An OCaml definition of OCaml evaluation, or,

Implementing OCaml in OCaml

COS 326
Andrew Appel
Princeton University
Defining Programming Language Semantics

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**: how interpret a program in 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
- **Axiomatic**: provide (other kinds of) reasoning rules about programs
Defining Program Semantics

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*

our focus today
PRELIMINARIES

Reading: Note on “Operational Semantics”
Implementing an Interpreter

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
REPRESENTING SYNTAX
Representing Syntax

Program syntax is a complicated tree-like data structure.
Representing Syntax

Program syntax is a complicated tree-like data structure.

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

This is the “parse tree.” Useful for some purposes, but for the semantics it’s Too Much Information.
Abstract Syntax Tree (AST)

Don’t need to represent all the “punctuation”

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

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

More generally each let expression has 3 parts:

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
Representing Syntax

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
- ...

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.
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.

(C: who cares?)

SCORE: OCAML 3.8, JAVA 0
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:

type variable = string

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:

```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:

```plaintext
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")
)
```
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
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")

ASTs as ... Trees

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

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*.

```
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”

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

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*.

```
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
```

This a is bound by this let
Abstract Syntax Trees

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

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

```ocaml
let x = 30 in
let y =
  (let a = 3 in a*4)
in
y+y
```
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 =
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) ->

  | Var_e z ->

  | Int_e i ->

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

```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

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) ->
```
Implementing Renaming

```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

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) ->
```
type var = string

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) ->
Implementing Renaming

```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

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:

\[
\begin{array}{c}
e_1 \quad \rightarrow \quad e_2 \\
\end{array}
\]

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

for example:

\[
\begin{array}{c}
2 + 3 \quad \rightarrow \quad 5 \\
\end{array}
\]
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
\( x + y \)

\[
\begin{align*}
\text{let } y &= 20 + 30 \\
30 + y
\end{align*}
\]

\[
\begin{align*}
\text{let } y &= 50 \\
30 + y
\end{align*}
\]

\[
\begin{align*}
30 + 50
\end{align*}
\]

*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
\( x + y \)
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} : value \rightarrow op \rightarrow value \rightarrow exp

(* substitute v x e: replace free occurrences of x with v in e *)
\textbf{substitute} : value \rightarrow variable \rightarrow exp \rightarrow 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 \rightarrow bool
\n\textbf{eval\_op} : value \rightarrow op \rightarrow value \rightarrow value
\n\textbf{substitute} : value \rightarrow variable \rightarrow exp \rightarrow exp

\textbf{let rec} eval (e:exp) : exp =
\quad \textbf{match} e \textbf{ with}
\quad | \text{Int\_e} i \rightarrow
\quad | \text{Op\_e} (e_1,op,e_2) \rightarrow
\quad | \text{Let\_e} (x,e_1,e_2) \rightarrow
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.
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.
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?
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
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.: the input might be ill formatted
- For errors that *cannot occur* (unless the program itself has a bug) and for which there are few "entry points" (few places checks needed) use exceptions
  - Java does not follow this rule: objects may be null *everywhere*
AUXILIARY FUNCTIONS
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: 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: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 el else subst el,
    if x = y then e2 else subst e2)

in
subst e
let substitute (v:expr) (x:variable) (e:expr) : expr =
let rec subst (e:expr) : expr =
    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

```
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)}
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)
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)
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)

*Note:* 
1. 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 with
         | Fun_e (x,e) -> eval (substitute (eval e2) 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 (ef,e1) ->
        (match eval ef with
        | Fun_e (x,e2) -> eval (substitute (eval e1) x e2)
        | _ -> raise TypeError)
Let and Lambda

```
let x = 1 in x+41
--> 1+41
--> 42
```

In general:

```
(fun x -> x+41) 1
--> 1+41
--> 42
```

```
(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

```plaintext
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
```

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

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)

(rewrite)
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 (_,_,_) -> 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
```
Interlude: Notation for Substitution

“Substitute value \( v \) for variable \( x \) in expression \( e \):” \( e [ v / x ] \)

Examples of substitution:

\[
\begin{align*}
(x + y) [7/y] & \quad \text{is} \quad (x + 7) \\
(\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y) [7/y] & \quad \text{is} \quad (\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y) \\
(\text{let } y = y \text{ in let } y = y \text{ in } y + y) [7/y] & \quad \text{is} \quad (\text{let } y = 7 \text{ in let } y = y \text{ in } y + y)
\end{align*}
\]
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
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:

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

The Substitution:

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

Substitute argument for parameter

Substitute entire function for function name

The Result:

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

\[(\begin{cases} 3 & \text{if } x \leq 0 \\ 3 + f(3-1) & \text{else} \end{cases}) (3-1)\]
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
(\text{rec } \text{fact } n = \text{if } n \leq 1 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)) \ 3 \\
\quad \Rightarrow \\
\text{if } 3 < 1 \text{ then } 1 \text{ else } \\
\quad 3 \times (\text{rec } \text{fact } n = \text{if } \ldots \text{ then } \ldots \text{ else } \ldots) \ (3-1) \\
\quad \Rightarrow \\
3 \times (\text{rec } \text{fact } n = \text{if } \ldots) \ (3-1) \\
\quad \Rightarrow \\
3 \times (\text{rec } \text{fact } n = \text{if } \ldots) \ 2 \\
\quad \Rightarrow \\
3 \times (\text{if } 2 \leq 1 \text{ then } 1 \text{ else } 2 \times (\text{rec } \text{fact } n = \ldots)(2-1)) \\
\quad \Rightarrow \\
3 \times (2 \times (\text{rec } \text{fact } n = \ldots)(2-1)) \\
\quad \Rightarrow \\
3 \times (2 \times (\text{rec } \text{fact } n = \ldots)(1)) \\
\quad \Rightarrow \\
3 \times 2 \times (\text{if } 1 \leq 1 \text{ then } 1 \text{ else } 1 \times (\text{rec } \text{fact } \ldots)(1-1)) \\
\quad \Rightarrow \\
3 \times 2 \times 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.