match eval e1 with  
    | Fun_e (x,e) ->  
      let v = eval e2 in  
      substitute e x v  
  
    | (Rec_e (f,x,e)) as g ->  
      let v = eval e2 in  
      substitute (substitute e x v) f g  
  
    | _ -> raise TypeError)
```

pattern as x

match the pattern
and binds x to value

More Evaluation

```
(rec fact n = if n <= 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 <= 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
```


From Code to Abstract Specification

OCaml code can give a language semantics

- **advantage**: it can be executed, so we can try it out
- **advantage**: it is amazingly concise
 - especially compared to what you would have written in Java
- **disadvantage**: it is a little ugly to operate over concrete ML datatypes like “`Op_e(e1,Plus,e2)`” as opposed to “`e1 + e2`”
- **disadvantage**: defining a language in itself is a logical fallacy, as we saw earlier in this lecture

From Code to Abstract Specification

OCaml code can give a language semantics

- **advantage**: it can be executed, so we can try it out
- **advantage**: it is amazingly concise
- **disadvantage**: it is a little ugly to operate over concrete ML datatypes like “`Op_e(e1,Plus,e2)`” as opposed to “`e1 + e2`”
- **disadvantage**: defining a language in itself is a logical fallacy,

PL researchers have developed their own standard notation for writing down how programs execute

- it has a mathematical “feel” that makes PL researchers feel special and gives us *goosebumps* inside
- it operates over abstract expression syntax like “`e1 + e2`”
- it is useful to know this notation if you want to read specifications of programming language semantics
 - e.g.: Standard ML (of which OCaml is a descendent) has a formal definition given in this notation (and C, and Java; but not OCaml...)
 - e.g.: most papers in the conference POPL (ACM Principles of Prog. Lang.)

Goal

Our goal is to explain how an expression e evaluates to a value v .

In other words, we want to define a mathematical *relation* between pairs of expressions and values.

Formal Inference Rules

We define the “evaluates to” relation using a set of (inductive) rules that allow us to *prove* that a particular (expression, value) pair is part of the relation.

A rule looks like this:

$$\frac{\text{premise 1} \quad \text{premise 2} \quad \dots \quad \text{premise 3}}{\text{conclusion}}$$

You read a rule like this:

- “if *premise 1* can be proven and *premise 2* can be proven and ... and *premise n* can be proven then *conclusion* can be proven”

Some rules have no premises

- this means their conclusions are always true
- we call such rules “axioms” or “base cases”

An example rule

As a rule:

$$\frac{e1 \rightarrow v1 \quad e2 \rightarrow v2 \quad \text{eval_op}(v1, \text{op}, v2) == v'}{e1 \text{ op } e2 \rightarrow v'}$$

In English:

“If $e1$ evaluates to $v1$
and $e2$ evaluates to $v2$
and $\text{eval_op}(v1, \text{op}, v2)$ is equal to v'
then
 $e1 \text{ op } e2$ evaluates to v' ”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Op_e(e1, op, e2) -> let v1 = eval e1 in  
                        let v2 = eval e2 in  
                        let v' = eval_op v1 op v2 in  
                        v'
```

An example rule

As a rule:

$$\frac{i \in \mathbb{Z}}{i \rightarrow i}$$

asserts i is
an integer



In English:

“If the expression is an integer value, **it evaluates to itself.**”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  | Int_e i -> Int_e i  
  ...
```


An example rule concerning evaluation

As a rule:

$$\frac{}{\lambda x.e \twoheadrightarrow \lambda x.e}$$

typical “lambda” notation
for a function with
argument x, body e

In English:

“A function value evaluates to itself.”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  ...  
  | Fun_e (x,e) -> Fun_e (x,e)  
  ...
```


An example rule concerning evaluation

As a rule:

$$\frac{e1 \rightarrow \lambda x.e \quad e2 \rightarrow v2 \quad e[v2/x] \rightarrow v}{e1 \ e2 \rightarrow v}$$

In English:

“if $e1$ evaluates to a function with argument x and body e
and $e2$ evaluates to a value $v2$
and e with $v2$ substituted for x evaluates to v
then $e1$ applied to $e2$ evaluates to v ”

In code:

```
let rec eval (e:exp) : exp =
  match e with
  ..
| FunCall_e (e1,e2) ->
  (match eval e1 with
   | Fun_e (x,e) -> eval (substitute e x (eval e2))
   | ...)
  ..
```

An example rule concerning evaluation

As a rule:

$$\frac{e1 \rightarrow \text{rec } f \ x = e \quad e2 \rightarrow v \quad e[\text{rec } f \ x = e/f][v/x] \rightarrow v2}{e1 \ e2 \rightarrow v2}$$

In English:

“uggh”

In code:

```
let rec eval (e:exp) : exp =  
  match e with  
  ...  
  | (Rec_e (f,x,e)) as g ->  
    let v = eval e2 in  
    substitute (substitute e x v) f g
```

Comparison: Code vs. Rules

complete eval code:

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1
     | Fun_e (x,e) -> eval (Let_e (x,e2,e))
     | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
    (Rec_e (f,x,e)) as g ->
    let v = eval e2 in
    substitute (substitute e x v) f g
```

complete set of rules:

$$\frac{i \in \mathbb{Z}}{i \rightarrow i}$$
$$\frac{e1 \rightarrow v1 \quad e2 \rightarrow v2 \quad \text{eval_op}(v1, \text{op}, v2) == v}{e1 \text{ op } e2 \rightarrow v}$$
$$\frac{e1 \rightarrow v1 \quad e2 [v1/x] \rightarrow v2}{\text{let } x = e1 \text{ in } e2 \rightarrow v2}$$
$$\frac{}{\lambda x. e \rightarrow \lambda x. e}$$
$$\frac{e1 \rightarrow \lambda x. e \quad e2 \rightarrow v2 \quad e[v2/x] \rightarrow v}{e1 e2 \rightarrow v}$$
$$\frac{e1 \rightarrow \text{rec } f x = e \quad e2 \rightarrow v2 \quad e[\text{rec } f x = e/f][v2/x] \rightarrow v3}{e1 e2 \rightarrow v3}$$

Almost isomorphic:

- one rule per pattern-matching clause
- recursive call to eval whenever there is a \rightarrow premise in a rule
- what's the main difference?

Comparison: Code vs. Rules

complete eval code:

```

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1
     | Fun_e (x,e) -> eval (Let_e (x,e2,e))
     | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
    (Rec_e (f,x,e)) as g ->
      let v = eval e2 in
      substitute (substitute e x v) f g
  
```

complete set of rules:

$$\frac{i \in \mathbb{Z}}{i \rightarrow i}$$

$$\frac{e1 \rightarrow v1 \quad e2 \rightarrow v2 \quad \text{eval_op}(v1, \text{op}, v2) == v}{e1 \text{ op } e2 \rightarrow v}$$

$$\frac{e1 \rightarrow v1 \quad e2 [v1/x] \rightarrow v2}{\text{let } x = e1 \text{ in } e2 \rightarrow v2}$$

$$\frac{}{\lambda x. e \rightarrow \lambda x. e}$$

$$\frac{e1 \rightarrow \lambda x. e \quad e2 \rightarrow v2 \quad e[v2/x] \rightarrow v}{e1 e2 \rightarrow v}$$

$$\frac{e1 \rightarrow \text{rec } f \text{ x} = e \quad e2 \rightarrow v2 \quad e[\text{rec } f \text{ x} = e/f][v2/x] \rightarrow v3}{e1 e2 \rightarrow v3}$$

- There's no formal rule for handling free variables
- No rule for evaluating function calls when a non-function in the caller position
- In general, *no rule when further evaluation is impossible*
 - the rules express the *legal evaluations* and say nothing about what to do in error situations
 - the code handles the error situations by raising exceptions
 - type theorists prove that well-typed programs don't run into undefined cases

Some Final Words

- The substitution model is only a model.
 - it does not accurately model all of OCaml's features
 - I/O, exceptions, mutation, concurrency, ...
 - we can build models of these things, but they aren't as simple.
 - even substitution was tricky to formalize!
- It's useful for reasoning about higher-order functions, correctness of algorithms, and optimizations.
 - we can use it to formally prove that, for instance:
 - $\text{map } f (\text{map } g \text{ } xs) == \text{map } (\text{comp } f \text{ } g) \text{ } xs$
 - proof: by induction on the length of the list xs , using the definitions of the substitution model.
 - we often model complicated systems (e.g., protocols) using a small functional language and substitution-based evaluation.
- It is *not* useful for reasoning about execution time or space
 - more complex models needed there