An OCaml definition of OCaml evaluation, or,

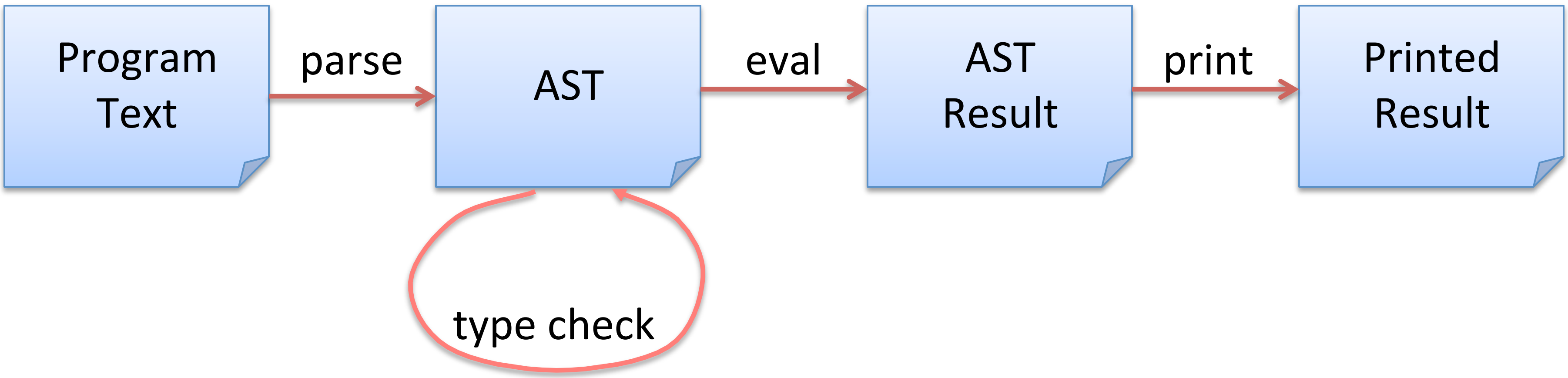
Implementing OCaml in OCaml (Part II)

COS 326
David Walker
Princeton University

slides copyright 2017 David Walker
permission granted to reuse these slides for non-commercial educational purposes

Last Time

Implementing an interpreter:



Components:

- Evaluator for primitive operations
- Substitution
- Recursive evaluation function for expressions

Our Interpreter


```
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)
```

Our Interpreter


```
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)
```

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

Our Interpreter

Example to interpret:

```
let z = 2 in
let z = 3 + z in
z
```

How to interpret a let expression:

```
eval (let x = e1 in e2) ->
eval {substitute (eval e1) x e2}
```

How to substitute v for x into a let expression

```
substitute v x (let y = e1 in e2) ==
        let y = substitute v x e1 in
        (if x = y then e2 else substitute v x e2)
```

Our Interpreter

Example to interpret:

```
let z = 2 in
let z = 3 + z in
z
```

How to interpret a let expression:

```
eval (let x = e1 in e2) ->
eval {substitute (eval e1) x e2}
```

```
== eval { substitute (eval 2) z (let z = 3 + z in z) }
```

How to substitute v for x into a let expression

```
substitute v x (let y = e1 in e2) ==
        let y = substitute v x e1 in
        (if x = y then e2 else substitute v x e2)
```

Our Interpreter

Example to interpret:

```
let z = 2 in
let z = 3 + z in
z
```

How to interpret a let expression:

```
eval (let x = e1 in e2) ->
eval {substitute (eval e1) x e2}
```

```
== eval { substitute (eval 2) z (let z = 3 + z in z) }
```

```
== eval { substitute 2 z (let z = 3 + z in z)        }
```

How to substitute v for x into a let expression

```
substitute v x (let y = e1 in e2) ==
        let y = substitute v x e1 in
        (if x = y then e2 else substitute v x e2)
```

Our Interpreter

Example to interpret:

```
let z = 2 in
let z = 3 + z in
z
```

How to interpret a let expression:

```
eval (let x = e1 in e2) ->
eval {substitute (eval e1) x e2}
```

```
== eval { substitute (eval 2) z (let z = 3 + z in z) }
```

```
== eval { substitute 2 z (let z = 3 + z in z)          }
```

```
== eval { (let z = (substitute 2 z (3 + z)) in z)      }
```

How to substitute v for x into a let expression

```
substitute v x (let y = e1 in e2) ==
        let y = substitute v x e1 in
        (if x = y then e2 else substitute v x e2)
```

Our Interpreter

Example to interpret:

```
let z = 2 in
let z = 3 + z in
z
```

How to interpret a let expression:

```
eval (let x = e1 in e2) ->
eval {substitute (eval e1) x e2}
```

```
== eval { substitute (eval 2) z (let z = 3 + z in z) }
```

```
== eval { substitute 2 z (let z = 3 + z in z)          }
```

```
== eval { (let z = (substitute 2 z (3 + z)) in z)      }
```

notice we don't substitute 2 in z here

How to substitute v for x into a let expression

```
substitute v x (let y = e1 in e2) ==
        let y = substitute v x e1 in
        (if x = y then e2 else substitute v x e2)
```

Our Interpreter

Example to interpret:

```
let z = 2 in
let z = 3 + z in
z
```

How to interpret a let expression:

```
eval (let x = e1 in e2) ->
eval {substitute (eval e1) x e2}
```

```
== eval { substitute (eval 2) z (let z = 3 + z in z) }
== eval { substitute 2 z (let z = 3 + z in z)        }
== eval { (let z = (substitute 2 z (3 + z)) in z)    }
== eval { let z = 3 + 2 in z                          }
```

How to substitute v for x into a let expression

```
substitute v x (let y = e1 in e2) ==
        let y = substitute v x e1 in
        (if x = y then e2 else substitute v x e2)
```

SCALING UP THE LANGUAGE

(MORE FEATURES, MORE FUN)

Scaling up the Language


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

Scaling up the Language

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

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

Scaling up the Language


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

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

Scaling up the Language


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

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

Functions are values!

Easy exam question:
What value does the OCaml interpreter produce when you enter (fun x -> 3) in to the prompt?
Answer: the value produced is (fun x -> 3)

Scaling up the Language


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

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

Function calls are not values.

Scaling up the Language


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

Scaling up the Language

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

values (including functions) always evaluate to themselves.

Scaling up the Language


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

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

Scaling up the Language


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

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

Scaling up the Language

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

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

Simplifying a little


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

Simplifying a little


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

This looks like the case for let!

Let and Lambda


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

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

In general:

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

So we could write:


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

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

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

Recursive definitions


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

(rewrite)

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

(alpha-convert)

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

(implement)

```
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


```
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


```
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 o  _,_,_) -> 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:

(x + y) [7/y]	is	(x + 7)
(let x =30 in let y=40 in x + y) [7/y]	is	(let x =30 in let y=40 in x + y)
(let y = y in let y = y in y + y) [7/y]	is	(let y = 7 in let y = y in y + y)

Evaluating Recursive Functions

Basic evaluation rule for recursive functions:

```
(rec f x = body) arg      -->      body [arg/x] [rec f x = body/f]
```

argument value substituted for parameter

entire function substituted for function name

Evaluating Recursive Functions

Start out with
a let bound to
a recursive function:

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

The Substitution:

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

The Result:

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

Evaluating Recursive Functions

Recursive Function Call:

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

The Substitution:

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

Substitute argument for parameter

Substitute entire function for function name

The Result:

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

Evaluating Recursive Functions


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

        | (Rec_e (f,x,e)) as f_val ->
            let v = eval e2 in
            eval (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 OCAML EVALUATION

* it's a partial definition and this is a big topic; for more, see COS 510

From Code to Abstract Specification

OCaml code can give a language semantics

- advantage: it can be executed, so we can try it out
- advantage: it is amazingly concise
  - especially compared to what you would have written in Java
- disadvantage: it is a little ugly to operate over concrete ML datatypes like "Op_e(e1,Plus,e2)" as opposed to "e1 + e2"

From Code to Abstract Specification

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

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

Goal

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

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

Formal Inference Rules

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

A rule looks like this:

$$\frac{\text{premise 1} \qquad \text{premise 2} \qquad \ldots \qquad \text{premise 3}}{\text{conclusion}}$$

You read a rule like this:

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

Some rules have no premises

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

An example rule

As a rule:

$$\frac{\texttt{e1 --> v1} \qquad \texttt{e2 --> v2} \qquad \texttt{eval\_op (v1, op, v2) == v'}}{\texttt{e1 op e2 --> v'}}$$

In English:

"If e1 evaluates to v1
and e2 evaluates to v2
and eval_op (v1, op, v2) is equal to v'
then
e1 op e2 evaluates to v'

In code:

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

An example rule

As a rule:

$$\frac{i \in Z}{i \dashrightarrow i}$$

asserts i is
an integer

In English:

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

In code:

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

An example rule concerning evaluation

As a rule:

$$\frac{\text{e1 --> v1} \qquad\qquad \text{e2 [v1/x] -->* v2}}{\text{let x = e1 in e2 --> v2}}$$

In English:

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

In code:

```
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:

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

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

In English:

"A function value evaluates to itself."

In code:

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

An example rule concerning evaluation

As a rule:

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

In English:

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

In code:

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

An example rule concerning evaluation

As a rule:

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

In English:

“uggh”

In code:

```
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

complete eval code:

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

complete set of rules:

$$\frac{i \in Z}{i \;\text{-->}\; i}$$

$$\frac{e1 \;\text{-->}\; v1 \qquad e2 \;\text{-->}\; v2 \qquad eval\_op\ (v1, op, v2) == v}{e1\ op\ e2 \;\text{-->}\; v}$$

$$\frac{e1 \;\text{-->}\; v1 \qquad e2\ [v1/x] \;\text{-->}\; v2}{let\ x = e1\ in\ e2 \;\text{-->}\; v2}$$

$$\frac{}{\lambda x.e \;\text{-->}\; \lambda x.e}$$

$$\frac{e1 \;\text{-->}\; \lambda x.e \qquad e2 \;\text{-->}\; v2 \qquad e[v2/x] \;\text{-->}\; v}{e1\ e2 \;\text{-->}\; v}$$

$$\frac{e1 \;\text{-->}\; rec\ f\ x = e \qquad e2 \;\text{-->}\; v2 \qquad e[rec\ f\ x = e/f][v2/x] \;\text{-->}\; v3}{e1\ e2 \;\text{-->}\; v3}$$

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:

```
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:

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

$$\frac{e1 \dashrightarrow v1 \qquad e2 \dashrightarrow v2 \qquad \text{eval\_op}\ (v1, op, v2) == v}{e1\ op\ e2 \dashrightarrow v}$$

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

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

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

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

- 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

Summary

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

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

- We can also specify the evaluation model using a set of *inference rules*
  - more on this in COS 510

Some Final Words

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

Some Final Words

- The substitution model is only a model.
  - it does not accurately model all of OCaml's features
    - I/O, exceptions, mutation, concurrency, …
    - we can build models of these things, but they aren't as simple.
    - even substitution was tricky to formalize!
- It's useful for reasoning about higher-ord… correctness of algorithms, and optimizati…
  - we can us… …prove that, for in…
    - …
    - …n of the li… …odel.
  - we … …stems (e.g. … …g a small fu… …and substitutio… …n.
- It is *not* useful for reasoning about exec… …e
  - 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

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)
```

Church's mistake

substitute:

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

the problem was that the value you substituted in had a *free variable* (map) in it that was *captured*.

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)
```

Church's mistake

substitute:

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

for f in:

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

to do it right, you rename (alpha-convert) some variables:

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

ASSIGNMENT #4

Two Parts

Part 1: Build your own interpreter

– More features: booleans, pairs, lists, match
– Different model: environment-based vs substitution-based
  - The abstract syntax tree Fun_e(_,_) *is no longer a value*
    – *a Fun_e is not a result of a computation*
  - There is one more computation step to do:
    – creation of a *closure* from a Fun_e expression

Part 2: Prove facts about programs using equational reasoning

– we already saw a bit of equational reasoning today:
  - if e1 --> e2 then e1 == e2
– more next week

FUNCTION CLOSURES

Closures

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

Closures

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
    choose (true, 1, 2)
```

Closures

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
    choose (true, 1, 2)
-->
    let (b, x, y) = (true, 1, 2) in
    if b then (fun n -> n + x)
    else (fun n -> n + y)
```

Closures

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
    choose (true, 1, 2)
-->
    let (b, x, y) = (true, 1, 2) in
    if b then (fun n -> n + x)
    else (fun n -> n + y)
-->
    if true then (fun n -> n + 1)
    else (fun n -> n + 2)
```

Closures

Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

Its execution behavior according to the substitution model:

```
    choose (true, 1, 2)
-->
    let (b, x, y) = (true, 1, 2) in
    if b then (fun n -> n + x)
    else (fun n -> n + y)
-->
    if true then (fun n -> n + 1)
    else (fun n -> n + 2)
-->
    (fun n -> n + 1)
```

Substitution and Compiled Code


```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

Substitution and Compiled Code


```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

compile →

```
choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret

main:
  ...
  jmp choose
```

Substitution and Compiled Code

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

compile

```
choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret

main:
  ...
  jmp choose
```

execute with substitution

```
let (b, x, y) = (true, 1, 2) in
if b then
  (fun n -> n + x)
else
  (fun n -> n + y)
```

Substitution and Compiled Code


```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

compile

```
choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret

main:
  ...
  jmp choose
```

execute with substitution

```
let (b, x, y) = (true, 1, 2) in
if b then
  (fun n -> n + x)
else
  (fun n -> n + y)
```

execute with substitution
==
generate new code block with parameters replaced by arguments

Substitution and Compiled Code


```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)


choose (true, 1, 2)
```

compile

```
choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret

main:
  ...
  jmp choose
```

execute with substitution

```
let (b, x, y) = (true, 1, 2) in
if b then
  (fun n -> n + x)
else
  (fun n -> n + y)
```

execute with substitution
==
generate new code block with parameters replaced by arguments

```
choose:
  mov rb
  mov rx
  mov ry
  ...
  jmp re

main:
  ...
  jmp choose
```

```
choose_subst:
  mov rb 0xF8[0]
  mov rx 0xF8[4]
  mov ry 0xF8[8]
  compare rb 0
  ...
  jmp ret
```

```
0xF8: 0
      1
      2
```

Substitution and Compiled Code


```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)

choose (true, 1, 2)
```

compile

```
choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret

main:
  ...
  jmp choose
```

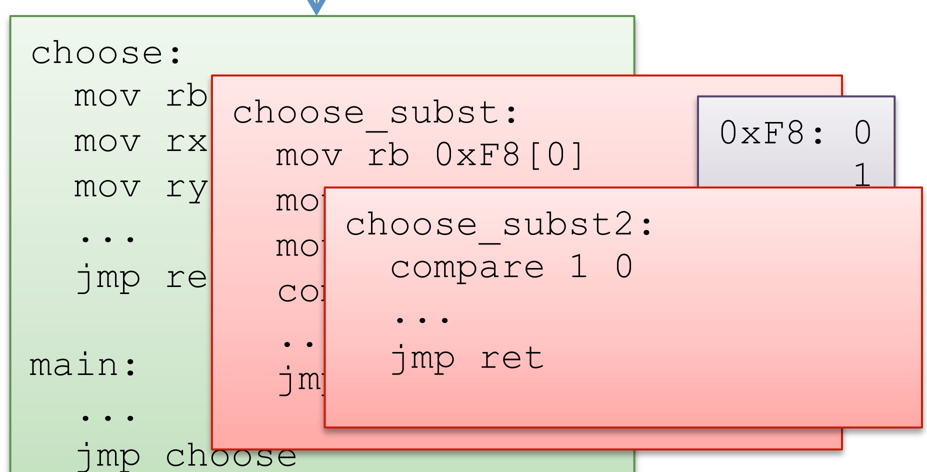
execute with substitution

```
let (b, x, y) = (true, 1, 2) in
if b then
  (fun n -> n + x)
else
  (fun n -> n + y)
```

execute with substitution

```
if true then
  (fun n -> n + 1)
else
  (fun n -> n + 2)
```

execute with substitution
==
generate new code block with parameters replaced by arguments



What we aren't going to do

- The substitution model of evaluation is *just a model*. It says that we generate new code at each step of a computation. We don't do that in reality. Too expensive!

- The substitution model is a faithful model for reasoning about the relationship between inputs and outputs of a function but it doesn't tell us much about the resources that are used along the way.

- I'm going to tell you a little bit about how ML programs are compiled so you can understand how much space your programs will use. Understanding the space consumption of your programs is an important component in making these programs more efficient.

Compiling functions

General tactic: Reduce the problem of compiling ML-like functions to the problem of compiling C-like functions.

Some functions are already C-like:

```
let add (x:int*int) : int =
  let (y,z) = x in
  y + z
```

```
# argument in r1
# return address in r0

add:
  ld r2, r1[0]      # y in r2
  ld r3, r1[4]      # z in r3
  add r4, r2, r3    # sum in r4
  jmp r0
```

But what about nested, higher-order functions?


```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
```

?

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    f1
  else
    f2
```

?

```
let f1 n = n + x
```

?

```
let f2 n = n + y
```

But what about nested, higher-order functions?

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
```

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    f1
  else
    f2
```

```
let f1 n = n + x
```

```
let f2 n = n + y
```

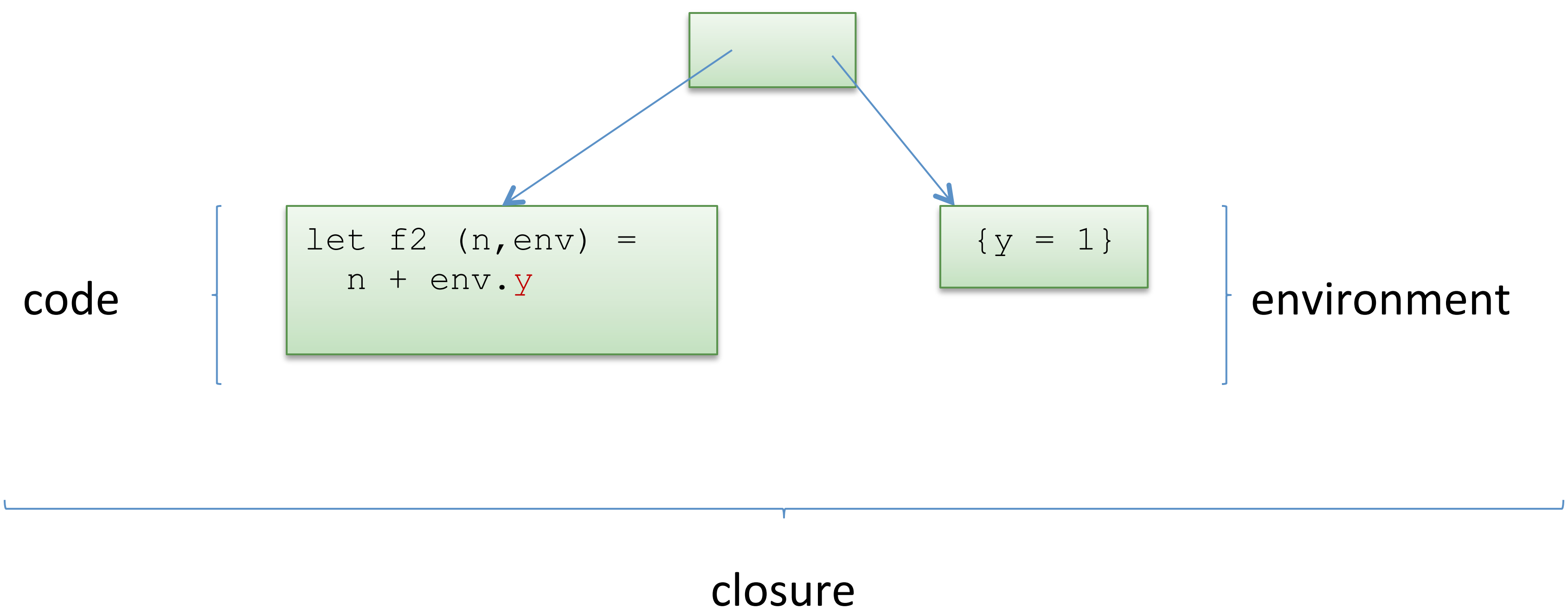
Darn! *Doesn’t work naively*. Nested functions contain *free variables*.
Simple unnesting leaves them undefined.

But what about nested, higher-order functions?

- We can't execute a function like the following:

```
let f2 n = n + y
```

- But we can execute a *closure* which is a pair of some code and an environment:



Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions into closed, top-level functions.

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)
```

Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions in to closed, top-level functions.

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)
```

```
let choose (arg,env) =
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})
```

```
let f1 (n,env) =
  n + env.xe + env.ye
```

```
let f2 (n,env) =
  n + env.ye
```

add environment parameter

create closures

use environment variables instead of free variables

Closure Conversion

Closure conversion converts open, nested functions in to closed, top-level functions.

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)
```

add environment parameter

```
let choose (arg,env) =
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})
```

create closures

```
let f1 (n,env) =
  n + env.xe + env.ye
```

```
let f2 (n,env) =
  n + env.ye
```

use environment variables instead of free variables

```
(choose (true,1,2)) 3
```

```
let c_closure        = (choose, ())                in (* create closure *)
let (c_code, c_cenv) = c_closure                   in (* extract code, env *)
let f_closure        = c_code ((true,1,2), c_env)  in (* call choose code, extract f code, env *)
let (f_code, f_env)  = f_closure                   in (* extract code, env *)
f_code (3, f_env)                                     (* call f code *)
;;
```

Closure Conversion

Closure conversion converts open, nested functions in to closed, top-level functions.

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)
```

add environment parameter

```
let choose (arg,env) =
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})
```

create closures

```
let f1 (n,env) =
  n + env.xe + env.ye
```

```
let f2 (n,env) =
  n + env.ye
```

use environment variables instead of free variables

```
(choose (true,1,2)) 3
```

```
let c_closure         = (choose, ())                in (* create closure *)
let (c_code, c_cenv) = c_closure                    in (* extract code, env *)
let f_closure         = c_code ((true,1,2), c_env) in (* call choose code, extract f code, env *)
let (f_code, f_env)   = f_closure                   in (* extract code, env *)
f_code (3, f_env)                                      (* call f code *)
```

Closure Conversion

Closure conversion converts open, nested functions in to closed, top-level functions.

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)
```

add environment parameter

```
let choose (arg,env) =
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})
```

create closures

```
let f1 (n,env) =
  n + env.xe + env.ye
```

```
let f2 (n,env) =
  n + env.ye
```

use environment variables instead of free variables

```
(choose (true,1,2)) 3
```

```
let c_closure         = (choose, ())          in (* create closure *)
let (c_code, c_cenv)  = c_closure             in (* extract code, env *)
let f_closure         = c_code ((true,1,2), c_env) in (* call choose code, extract f code, env *)
let (f_code, f_env)   = f_closure             in (* extract code, env *)
f_code (3, f_env)                                (* call f code *)
```

Closure Conversion

Closure conversion converts open, nested functions in to closed, top-level functions.

```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)
```

```
let choose (arg,env) =
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})
```

add environment parameter

create closures

```
let f1 (n,env) =
  n + env.xe + env.ye
```

```
let f2 (n,env) =
  n + env.ye
```

use environment variables instead of free variables

```
(choose (true,1,2)) 3
```

```
let c_closure        = (choose, ())              in (* create closure *)
let (c_code, c_cenv) = c_closure                 in (* extract code, env *)
let f_closure        = c_code ((true,1,2), c_env) in (* call choose code, extract f code, env *)
let (f_code, f_env)  = f_closure                 in (* extract code, env *)
f_code (3, f_env)                                   (* call f code *)
```

Summary: Assignment #4

- In environment-based evaluator, values are drawn from an environment
- In order to implement, nested, higher-order functions, one needs to perform closure conversion, which is the process of implementing functions using a data structure: a pair of code plus an environment that gives values to the (previously) free variables in the code (making that code "closed")
- You have two weeks for assignment #4
  - Why? because last year student found understanding and writing the evaluator pretty tough!
  - Don't wait until next week to start!
  - Put in a full week's worth of work this week