Implementing OCaml in OCaml

COS 326
David Walker
Princeton University
Implementing an Interpreter

- Text file containing program as a sequence of characters:
  ```plaintext
  let x = 3 in
  x + x
  ```

- Data structure representing program:
  ```plaintext
  Let ("x", 
       Num 3, 
       Binop(Plus, Var "x", Var "x"))
  ```

- Data structure representing result of evaluation:
  ```plaintext
  Num 6
  ```

- Parsing:

- Evaluation:
  ```plaintext
  Num 6
  ```

- Pretty Printing:

- 6

- Text file/stdout containing formatted output:
  ```plaintext
  the data type and evaluator tell us a lot about program semantics
  ```
We can define a datatype for simple OCaml expressions:

```ocaml
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
```
We can define a datatype for simple OCaml expressions:

```ocaml
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
```
We can define a datatype for simple OCaml expressions:

```ocaml
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
```
We can define a datatype for simple OCaml expressions:

```ocaml
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”
We can represent the OCaml program:

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

This is called concrete syntax (concrete syntax pertains to parsing).

This is called an abstract syntax tree (AST).

as an exp value:

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

Notice how the OCaml expression can be drawn as a tree.
An occurrence of a variable where we are defining it via let is said to be a *binding occurrence* of the variable.

```plaintext
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
A non-binding occurrence of a variable is a *use* of a variable as opposed to a definition.

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Given a variable occurrence, we can find where it is bound by ...

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a

crawling up the tree to the nearest enclosing let...

Abstract Syntax Trees
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a

crawling up the tree to the nearest enclosing let...
Abstract Syntax Trees

crawling up the tree to the nearest enclosing let...

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
and checking if the “let” binds the variable – if so, we’ve found the nearest enclosing definition. If not, we keep going up.

let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
Now we can also systematically rename the variables so that it’s not so confusing. Systematic renaming is called \textit{alpha-conversion}.

\begin{verbatim}
let a = 30 in
let a = (let a = 3 in a*4) in
a+a
\end{verbatim}
Start with a let, and pick a fresh variable name, say “x”

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Rename the binding occurrence from “a” to “x”.

```plaintext
let x = 30 in
let a =
    (let a = 3 in a*4)
in
a+a
```
Then rename all of the occurrences of the variables that this let binds.

```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
There are none in this case!

```
let x = 30 in
let a =
    (let a = 3 in a*4)
in
a+a
```
There are none in this case!

\[
\text{let } x = 30 \text{ in } \\
\text{let } a = \\
\quad (\text{let } a = 3 \text{ in } a \times 4) \\
\text{in } \\
a + a
\]
Let’s do another let, renaming “a” to “y”.

```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Abstract Syntax Trees

Let’s do another let, renaming “a” to “y”.

```
let x = 30 in
let y =
  (let a = 3 in a*4)
in
y+y
```
And if we rename the other let to “z”:

```
let x = 30 in
let y =
  (let z = 3 in z*4)
in
y+y
```
And if we rename the other let to “z”:

```
let x = 30 in
let y =
  (let z = 3 in z*4)
in
y+y
```
Free vs Bound Variables

let x = 30 in x+y

```
let x = 30 in x+y
```
Free vs Bound Variables

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

```
let
x 30
+
  x y

this use of x is bound here
```
Free vs Bound Variables

let x = 30 in
x+y

this use of y is free

we say: "y is a free variable in this expression"
Other Examples

\[
\text{fun } z \rightarrow z + y
\]

\[z \text{ is bound}\]
\[y \text{ is a free variable}\]

\[
\text{match } x \text{ with}\]
\[
(y,z) \rightarrow y + z + w
\]

\[x, w \text{ are free variables}\]
\[y, z \text{ are bound}\]

\[
\text{let rec } f \ x =
\text{match } x \text{ with}\]
\[
[] \rightarrow y
| \text{hd:tl} \rightarrow \text{hd::f tl}
\]

\[y \text{ is a free variable}\]
\[f, x, \text{hd, tl are all bound}\]
recall, we write:

e1 \rightarrow e2

to indicate that e1 evaluates to e2 in a single step

for example:

2 + 3 \rightarrow 5
let x = 30 in
let y = 20 + x in
x+y
let x = 30 in  
let y = 20 + x in  
x+y

--> 

let y = 20 + 30 in  
30+y

Notice: we do a step of evaluation by substituting the value 30 for all the uses of x
In this step, we just evaluated the right-hand side of the let. We now have a *value* (50) on the right-hand side.
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

substitution again
**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*
let x = 30 in
let y = 20 in
x+y
let x = 30 in
let y = 20 in
x + y

--> let y = 20 in
    30 + y

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

-->
let y = 20 in
30 + y
```
Binding occurrences versus applied occurrences

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

This is a binding occurrence of a variable

This is a use of a variable
A Useful Auxiliary Function

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

Recall: A **value** is a successful result of a computation. Once we have computed a value, there is no more work to be done.

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

Operations ("x + 2"), function calls ("f x"), match statements are not value.
(* eval_op v1 o v2: 
apply o to v1 and v2 *)

\textbf{eval\_op} : \text{value} \rightarrow \text{op} \rightarrow \text{value} \rightarrow \text{exp}

(* substitute v x e: 
replace free occurrences of x with v in e *)

\textbf{substitute} : \text{value} \rightarrow \text{variable} \rightarrow \text{exp} \rightarrow \text{exp}
A Simple Evaluator

\[\text{is\_value} : \text{exp} \rightarrow \text{bool}\]
\[\text{eval\_op} : \text{value} \rightarrow \text{op} \rightarrow \text{value} \rightarrow \text{value}\]
\[\text{substitute} : \text{value} \rightarrow \text{variable} \rightarrow \text{exp} \rightarrow \text{exp}\]

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

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

```ocaml
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.
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 x 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 can have limitations.
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)
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)

Quick question:
Do you notice anything else suspicious here about this code?
Anything OCaml might flag?
Oops! We Missed a Case:

```ocaml
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 -> ???
```

If we start out with an expression with no *free variables*, we will never run into a free variable when we evaluate. Every variable gets replaced by a value as we compute, via substitution.

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

We could leave out the case for variables, but that will create a mess of Ocaml warnings – bad style. (Bad for debugging.)
We Could Use Options:

```ocaml
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!
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.
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.
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 (e.g., because 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 objects may be null everywhere
AUXILIARY FUNCTIONS
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

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

Want to replace \( x \) (and only \( x \)) with \( v \).
let substitute (v:exp) (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
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) ->
  | Var_e y ->
  | Let_e (y,e1,e2) ->
in
subst e
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 ->
      | Let_e (y,e1,e2) ->
    in
    subst e
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
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
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
Substitution

```ocaml
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
;;
```

evaluation/substitution must implement our variable scoping rules correctly
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)
```
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

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.
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)
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.
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.
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.
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.
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.
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:

```ocaml
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

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

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

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

let e = 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

```ml
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
```

```ml
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
```

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 _

Fun_e (x, body) == Rec_e("unused", x, body)

A better IR would just delete Fun_e – avoid unnecessary redundancy
```
“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 $(\text{let } y = 7 \text{ in let } y = y \text{ in } y + y)$
Basic evaluation rule for recursive functions:

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

- argument value substituted for parameter
- entire function substituted for function name
Start out with a let bound to a recursive function:

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

The Substitution:

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

The Result:

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

Recursive Function Call:

\[
\begin{align*}
&\text{(rec } f \ x \rightarrow \\
&\quad \text{if } x \leq 0 \text{ then } x \text{ else } x + f (x-1)\text{)} \ 3
\end{align*}
\]

The Substitution:

\[
\begin{align*}
&\text{(if } x \leq 0 \text{ then } x \text{ else } x + f (x-1)) \\
&\quad \text{[ rec } f \ x \rightarrow \\
&\quad \quad \text{if } x \leq 0 \text{ then } x \\\n&\quad \quad \text{else } x + f (x-1) \text{ / } f \text{ ]} \\
&\quad \text{[ 3 / } x \text{ ]}
\end{align*}
\]

Substitute argument for parameter
Substitute entire function for function name

The Result:

\[
\begin{align*}
&\text{(if } 3 \leq 0 \text{ then } 3 \text{ else } 3 + \\
&\quad \text{(rec } f \ x \rightarrow \\
&\quad \quad \text{if } x \leq 0 \text{ then } x \\\n&\quad \quad \text{else } x + f (x-1)\text{)) (3-1))}
\end{align*}
\]
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 f_val ->
       let v = eval e2 in
       substitute f_val f (substitute v x e)
     | _ -> raise TypeError)

pattern as x
match the pattern and binds x to value
(\texttt{rec} \ fact \ n = \texttt{if} \ n \ <= \ 1 \ \texttt{then} \ 1 \ \texttt{else} \ n \ * \ fact(n-1)) \ 3

\[ \rightarrow \]

\texttt{if} \ 3 \ < \ 1 \ \texttt{then} \ 1 \ \texttt{else} \ \texttt{3} \ * \ (\texttt{rec} \ fact \ n = \texttt{if} \ \ldots \ \texttt{then} \ \ldots \ \texttt{else} \ \ldots)) \ (3-1)

\[ \rightarrow \]

\texttt{3} \ * \ (\texttt{rec} \ fact \ n = \texttt{if} \ \ldots \ ) \ (3-1)

\[ \rightarrow \]

\texttt{3} \ * \ (\texttt{rec} \ fact \ n = \texttt{if} \ \ldots \ ) \ 2

\[ \rightarrow \]

\texttt{3} \ * \ (\texttt{if} \ 2 \ \leq \ 1 \ \texttt{then} \ 1 \ \texttt{else} \ 2 \ * \ (\texttt{rec} \ fact \ n = \ldots)(2-1))

\[ \rightarrow \]

\texttt{3} \ * \ (2 \ * \ (\texttt{rec} \ fact \ n = \ldots)(2-1))

\[ \rightarrow \]

\texttt{3} \ * \ (2 \ * \ (\texttt{rec} \ fact \ n = \ldots)(1))

\[ \rightarrow \]

\texttt{3} \ * \ 2 \ * \ (\texttt{if} \ 1 \ \leq \ 1 \ \texttt{then} \ 1 \ \texttt{else} \ 1 \ * \ (\texttt{rec} \ fact \ \ldots)(1-1))

\[ \rightarrow \]

\texttt{3} \ * \ 2 \ * \ 1
A MATHEMATICAL DEFINITION*
OF OCAML EVALUATION

* it’s a partial definition and this is a big topic; for more, see COS 510
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”
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.)
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.
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:

```
premise 1    premise 2    ...    premise n
  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:

\[
\begin{align*}
  e_1 & \rightarrow v_1 \\
  e_2 & \rightarrow v_2 \\
  \text{eval}_\text{op} (v_1, \text{op}, v_2) & = v' \\
  e_1 \text{ op } e_2 & \rightarrow v'
\end{align*}
\]

In English:

“If \( e_1 \) evaluates to \( v_1 \)
and \( e_2 \) evaluates to \( v_2 \)
and \( \text{eval}_\text{op} (v_1, \text{op}, v_2) \) is equal to \( v' \)
then
\( e_1 \text{ op } e_2 \) evaluates to \( v' \)

In code:

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

\[ i \in \mathbb{Z} \implies i \mapsto i \]

In English:

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

In code:

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

\[
\begin{align*}
\text{e1} & \rightarrow \text{v1} \\
\text{e2} & \rightarrow \text{v2}
\end{align*}
\]

\[\text{let } x = \text{e1} \text{ in } \text{e2} \rightarrow \text{v2}\]

In English:

“If e1 evaluates to v1 (which is a value) and e2 with v1 substituted for x evaluates to v2 then let x=e1 in e2 evaluates to v2.”

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  | Let_e(x,e1,e2) -> let v1 = eval e1 in
    eval (substitute v1 x e2)
  ...
```
An example rule concerning evaluation

As a rule:

\[ \lambda x. e \rightarrow \lambda x. e \]

In English:

“A function value evaluates to itself.”

In code:

```ocaml
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:

\[
\begin{align*}
    e_1 \rightarrow \lambda x.e & \quad e_2 \rightarrow v_2 & \quad e[v_2/x] \rightarrow v \\
    e_1 \, e_2 \rightarrow v
\end{align*}
\]

In English:

“if \( e_1 \) evaluates to a function with argument \( x \) and body \( e \) and \( e_2 \) evaluates to a value \( v_2 \) and \( e \) with \( v_2 \) substituted for \( x \) evaluates to \( v \) then \( e_1 \) applied to \( e_2 \) evaluates to \( v \)”

In code:

```ocaml
let rec eval (e:exp) : exp =
    match e with
    ...
    | FunCall_e (e1,e2) ->
        (match eval e1 with
        | Fun_e (x,e) -> eval (substitute (eval e2) x e)
        | ...)"
    ...
```
An example rule concerning evaluation

As a rule:

\[ e_1 \rightarrow \text{rec } f \ x = e \quad e_2 \rightarrow v \quad e[\text{rec } f \ x = e/f][v/x] \rightarrow v_2 \]
\[ e_1 \ e_2 \rightarrow v_2 \]

In English:

“uggh”

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  ...
  | (Rec e (f,x,e)) as f_val ->
    let v = eval e2 in
    substitute f_val (substitute v x e) g
```
Comparison: Code vs. Rules

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?

**complete eval code:**

```ocaml
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 f_val ->
    let v = eval e2 in
    substitute f_val f (substitute v x e)
```

**complete set of rules:**

\[
\begin{align*}
  &i \in \mathbb{Z} \\
  &i \rightarrow i \\
  &e1 \rightarrow v1 \\
  &e2 \rightarrow v2 \\
  &\text{eval}_\text{op} (v1, op, v2) = v \\
  &e1 \text{ op } e2 \rightarrow v \\
  &e1 \rightarrow v1 \\
  &e2 [v1/x] \rightarrow v2 \\
  &\text{let } x = e1 \text{ in } e2 \rightarrow v2 \\
  &\lambda x. e \rightarrow \lambda x. e \\
  &e1 \rightarrow \lambda x. e \\
  &e2 \rightarrow v2 \\
  &e[v2/x] \rightarrow v \\
  &e1 \text{ e2 } \rightarrow v \\
  &e1 \rightarrow \text{rec}_\text{f} x = e \\
  &e2 \rightarrow v2 \\
  &e[\text{rec}_\text{f} x = e/ f][v2/x] \rightarrow v3 \\
  &e1 \text{ e2 } \rightarrow v3
\end{align*}
\]
Comparison: Code vs. Rules

**complete eval code:**

```ocaml
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 f_val ->
    let v = eval e2 in
    substitute f_val f (substitute v x e)
```

**complete set of rules:**

1. \( \frac{i \in \mathbb{Z}}{i \rightarrow i} \)

2. \( \frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad \text{eval}_\text{op} (v_1, \text{op}, v_2) = v}{e_1 \text{ op } e_2 \rightarrow v} \)

3. \( \frac{e_1 \rightarrow v_1 \quad e_2 [v_1/x] \rightarrow v_2}{\text{let } x = e_1 \text{ in } e_2 \rightarrow v} \)

4. \( \lambda x.e \rightarrow \lambda x.e \)

5. \( \frac{e_1 \rightarrow \lambda x.e \quad e_2 \rightarrow v_2 \quad e[v_2/x] \rightarrow v}{e_1 e_2 \rightarrow v} \)

6. \( \frac{e_1 \rightarrow \text{rec } f \ x = e \quad e_2 \rightarrow v_2 \quad e[\text{rec } f \ x = e/f][v_2/x] \rightarrow v_3}{e_1 e_2 \rightarrow v_3} \)

- 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
Summary

• We can reason about OCaml programs using a substitution model.
  – integers, bools, strings, chars, and functions are values
  – value rule: values evaluate to themselves
  – let rule: “let x = e1 in e2” : substitute e1’s value for x into e2
  – fun call rule: “(fun x -> e2) e1”: substitute e1’s value for x into e2
  – rec call rule: “(rec x = e1) e2” : like fun call rule, but also substitute recursive function for name of function
    • To unwind: substitute (rec x = e1) for x in e1

• We can make the evaluation model precise by building an interpreter and using that interpreter as a specification of the language semantics.

• We can also specify the evaluation model using a set of inference rules
  – more on this in COS 510
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 is 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:
    • map f (map g xs) == map (comp f g) 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
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 optimization.
  – we can use it to formally prove that, for instance:
    • \( \text{map (comp f g) xs} = \text{map \( \text{comp (map f) g) xs} \)\)
  – we can open 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.

You can say that again!
I got it wrong the first time I tried, in 1932.
Fixed the bug by 1934, though.

Alonzo Church,
1903-1995
Princeton Professor,
1929-1967
substitute:

```haskell
fun xs -> map (+) xs
```

for \( f \) in:

```haskell
fun ys ->
    let map xs = 0::xs in
    f (map ys)
```

and if you don't watch out, you will get:

```haskell
fun ys ->
    let map xs = 0::xs in
    (fun xs -> map (+) xs) (map ys)
```
substitute:

fun xs -> map (+) xs

for f in:

fun ys ->
  let map xs = 0::xs in
  f (map ys)

and if you don't watch out, you will get:

fun ys ->
  let map xs = 0::xs in
  (fun xs -> map (+) xs) (map ys)

the problem was that the value you substituted in had a \textit{free variable} (map) in it that was \textit{captured}. 
substitute:

\[
\begin{align*}
\text{fun } xs & \rightarrow \text{map } (+) xs \\
\end{align*}
\]

for \( f \) in:

\[
\begin{align*}
\text{fun } ys & \rightarrow \\
& \quad \text{let } \text{map } xs = 0::xs \text{ in} \\
& \quad f (\text{map } ys) \\
\end{align*}
\]

to do it right, you need to rename some variables:

\[
\begin{align*}
\text{fun } ys & \rightarrow \\
& \quad \text{let } z \; xs = 0::xs \text{ in} \\
& \quad (\text{fun } xs \rightarrow \text{map } (+) xs) \; (z \; ys) \\
\end{align*}
\]
NOW WE ARE REALLY DONE!