An OCaml definition of OCaml evaluation, or,

Implementing OCaml in OCaml

COS 326
David Walker
Princeton University
To write a program, you have to know how the language works.

**Semantics**: The study of “how a programming language works”

**Methods** for defining program semantics:

- **Operational**: show how to rewrite program expressions step-by-step until you end up with a value
  - we’ve done some of this already
- **Denotational**: who how to compile a program into a different language that is well understood
  - we aren’t going to do much of this – see COS 510
- **Equational**: specify the equal programs
  - we’ll do more of this later & use this semantics to prove things about our programs
Today, we’ll focus on operational definitions

We’ll use the following techniques to communicate:

1. *examples* (good for intuition, but highly incomplete)
   – this doesn’t get at the corner cases
2. *an interpreter program* written in OCaml
3. *mathematical notation*
Today, we’ll focus on operational definitions

We’ll use the following techniques to communicate:

1. *examples* (good for intuition, but highly incomplete)
   – this doesn’t get at the corner cases
2. *an interpreter program* written in OCaml
3. *mathematical notation*
PRELIMINARIES

Reading: Note on “Operational Semantics”
https://www.cs.princeton.edu/courses/archive/fall18/cos326/notes/evaluation.php
Implementing an Interpreter

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"))`
- Evaluation
- data structure representing result of evaluation
  - `Num 6`
- Pretty Printing
- text file/stdout containing formatted output
  - `6`
- the fun parts
REPRESENTING SYNTAX
Program syntax is a complicated tree-like data structure.
Program syntax is a complicated tree-like data structure.

```
let x = 3 in
x + x
```
Program syntax is a complicated tree-like data structure.

let x = 3 in
x + x
Representing Syntax

More generally each let expression has 3 parts:

let  =  in
More generally each let expression has 3 parts:

And you can represent a let expression using a tree like this:
More generally each let expression has 3 parts:

```
let ∈ = ∈ in ∈
```

And you can represent a let expression using a tree like this:

- This part has to contain a variable, like x
- These parts contain arbitrary subexpressions
More generally each let expression has 3 parts:

```
let  =  in
```

And you create complicated programs by nesting let expressions (or any other expression) recursively inside one another:
OCaml for the Win

Functional programming languages have sometimes been called “domain-specific languages for compiler writers”

Datatypes are amazing for representing complicated tree-like structures and that is exactly what a program is.

Use a different constructor for every different sort of expression

• one constructor for variables
• one constructor for let expressions
• one constructor for numbers
• one constructor for binary operators, like add
• ...
Aside: Java for the loss

Languages like Java, that are based exclusively around heavy-weight class tend to be vastly more verbose when trying to represent syntax trees:

• one whole class for each different kind of syntax
• one class for variables
• one class for let expressions
• one class for numbers ...

In addition, writing traversals over the syntax is annoying, because your code is spread over N different classes (using a visitor pattern) rather than in one place.
Aside: Java for the loss

Languages like Java, that are based exclusively around heavy-weight class tend to be vastly more verbose when trying to represent syntax trees:

• one whole class for each different kind of syntax
• one class for variables
• one class for let expressions
• one class for numbers ...

In addition, writing traversals over the syntax is annoying, because your code is spread over N different classes (using a visitor pattern), rather than in one place.

SCORE: OCAML 3.8, JAVA 0

(C: who cares?)
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
```
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
```
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
```
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:

```
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:

```
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"))
  )
```

Making These Ideas Precise

Notice how the OCaml expression can be drawn as a tree.
Let \( x = 30 \),
\[
\begin{align*}
\text{let } & \ y = \text{let } z = 3, \\
& \quad \text{Op } (\text{Var } z, \times, \text{Int } 4), \\
& \quad \text{Op } (\text{Var } y, +, \text{Var } y)
\end{align*}
\]

By thinking about programs as their abstract syntax trees we can make certain notions, like the scope of a variable, which we’ve talked about before, more precise.
Free vs Bound Variables

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

let x = 30 in
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 the expression (let $x = 30$ in $x + y$)"
Other Examples

fun z -> z + y

- z is bound
- y is a free variable

match x with
  (y,z) -> y + z + w

- x, w are free variables
- y, z are bound

let rec f x =
  match x with
    [] -> y
    | hd:tl -> hd::f tl

- y is a free variable
- f, x, hd, tl are all bound
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
```
crawling up the tree to the nearest enclosing let...

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
crawling up the tree to the nearest enclosing let...

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

```plaintext
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```
Start with a let, and pick a fresh variable name, say “x”

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

```
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.

```plaintext
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!

```
let x = 30 in
let a =
    (let a = 3 in a*4)
in
a+a
```
Let’s do another let, renaming “a” to “y”.

```let
let x = 30 in
let a = (let a = 3 in a*4) in
a+a
```
Let’s do another let, renaming “a” to “y”.

\[
\begin{align*}
\text{let } x &= 30 \text{ in } \\
\text{let } y &= (\text{let } a = 3 \text{ in } a \times 4) \text{ in } \\
y + y
\end{align*}
\]
And if we rename the other let to “z”:

```plaintext
let x = 30 in
let y = (let z = 3 in z*4) in
y+y
```
type var = string

type op = Plus | Minus

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

let rec rename (x:var) (y:var) (e:exp) : exp =
Implementing Renaming

```ocaml
let rec rename (x:var) (y:var) (e:exp) : exp =
match e with
| Op_e (e1, op, e2) ->
| Var_e z ->
| Int_e i ->
| Let_e (z,e1,e2) ->
```

```ocaml
type var = string
type op = Plus | Minus
type exp =
| Int_e of int
| Op_e of exp * op * exp
| Var_e of var
| Let_e of var * exp * exp
```
Implementing Renaming

```ocaml
let rec rename (x:var) (y:var) (e:exp) : exp =
    match e with
    | Op_e (e1, op, e2) ->
      Op_e (rename x y e1, op, rename x y e2)
    | Var_e z ->
    | Int_e i ->
    | Let_e (z,e1,e2) ->
```

```ocaml
type var = string

type op = Plus | Minus

type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of var
  | Let_e of var * exp * exp
```
Implementing Renaming

define type var = string
define type op = Plus | Minus
define type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of var
  | Let_e of var * exp * exp

define let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op_e (e1, op, e2) ->
    Op_e (rename x y e1, op, rename x y e2)
  | Var_e z ->
    if z = x then Var_e y else e
  | Int_e i ->

  | Let_e (z,e1,e2) ->
Implementing Renaming

type var = string

type op = Plus | Minus

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

let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op_e (e1, op, e2) ->
    Op_e (rename x y e1, op, rename x y e2)
  | Var_e z ->
    if z = x then Var_e y else e
  | Int_e i ->
    Int_e i
  | Let_e (z,e1,e2) ->
type var = string

type op = Plus | Minus

let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op_e (e1, op, e2) ->
    Op_e (rename x y e1, op, rename x y e2)
  | Var_e z ->
    if z = x then Var_e y else e
  | Int_e i ->
    Int_e i
  | Let_e (z,e1,e2) ->
    Let_e (z, rename x y e1, if z = x then e2 else rename x y e2)
recall, we write:

\[ e_1 \rightarrow e_2 \]

\[ 2 + 3 \rightarrow 5 \]

to indicate that \( e_1 \) evaluates to \( e_2 \) in a single step

for example:
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
let x = 30 in
let y = 20 + x in
x+y

-->  
let y = 20 + 30 in
30+y

-->  
let y = 50 in
30+y

In this step, we just evaluated the right-hand side of the let. We now have a \textit{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
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
\( x + y \)

\[
\text{let } y = 20 \text{ in } 30 + y
\]
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
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.
Two Other Auxiliary Functions

(* 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}
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> 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)
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

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.

```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)
```
Simpler but Dangerous

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!
**Exceptions**

```ocaml
define 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.
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, 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

"Do I contradict myself?
Very well then, I contradict myself.
I am large; I contain multitudes."
Walt Whitman
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 $v$.

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

... 
```
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 \((v: \text{exp}) (x: \text{variable}) (e: \text{exp})\) : \text{exp} =

let \text{rec} subst \((e: \text{exp})\) : \text{exp} =

match \(e\) with

| \(\text{Int}_e \_\_\) -> \(e\) \\
| \(\text{Op}_e(e1,op,e2)\) -> \(\text{Op}_e(\text{subst } e1,op,\text{subst } e2)\) \\
| \(\text{Var}_e y\) -> if \(x = y\) then \(v\) else \(e\) \\
| \(\text{Let}_e (y,e1,e2)\) ->

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
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)
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**

```ocaml
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)
A function call

\texttt{fact 3}

is implemented as

\texttt{FunCall_e (Var_e "fact", Int_e 3)}
**Scaling up the Language**

```ocaml
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)`
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)
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.
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 with
     | 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)
Let and Lambda

\[
\text{let } x = 1 \text{ in } x + 41
\]

\[
\rightarrow
\]

\[
1 + 41
\]

\[
\rightarrow
\]

\[
42
\]

In general:

\[
(f\text{un } x \to x + 41) \ 1
\]

\[
\rightarrow
\]

\[
1 + 41
\]

\[
\rightarrow
\]

\[
42
\]

\[
(f\text{un } x \to e2) \ e1 \quad == \quad \text{let } x = e1 \text{ in } e2
\]
So we could write:

```ml
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

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

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

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

```ocaml
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
```
Recursive definitions

```ocaml
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 _ -> false
```

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

A better IR would just delete Fun_e – avoid unnecessary redundancy
Interlude: Notation for Substitution

“Substitute value $v$ for variable $x$ in expression $e$:” \[ e \left[ \frac{v}{x} \right] \]

deﬁnitions of substitution:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Substitution</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x + y) \left[ \frac{7}{y} \right]$</td>
<td>is</td>
<td>$(x + 7)$</td>
</tr>
<tr>
<td>$(\text{let } x = 30 \text{ in } \text{let } y = 40 \text{ in } x + y) \left[ \frac{7}{y} \right]$</td>
<td>is</td>
<td>$(\text{let } x = 30 \text{ in } \text{let } y = 40 \text{ in } x + y)$</td>
</tr>
<tr>
<td>$(\text{let } y = y \text{ in } \text{let } y = y \text{ in } y + y) \left[ \frac{7}{y} \right]$</td>
<td>is</td>
<td>$(\text{let } y = 7 \text{ in } \text{let } y = y \text{ in } y + y)$</td>
</tr>
</tbody>
</table>
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:

```plaintext
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:

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

The Substitution:

\[(\text{if } x \leq 0 \text{ then } x \text{ else } x + f(x-1)) \left[ \begin{array}{c} \text{rec } f \ x \rightarrow \\ \begin{cases} x \leq 0 & \text{then } x \\ \text{else } x + f(x-1) \end{cases} \\ 3 / x \end{array} \right]\]

- Substitute entire function for function name
- Substitute argument for parameter

The Result:

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

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

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

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

-->
3 * (\texttt{rec fact n = if ...}) 2

-->
3 * (\texttt{if 2 <= 1 then 1 else 2 * (rec fact n = ...)(2-1)})

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

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

-->
3 * 2 * (\texttt{if 1 <= 1 then 1 else 1 * (rec fact ...})(1-1))

-->
3 * 2 * 1
Datatypes are very useful for *representing* the *abstract syntax* of programming languages

- Moral: If you are going to implement a programming language, you really should be using a functional language with data types

Interpreters are recursive programs that evaluate expressions and produce values.