Implementing OCaml in OCaml

COS 326

David Walker

Princeton University
Implementing an Interpreter

text file containing program as a sequence of characters

let x = 3 in x + x

Parsing

data structure representing program

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

data structure representing result of evaluation

Num 6

Evaluation

Pretty Printing

6

text file/stdout containing with formatted output

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

as an exp value:

```ocaml
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 this reflects the “tree”:

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

```
type exp =
    | Int_e of int
    | Op_e of exp * op * exp
    | Var_e of variable
    | Let_e of variable * exp * exp
```
let is_value (e: exp) : bool =
match e with
| Int_e _ -> true
| ( Op_e _
  | Let_e _
  | Var_e _ ) -> false

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

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

substitute : value -> variable -> exp -> exp
A Simple Evaluator

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

let rec eval (e:exp) : exp =
  match e with
  | Int_e i ->
  | Op_e(e1,op,e2) ->
  | Let_e(x,e1,e2) ->
is_value : \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 \exp \rightarrow \exp  

let rec eval (e:exp) : exp = 
    match e with  
    | \text{Int}_e i \rightarrow \text{Int}_e i  
    | \text{Op}_e(e1,op,e2) \rightarrow  
    | \text{Let}_e(x,e1,e2) \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 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′
Shorter but Dangerous

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

\textbf{let rec eval (e:exp) : exp =}
  \text{match e with}
  | \text{Int\_e i} \rightarrow \text{Int\_e i}
  | \text{Op\_e(e1,op,e2)} \rightarrow
    \text{eval\_op (eval e1) op (eval e2)}
  | \text{Let\_e(x,e1,e2)} \rightarrow
    \text{eval (substitute (eval e1) x e2)}

Why?
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.
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
        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)
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 -> ???
```

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

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

But that will create a mess of Ocaml warnings – bad style. (Bad for debugging.)
We Could Use Options:

```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.
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 explained why to use an option type (or a Success/Failure datatype) instead of raising an exception.

No! I do not contradict myself!

• The other example was for errors that *will occur* (because the input might be ill formatted).
• This example is for errors that *cannot occur* (unless the program itself has a bug).
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: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.
**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
;;
```

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

A function call
fact 3
is implemented as
FunCall_e (Var_e “fact”, Int_e 3)
**Scaling up the Language**

```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)`
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)

43
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 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:

```plaintext
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

\[
\text{let rec } f \ x = f \ (x+1) \ \text{in } f \ 3
\]

\[
\text{let } f = (\text{rec } f \ x \rightarrow f \ (x+1)) \ \text{in } f \ 3
\]

\[
\text{let } g = (\text{rec } f \ x \rightarrow f \ (x+1)) \ \text{in } g \ 3
\]

\[
\begin{align*}
\text{Let}_e \ ("g", \\
\quad \text{Rec}_e \ ("f", "x", \\
\quad \quad \text{FunCall}_e \ (\text{Var}_e \ "f", \ \text{Op}_e \ (\text{Var}_e \ "x", \ Plus, \ \text{Int}_e \ 1))) \\
\quad \), \\
\quad \text{FunCall} \ (\text{Var}_e \ "g", \ \text{Int}_e \ 3)
\end{align*}
\]
Recursive definitions

```ocaml
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 ;;
```

```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 ;;
```
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 ;;
```

```ocaml
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 $e$:”

$e \ [ \ v / x \ ]$

examples of substitution:

$(x + y) \ [ / y] \quad \text{is} \quad (x + 7)$

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

$(\text{let } y = y \ \text{in let } y = y \ \text{in } y + y) \ [ / y] \quad \text{is} \quad (\text{let } y = 7 \ \text{in let } y = y \ \text{in } y + y)$
Basic evaluation rule for recursive functions:

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

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

The Substitution:

\[
(\text{if } x \leq 0 \text{ then } x \text{ else } x + f(x-1))
\]

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

Substitute argument for parameter
Substitute entire function for function name

The Result:

\[
(\text{if } 3 \leq 0 \text{ then } 3 \text{ else } 3 + \\
(\text{rec } f \ x \rightarrow \\
\text{if } x \leq 0 \text{ then } x \\
\text{else } x + f(x-1)) \ (3-1))
\]
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
             substitute f_val f (substitute v x e)
         | _ -> raise TypeError)

**pattern** as x
match the pattern and binds x to value
More Evaluation

(rec fact n = if n <= 1 then 1 else n * fact(n-1)) 3

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

--> 3 * (rec fact n = if ... ) (3-1)

--> 3 * (rec fact n = if ... ) 2

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

--> 3 * (2 * (rec fact n = ...)(2-1))

--> 3 * (2 * (rec fact n = ...)(1))

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

--> 3 * 2 * 1
A MATHEMATICAL DEFINITION* OF OCAMLR 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”
- **disadvantage:** defining a language in itself is a logical fallacy
PL researchers have developed their own standard notation for writing down how programs execute

– it has a mathematical “feel” that makes PL researchers feel special and gives us goosebumps inside
– it operates over abstract expression syntax like “e1 + e2”
– it is useful to know this notation if you want to read specifications of programming language semantics

• e.g.: Standard ML (of which OCaml is a descendent) has a formal definition given in this notation (and C, and Java; but not OCaml...)
• e.g.: most papers in the conference POPL (ACM Principles of Prog. Lang.)
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:

\[
\text{premise 1} \quad \text{premise 2} \quad \ldots \quad \text{premise 3} \\
\text{conclusion}
\]

You read a rule like this:

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

Some rules have no premises

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

As a rule:

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

In English:

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

In code:

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

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

As a rule:

\[
\frac{i \in \mathbb{Z}}{i \rightarrow i}
\]

asserts \( i \) is an integer

In English:

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

In code:

```ocaml
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  ...
```
An example rule concerning evaluation

As a rule:

\[
\begin{align*}
  e_1 &\rightarrow v_1 & e_2 [v_1/x] &\rightarrow v_2 \\
  \text{let } x = e_1 \text{ in } e_2 &\rightarrow v_2
\end{align*}
\]

In English:

“If \( e_1 \) evaluates to \( v_1 \) (which is a value) and \( e_2 \) with \( v_1 \) substituted for \( x \) evaluates to \( v_2 \) then \( \text{let } x = e_1 \text{ in } e_2 \) evaluates to \( v_2 \).”

In code:

```
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:

```plaintext
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:

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

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:

```
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:

\[
\begin{align*}
e1 \rightarrow & \text{rec } f \ x = e \\
e2 \rightarrow & \ v \\
e[\text{rec } f \ x = e/f][v/x] \rightarrow & \ v2 \\
e1 \ e2 \rightarrow & \ v2
\end{align*}
\]

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 `-->` premise in a rule
- what’s the main difference?
Comparison: Code vs. Rules

complete eval code:

```plaintext
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:

```
e1 --> v1    e2 --> v2    eval_op (v1, op, v2) = v
  e1 op e2 --> v
```

```
e1 --> v1    e2 [v1/x] --> v2
  let x = e1 in e2 --> v2
```

```
lx.e --> lx.e
```

```
e1 --> lx.e    e2 --> v2    e[v2/x] --> v
  e1 e2 --> v
```

```
e1 --> rec fx = e    e2 --> v2    e[rec fx = e/f][v2/x] --> v3
  e1 e2 --> v3
```

- There’s no formal rule for handling free variables
- No rule for evaluating function calls when a non-function in the caller position
- In general, **no rule when further evaluation is impossible**
  - the rules express the **legal evaluations** and say nothing about what to do in error situations
  - the code handles the error situations by raising exceptions
  - type theorists prove that well-typed programs don’t run into undefined cases
• We can reason about OCaml programs using a substitution model.
  – integers, booleans, strings, characters, 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:
    • \( \text{map } f (\text{map } g \, xs) = \text{map } (\text{comp } f \, g) \, xs \)
  – proof: by induction on the length of the list \( xs \), using the definitions of the substitution model.
  – we can even model complicated systems (e.g., by using a small functional language and substitution-based evaluation.

• It is not useful for reasoning about execution semantics 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:

\[
\text{fun } xs \to \text{map } (+) \ xs
\]

for \( f \) in:

\[
\text{fun } ys \to \\
\quad \text{let } \text{map } xs = 0::xs \text{ in} \\
\quad f (\text{map } ys)
\]

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

\[
\text{fun } ys \to \\
\quad \text{let } \text{map } xs = 0::xs \text{ in} \\
\quad (\text{fun } xs \to \text{map } (+) \ xs) (\text{map } ys)
\]
Church's mistake

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 free variable (map) in it that was captured.
substitute:

fun xs -> map (+) xs

for f in:

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

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

fun ys ->
  let z xs = 0::xs in
  (fun xs -> map (+) xs) (z ys)
NOW WE ARE REALLY DONE!