Implementing OCaml in OCaml (Part II)

COS 326
David Walker
Princeton University
Implementing an interpreter:

Components:

• Evaluator for primitive operations
• Substitution
• Recursive evaluation function for expressions
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)
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)
Example to interpret:

\begin{align*}
\text{let } z &= 2 \text{ in } \\
\text{let } z &= 3 + z \text{ in } \\
z
\end{align*}

How to interpret a let expression:

\begin{align*}
\text{eval } (\text{let } x = e_1 \text{ in } e_2) &\rightarrow \\
\text{eval } \{\text{substitute } (\text{eval } e_1) x e_2\}
\end{align*}

How to substitute \( v \) for \( x \) into a let expression

\begin{align*}
\text{substitute } v \ x \ (\text{let } y = e_1 \text{ in } e_2) &= \\
\text{let } y &= \text{substitute } v \ x \ e_1 \text{ in } \\
\text{(if } x = y \text{ then } e_2 \text{ else } \text{substitute } v \ x \ e_2) &\end{align*}
Our Interpreter

Example to interpret:

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

How to interpret a let expression:

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

How to substitute v for x into a let expression

```plaintext
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)
```
Example to interpret:

\[
\text{let } z = 2 \text{ in } \\
\text{let } z = 3 + z \text{ in } \\
z
\]

How to interpret a let expression:

\[
\text{eval } (\text{let } x = e_1 \text{ in } e_2) \rightarrow \\
\text{eval } \{\text{substitute } (\text{eval } e_1) x e_2\}
\]

\[
\text{eval } \{\text{substitute } (\text{eval 2}) z \ (\text{let } z = 3 + z \text{ in } z) \} 
\]

\[
\text{eval } \{\text{substitute } 2 z \ (\text{let } z = 3 + z \text{ in } z) \} 
\]

\[
\text{eval } \{ \text{(let } z = \text{(substitute } 2 z (3 + z)) \text{ in } z) \} 
\]

How to substitute \(v\) for \(x\) into a let expression

\[
\text{substitute } v x (\text{let } y = e_1 \text{ in } e_2) == \\
\text{let } y = \text{substitute } v x e_1 \text{ in } \\
(\text{if } x = y \text{ then } e_2 \text{ else } \text{substitute } v x e_2)
\]
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)
```

notice we don't substitute 2 in z here
Our Interpreter

Example to interpret:

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

How to interpret a let expression:

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

==

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

==

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

==

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

==

```plaintext
eval { let z = 3 + 2 in z }
```

How to substitute v for x into a let expression

```plaintext
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

```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
```
**Scaling up the Language**

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

OCaml’s

```
fun x -> e
```

is represented as

```
Fun_e(x, e)
```
A function call

\[ \text{fact 3} \]

is implemented as

\[ \text{FunCall\_e (Var\_e “fact”, Int\_e 3)} \]
Scaling up the Language

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

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

Easy exam question:
What value does the OCaml interpreter produce when you enter
(fun x -> 3) in to the prompt?
Answer: the value produced is (fun x -> 3)
type exp = Int_e of int | Op_e of exp * op * exp
  | Var_e of variable | Let_e of variable * exp * exp
  | Fun_e of variable * exp | FunCall_e of exp * exp;;

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

Function calls are not values.
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1, eval e2 with
    | Fun_e (x,e), v2 -> eval (substitute v2 x e)
    | _ -> raise TypeError)
let rec eval (e:exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
    | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
    | Var_e x -> raise (UnboundVariable x)
    | Fun_e (x,e) -> Fun_e (x,e)
    | FunCall_e (e1,e2) ->
        (match eval e1, eval e2 with
        | Fun_e (x,e), v2 -> eval (substitute v2 x e)
        | _ -> raise TypeError)
let rec eval (e:exp) : exp =

match e with

| Int_e i -> Int_e i
| Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
| Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
| Var_e x -> raise (UnboundVariable x)
| Fun_e (x,e) -> Fun_e (x,e)
| FunCall_e (e1,e2) ->

  (match eval e1, eval e2 with
   | Fun_e (x,e), v2 -> eval (substitute v2 x e)
   | _ -> raise TypeError)

To evaluate a function call, we first evaluate both e1 and e2 to values.
let rec eval (e:exp) : exp =
match e with
| Int_e i -> Int_e i
| Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
| Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
| Var_e x -> raise (UnboundVariable x)
| Fun_e (x,e) -> Fun_e (x,e)
| FunCall_e (e1,e2) ->
  (match eval e1, eval e2 with
   | Fun_e (x,e), v2 -> eval (substitute v2 x e)
   | _ -> raise TypeError)

e1 had better evaluate to a function value, else we have a type error.
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1, eval e2 with
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)
     | _ -> raise TypeError)

Then we substitute e2’s value (v2) for x in e and evaluate the resulting expression.
Simplifying a little

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

We don’t really need to pattern-match on e2. Just evaluate here.
let rec eval (e:exp) : exp =
match e with
| Int_e i -> Int_e i
| Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
| Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
| Var_e x -> raise (UnboundVariable x)
| Fun_e (x,e) -> Fun_e (x,e)
| FunCall_e (ef,e1) ->
  (match eval ef with
   | Fun_e (x,e2) -> eval (substitute (eval e1) x e2)
   | _ -> raise TypeError)

This looks like the case for let!
Let and Lambda

\[
\text{let } x = 1 \ \text{in } x+41
\]
\[
\rightarrow
\]
\[
1+41
\]
\[
\rightarrow
\]
\[
42
\]

In general:

\[
(\text{fun } x -> x+41) \ 1
\]
\[
\rightarrow
\]
\[
1+41
\]
\[
\rightarrow
\]
\[
42
\]

\[
(\text{fun } x -> e2) \ e1 \quad \equiv \quad \text{let } x = e1 \ \text{in } e2
\]
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

\[
\text{type } \text{exp} = \begin{cases} 
\text{Int}_e \text{ of } \text{int} & | \ 
\text{Op}_e \text{ of } \text{exp} \times \text{op} \times \text{exp} \\
\text{Var}_e \text{ of } \text{variable} & | \\
\text{Let}_e \text{ of } \text{variable} \times \text{exp} \times \text{exp} & | \\
\text{Fun}_e \text{ of } \text{variable} \times \text{exp} & | \\
\text{Rec}_e \text{ of } \text{variable} \times \text{variable} \times \text{exp} 
\end{cases}
\]

\[
\begin{align*}
\text{let rec } & f \ x = f \ (x+1) \ \text{in} \ f \ 3 \\
\text{let } & f = (\text{rec } f \ x \ \to \ f \ (x+1)) \ \text{in} \ f \ 3 \\
\text{let } & g = (\text{rec } f \ x \ \to \ f \ (x+1)) \ \text{in} \ g \ 3
\end{align*}
\]
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
let is_value (e:exp) : bool =
  match e with
  | Int_e _ -> true
  | Fun_e (_,_) -> true
  | Rec_e of (_,_,_) -> true
  | (Op_e (_,_,_) | Let_e (_,_,_,_) | Var_e )
```

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

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

“Substitute value $v$ for variable $x$ in expression $e$:” $e [ v / x ]$

examples of substitution:

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

$$(\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y) [7/y] \quad \text{is} \quad (\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y)$$

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

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

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

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

The Substitution:

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

The Result:

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

Recursive Function Call:

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

The Substitution:

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

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

\[
[ 3 / x ]
\]

Substitute argument for parameter

Substitute entire function for function name

The Result:

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

\rightarrow

\text{if } 3 < 1 \text{ then } 1 \text{ else }

3 \times (\text{rec} \text{ fact } n = \text{if } \ldots \text{ then } \ldots \text{ else } \ldots) \ (3-1)

\rightarrow

3 \times (\text{rec} \text{ fact } n = \text{if } \ldots) \ (3-1)

\rightarrow

3 \times (\text{rec} \text{ fact } n = \text{if } \ldots) \ 2

\rightarrow

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

\rightarrow

3 \times (2 \times (\text{rec} \text{ fact } n = \ldots)(2-1))

\rightarrow

3 \times (2 \times (\text{rec} \text{ fact } n = \ldots)(1))

\rightarrow

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

\rightarrow

3 \times 2 \times 1
A MATHEMATICAL DEFINITION* OF OCAMML EVALUATION

* it’s a partial definition and this is a big topic; for more, see COS 510
OCaml code can give a language semantics

- **advantage**: it can be executed, so we can try it out
- **advantage**: it is amazingly concise
  - especially compared to what you would have written in Java
- **disadvantage**: it is a little ugly to operate over concrete ML datatypes like “Op_e(e1,Plus,e2)” as opposed to “e1 + e2”
PL researchers have developed their own standard notation for writing down how programs execute

– it has a mathematical “feel” that makes PL researchers feel special and gives us *goosebumps* inside

– it operates over abstract expression syntax like “e₁ + e₂”

– it is useful to know this notation if you want to read specifications of programming language semantics

  • e.g.: Standard ML (of which OCaml is a descendent) has a formal definition given in this notation (and C, and Java; but not OCaml…)

  • e.g.: most papers in the conference POPL (ACM Principles of Prog. Lang.)
Our goal is to explain how an expression $e$ evaluates to a value $v$.

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

\[
\begin{array}{cccccc}
\text{premise 1} & \text{premise 2} & \ldots & \text{premise n} \\
\text{conclusion} \\
\end{array}
\]

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”
As a rule:

\[
\begin{align*}
& \text{e1} \rightarrow v_1 \quad \text{e2} \rightarrow v_2 \quad \text{eval}_\text{op} (v_1, \text{op}, v_2) = v' \\
& \text{e1 op e2} \rightarrow v'
\end{align*}
\]

In English:

“If \text{e1} evaluates to \( v_1 \)
and \text{e2} evaluates to \( v_2 \)
and \( \text{eval}_\text{op} (v_1, \text{op}, v_2) \) is equal to \( v' \)
then
\( \text{e1 op e2} \) evaluates to \( v' \)

In code:

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

As a rule:

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

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:

\[ e_1 \rightarrow v_1 \quad e_2 [v_1/x] \rightarrow^* v_2 \]

\[ \text{let } x = e_1 \text{ in } e_2 \rightarrow v_2 \]

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:

```ocaml
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:

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

As a rule:

\[
\begin{align*}
  e_1 \rightarrow \text{rec } f \ x = e & \quad e_2 \rightarrow v \\
  e[\text{rec } f \ x = e/f][v/x] & \rightarrow v_2 \\
  e_1 \; e_2 & \rightarrow v_2
\end{align*}
\]

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 \( \rightarrow \) premise in a rule
- what's the main difference?

complete eval code:

```ocaml
define eval (e:exp) : exp =
  match e with
  | Int e i -> Int e i
  | Op e (op) (e1, e2) -> eval_op (eval e1) op (eval e2)
  | Let e (e1, e2) -> eval (substitute (eval e1) x e2)
  | Var e -> 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 (x, e)) as f_val ->
    let v = eval e2 in
    substitute f_val f (substitute v x e)
```

complete set of rules:

\[
\begin{align*}
\text{i} & \in \mathbb{Z} \\
\text{i} & \rightarrow \text{i} \\
\text{e}_1 & \rightarrow \text{v}_1 \\
\text{e}_2 & \rightarrow \text{v}_2 \\
\text{eval}_\text{op} (\text{v}_1, \text{op}, \text{v}_2) & \rightarrow \text{v} \\
\text{e}_1 \text{ op e}_2 & \rightarrow \text{v} \\
\text{e}_1 \rightarrow \text{v}_1 \\
\text{e}_2 [\text{v}_1/x] & \rightarrow \text{v}_2 \\
\text{let} x = \text{e}_1 \text{ in} \text{e}_2 & \rightarrow \text{v}_2 \\
\lambda x. e & \rightarrow \lambda x. e \\
\text{e}_1 \rightarrow \lambda x.e \\
\text{e}_2 & \rightarrow \text{v}_2 \\
\text{e}[\text{v}_2/x] & \rightarrow \text{v} \\
\text{e}_1 \text{ e}_2 & \rightarrow \text{v} \\
\text{e}_1 \rightarrow \text{rec f x = e} \\
\text{e}_2 & \rightarrow \text{v}_2 \\
\text{e}[\text{rec f x = e/f}[\text{v}_2/x] & \rightarrow \text{v}_3 \\
\text{e}_1 \text{ e}_2 & \rightarrow \text{v}_3
\end{align*}
\]
Comparison: Code vs. Rules

**complete eval code:**

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

**complete set of rules:**

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

- There’s no formal rule for handling free variables
- No rule for evaluating function calls when a non-function in the caller position
- In general, *no rule when further evaluation is impossible*
  - the rules express the *legal evaluations* and say nothing about what to do in error situations
  - the code handles the error situations by raising exceptions
  - type theorists prove that well-typed programs don’t run into undefined cases
Summary

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

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

• We can also specify the evaluation model using a set of inference rules
  – more on this in COS 510
Some Final Words

• The substitution model is only a model.
  – it does not accurately model all of OCaml’s features
    • I/O, exceptions, mutation, concurrency, ...
    • we can build models of these things, but they aren’t as simple.
    • even substitution is tricky to formalize!

• It’s useful for reasoning about higher-order functions, correctness of algorithms, and optimizations.
  – we can use it to formally prove that, for instance:
    • map f (map g xs) == map (comp f g) xs
    • proof: by induction on the length of the list xs, using the definitions of the substitution model.
  – we often model complicated systems (e.g., protocols) using a small functional language and substitution-based evaluation.

• It is not useful for reasoning about execution time or space
  – more complex models needed there
Some Final Words

• The substitution model is only a model.
  – it does not accurately model all of OCaml’s features
    • I/O, exceptions, mutation, concurrency, ...
    • we can build models of these things, but they aren’t as simple.
  • even substitution was tricky to formalize!

• It’s useful for reasoning about higher-order functions, correctness of algorithms, and optimization.
  – we can use it to formally prove that, for instance:
    • map (map g xs) == map (comp f g) xs
  – we can use it to build complicated systems (e.g., a small functional programming version of the Unix shell) using a small functional language and substitution-based evaluation.

• It is not useful for reasoning about execution time or space – more complex models needed there.

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

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

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

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.
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 $\text{Fun}_e(\_,\_)$ is no longer a value
    - a $\text{Fun}_e$ is not a result of a computation
  - There is one more computation step to do:
    - creation of a closure from a $\text{Fun}_e$ expression

Part 2: Prove facts about programs using equational reasoning

- we already saw a bit of equational reasoning today:
  - if $e_1 \rightarrow e_2$ then $e_1 = e_2$
  - more next week
FUNCTION CLOSURES
Consider the following program:

```ocaml
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)
```
Consider the following program:

```ocaml
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:

```ml
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:

```ml
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:

```ocaml
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:

```ocaml
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:

```ml
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:

```ml
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

```ml
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)
```
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)

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

main:
    ...
    jmp choose
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)
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
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)

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)
choose (true, 1, 2)
Substitution and Compiled Code

```plaintext
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)
```

```plaintext
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

```plaintext
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)
```

execute with substitution

```plaintext
if true then
    (fun n -> n + 1)
else
    (fun n -> n + 2)
```
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.
General tactic: Reduce the problem of compiling ML-like functions to the problem of compiling C-like functions.

Some functions are already C-like:

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

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

# argument in r1
# return address in r0
But what about nested, higher-order functions?

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

```ocaml
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:
  ```
  let f2 (n, env) = n + env.y
  ```
  ```
  {y = 1}
  ```
Closure Conversion

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

```ocaml
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 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)

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
```
Closure Conversion

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

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

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

```plaintext
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

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

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

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

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

```ocaml
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

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

- **Closure conversion** converts open, nested functions into closed, top-level functions.
- **Add environment parameter** to capture the context.
- **Create closures** to ensure function closure.
- **Use environment variables** instead of free variables.
- **Extract code and env** to manage function calculations.
- **Call chosen functions** with provided arguments.
Closure conversion converts open, nested functions into closed, top-level functions.

```ocaml
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

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