A Functional Evaluation Model

COS 326
David Walker
Princeton University

A Functional Evaluation Model

In order to be able to write a program, you have to have a solid grasp of how a programming language works.

We often call the definition of "how a programming language works" its *semantics*.

There are many kinds of programming language semantics.

In this class, we will look at O'Caml's *call-by-value* evaluation:

- First, informally, giving *program rewrite rules by example*
- Second, using code, by specifying an *OCaml interpreter* in OCaml
- Third, more formally, using logical *inference rules*

In each case, we are specifying what is known as OCaml's *operational semantics*

O'CAML BASICS:
CORE EXPRESSION EVALUATION

Evaluation

- Execution of an O'Caml expression
  - produces a value
  - and may have some effect (eg: it may raise an exception, print a string, read a file, or store a value in an array)
- A lot of O'Caml expressions have no effect
  - they are pure
  - they produce a value and do nothing more
  - the pure expressions are the easiest kinds of expressions to reason about
- We will focus on evaluation of pure expressions

Evaluation of Pure Expressions

- Given an expression e, we write:

```
e --> v
```

  to state that expression e evaluates to value v

- Note that "e --> v" is not itself a program -- it is some notation that we use talk about how programs work

Evaluation of Pure Expressions

- Given an expression e, we write:

  e --> v

  to state that expression e evaluates to value v
- Some examples:

Evaluation of Pure Expressions

- Given an expression e, we write:

e --> v

to state that expression e evaluates to value v

- Some examples:

1 + 2

Evaluation of Pure Expressions

- Given an expression e, we write:

e --> v

to state that expression e evaluates to value v

- Some examples:

1 + 2 --> 3

Evaluation of Pure Expressions

- Given an expression e, we write:

e --> v

to state that expression e evaluates to value v

- Some examples:

1 + 2 --> 3

2

Evaluation of Pure Expressions

- Given an expression e, we write:

  e --> v

  to state that expression e evaluates to value v
- Some examples:

  1 + 2 --> 3

  2 --> 2

  values step to values

Evaluation of Pure Expressions

- Given an expression e, we write:

e --> v

to state that expression e evaluates to value v

- Some examples:

1 + 2 --> 3

2 --> 2

int_to_string 5 --> "5"

Evaluation of Pure Expressions

More generally, we say expression e (partly) evaluates to expression e':

$$e \dashrightarrow e'$$

Evaluation of Pure Expressions

More generally, we say expression e (partly) evaluates to expression e':

e --> e'

Evaluation is *complete* when e' is a value

- In general, I'll use the letter "v" to represent an arbitrary value
- The letter "e" represents an arbitrary expression
- Concrete numbers, strings, characters, etc. are all values, as are:
  - tuples, where the fields are values
  - records, where the fields are values
  - datatype constructors applied to a value
  - *functions*

Evaluation of Pure Expressions

- Some expressions (all the interesting ones!) take many steps to evaluate them:

```
(2 * 3) + (7 * 5)
```

Evaluation of Pure Expressions

- Some expressions (all the interesting ones!) take many steps to evaluate them:

```
(2 * 3) + (7 * 5)
--> 6 + (7 * 5)
```

Evaluation of Pure Expressions

- Some expressions (all the interesting ones!) take many steps to evaluate them:

```
(2 * 3) + (7 * 5)
--> 6 + (7 * 5)
--> 6 + 35
```

Evaluation of Pure Expressions

- Some expressions (all the interesting ones!) take many steps to evaluate them:

```
(2 * 3) + (7 * 5)
--> 6 + (7 * 5)
--> 6 + 35
--> 41
```

Evaluation of Pure Expressions

- Some expressions do not compute a value and it is not obvious how to proceed:

"hello" + 1 --> no value!

- A *strongly typed language rules out a lot of nonsensical expressions* that compute no value, like the one above
- Other expressions compute no value but raise an exception:

7 / 0 --> raise Divide_by_zero

- Still others simply fail to terminate …

Let Expressions: Evaluate using Substitution

```
let x = 30 in
let y = 12 in
x+y
```

-->

```
let y = 12 in
30+y
```

-->

```
30+12
```

-->

```
42
```

Informal Evaluation Model

To evaluate a function call "f a"

- first evaluate f until we get a function value (fun x -> e)
- then evaluate a until we get an argument value v
- then substitute v for x in e, the function body
- then evaluate the resulting expression.

this is why we say O'Caml is "call by value"

```
let inc = (fun x -> x+1) in
inc 41                                -->

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

41+1                                  -->

42
```

Informal Evaluation Model

Another example:

```
let add x y = x+y in
let inc = add 1 in
let dec = add -1 in
dec(inc 42)
```

Informal Evaluation Model

Recall the syntactic sugar:

```
let add = fun x -> (fun y -> x+y) in
let inc = add 1 in
let dec = add -1 in
dec(inc 42)
```

Informal Evaluation Model

Then we use the let rule – we substitute the *value* for add:

```
let add = fun x -> (fun y -> x+y) in
let inc = add 1 in
let dec = add -1 in
dec(inc 42)

-->

let inc = (fun x -> (fun y -> x+y)) 1 in
let dec = (fun x -> (fun y -> x+y)) -1 in
dec(inc 42)
```

functions are values

Informal Evaluation Model

not a value; must reduce before substituting for inc

```
let inc = (fun x -> (fun y -> x+y)) 1 in
let dec = (fun x -> (fun y -> x+y)) -1 in
dec(inc 42)

-->

let inc = fun y -> 1+y in
let dec = (fun x -> (fun y -> x+y)) -1 in
dec(inc 42)
```

Informal Evaluation Model

now a value

```
let inc = fun y -> 1+y in
let dec = (fun x -> (fun y -> x+y)) -1 in
dec(inc 42)

-->

let dec = (fun x -> (fun y -> x+y)) -1 in
dec((fun y -> 1+y) 42)
```

Informal Evaluation Model

Next: simplify dec's definition using the function-call rule.

```
let dec = (fun x -> (fun y -> x+y)) -1 in
dec((fun y -> 1+y) 42)

-->

let dec = fun y -> -1+y in
dec((fun y -> 1+y) 42)
```

now a value

Informal Evaluation Model

And we can use the let-rule now to substitute dec:

```
let dec = fun y -> -1+y in
dec((fun y -> 1+y) 42)          -->


(fun y -> -1+y) ((fun y -> 1+y) 42)
```

Informal Evaluation Model

Now we can't yet apply the first function because the argument is not yet a value – it's a function call. So we need to use the function-call rule to simplify it to a value:

```
(fun y -> -1+y) ((fun y -> 1+y) 42) -->

(fun y -> -1+y) (1+42) -->

(fun y -> -1+y) 43 -->

-1+43 -->

42
```

Variable Renaming

Consider the following Ocaml code:

```
let x = 30 in
let y = 12 in
x+y;;
```

Does this evaluate any differently than the following?

```
let a = 30 in
let b = 12 in
a+b;;
```

Renaming

A basic principle of programs is that systematically changing the names of variables shouldn't cause the program to behave any differently – it should evaluate to the same thing.

```
let x = 30 in
let y = 12 in
x+y;;
```

But we do have to be careful about *systematic* change.

```
let a = 30 in
let a = 12 in
a+a;;
```

Systematic change of variable names is called *alpha-conversion*.

Substitution

Wait a minute, how do we evaluate this using the let-rule? If we substitute 30 for “a” naively, then we get:

```
let a = 30 in
let a = 12 in
a+a
```

-->

```
let 30 = 12 in
30+30
```

Which makes no sense at all!

Besides, Ocaml returns 24 not 60.

What went wrong with our informal model?

Scope and Modularity

- Lexically scoped (a.k.a. statically scoped) variables have a simple rule: the nearest enclosing “let” in the code defines the variable.
- So when we write:

```
let a = 30 in
let a = 12 in
a+a;;
```

- we know that the “a+a” corresponds to “12+12” as opposed to “30+30” or even weirder “30+12”.

A Revised Let-Rule:

- To evaluate "**let** x = e_1 **in** e_2":
  - First, evaluate e_1 to a value v.
  - Then substitute v for the *corresponding uses* of x in e_2.
  - Then evaluate the resulting expression.

```
let a = 30 in
let a = 12 in
a+a
```

This "a" doesn't correspond to the uses of "a" below.

-->

```
let a = 12 in
a+a
```

So when we substitute 30 for it, it doesn't change anything.

-->

```
12+12
```

-->

```
24
```

Scope and Modularity

- But what does “corresponding uses” mean?

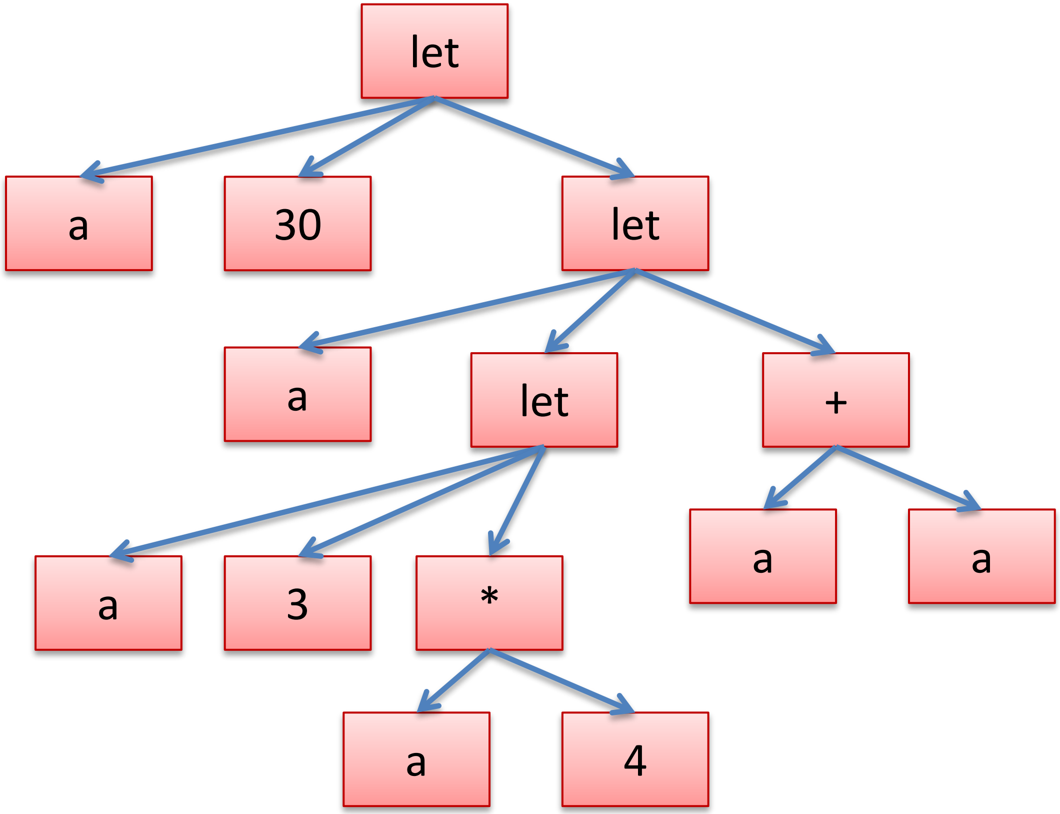
- Consider:

```
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```

Abstract Syntax Trees

- We can view a program as a tree – the parentheses and precedence rules of the language help determine the structure of the tree.

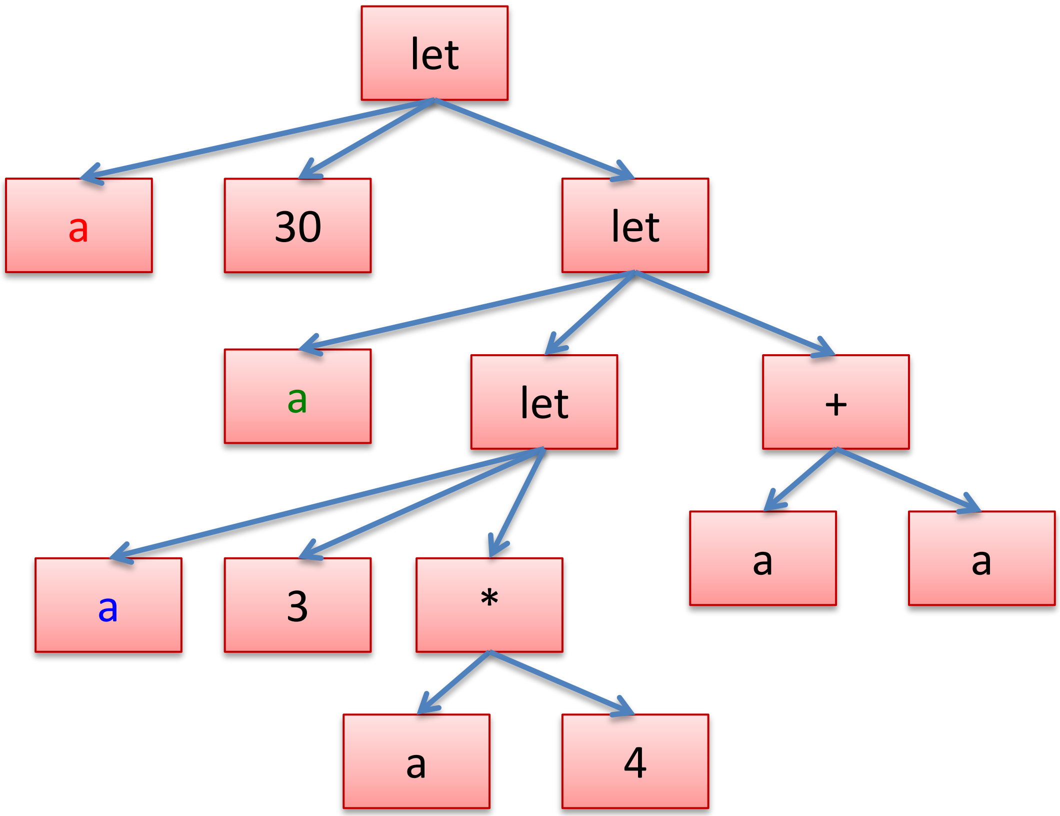
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;

==

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



Binding Occurrences

- An occurrence of a variable where we are defining it via let is said to be a *binding occurrence* of the variable.

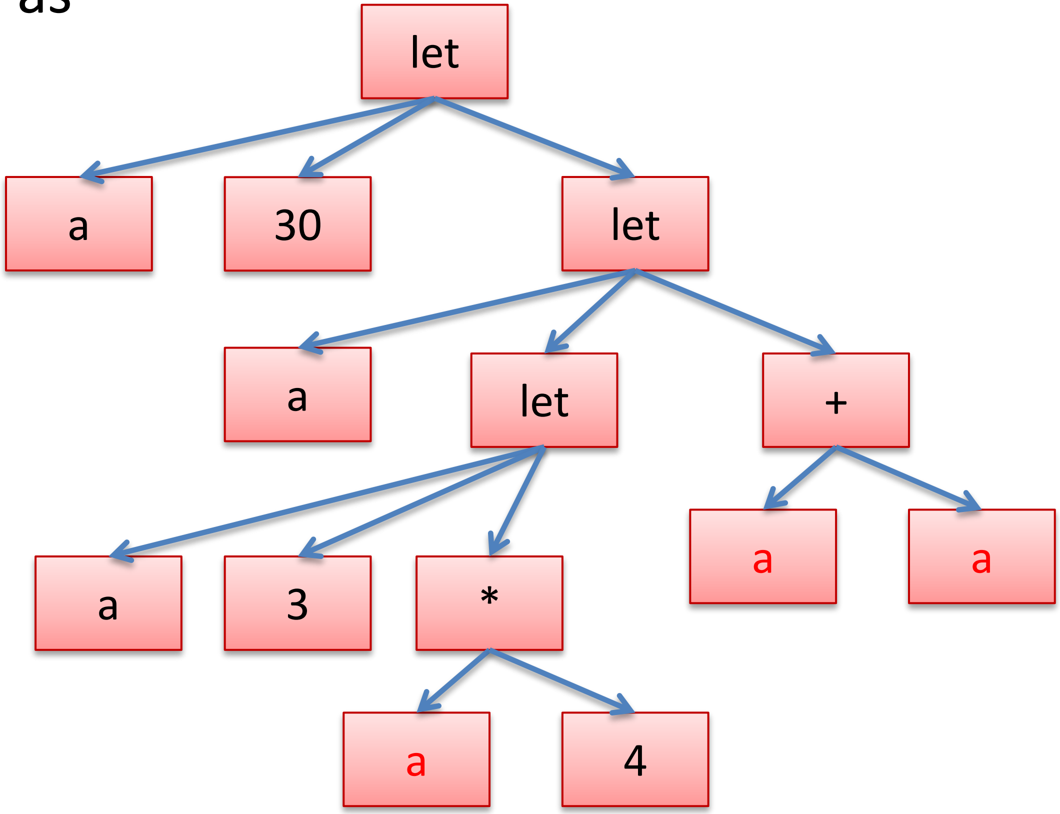
```
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```



Free Occurrences

- A non-binding occurrence of a variable is said to be a *free variable*.
- That is a *use* of a variable as opposed to a definition.

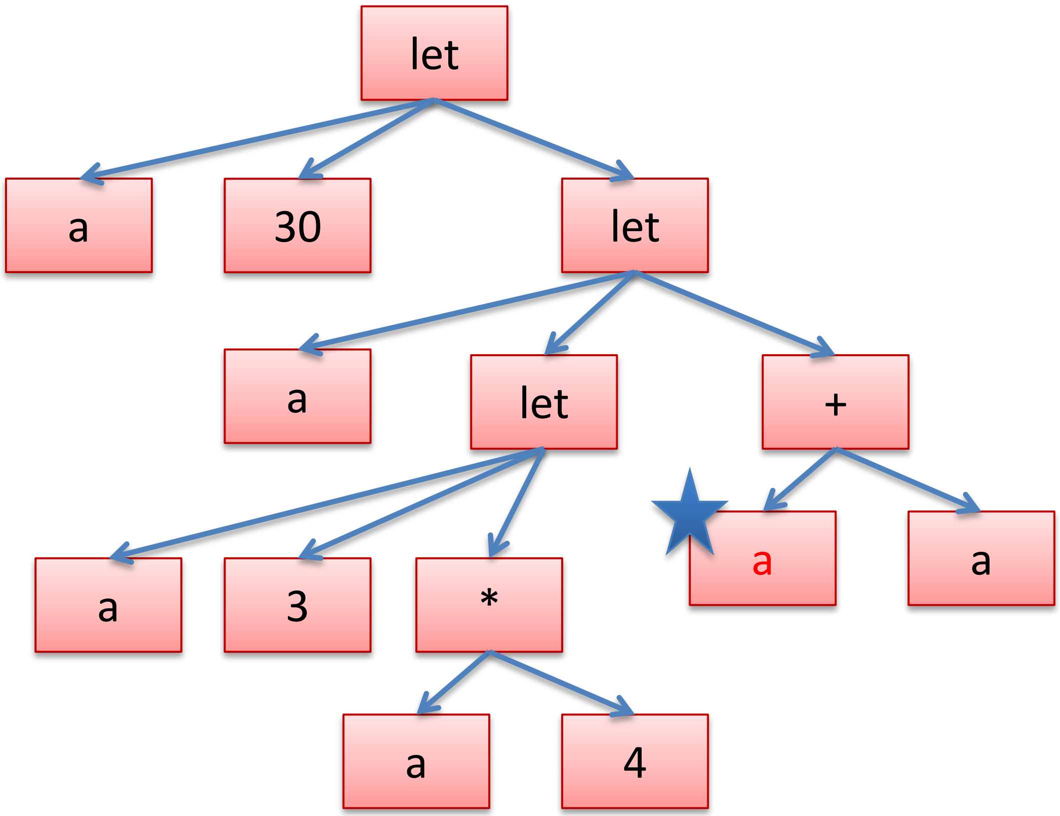
```
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```



Abstract Syntax Trees

- Given a free variable occurrence, we can find where it is bound by …

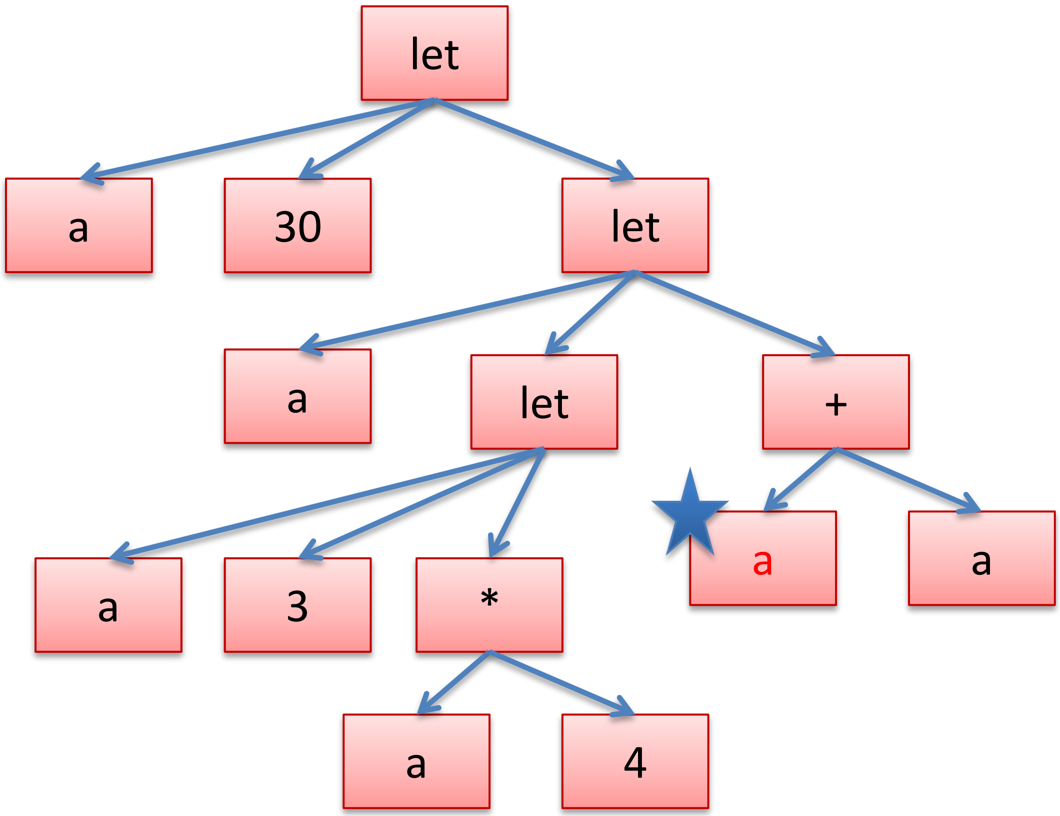
```
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```



Abstract Syntax Trees

- crawling up the tree to the nearest enclosing let…

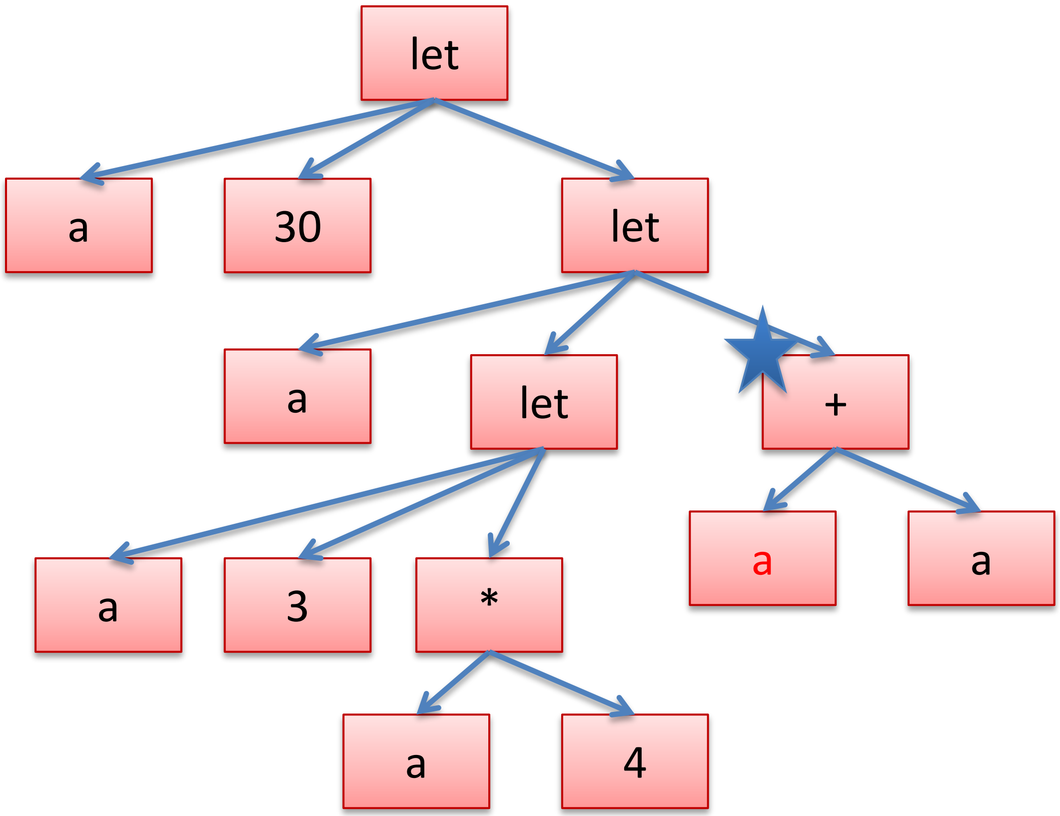
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- crawling up the tree to the nearest enclosing let…

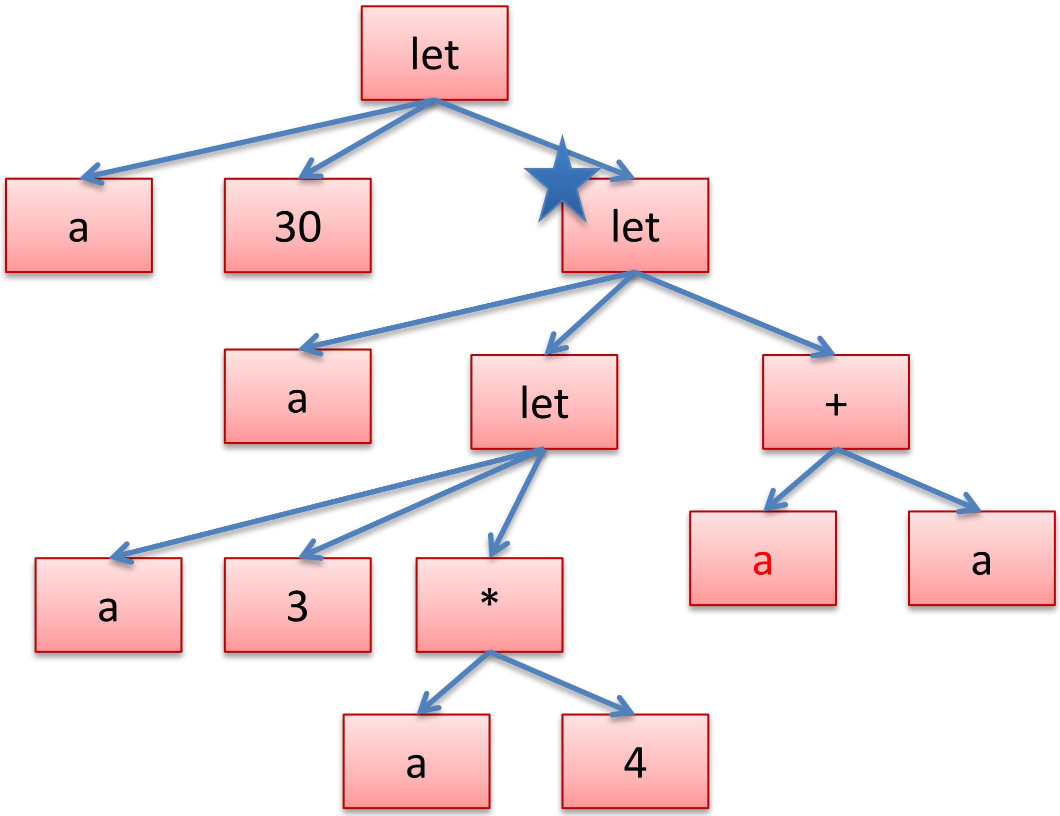
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- crawling up the tree to the nearest enclosing let…

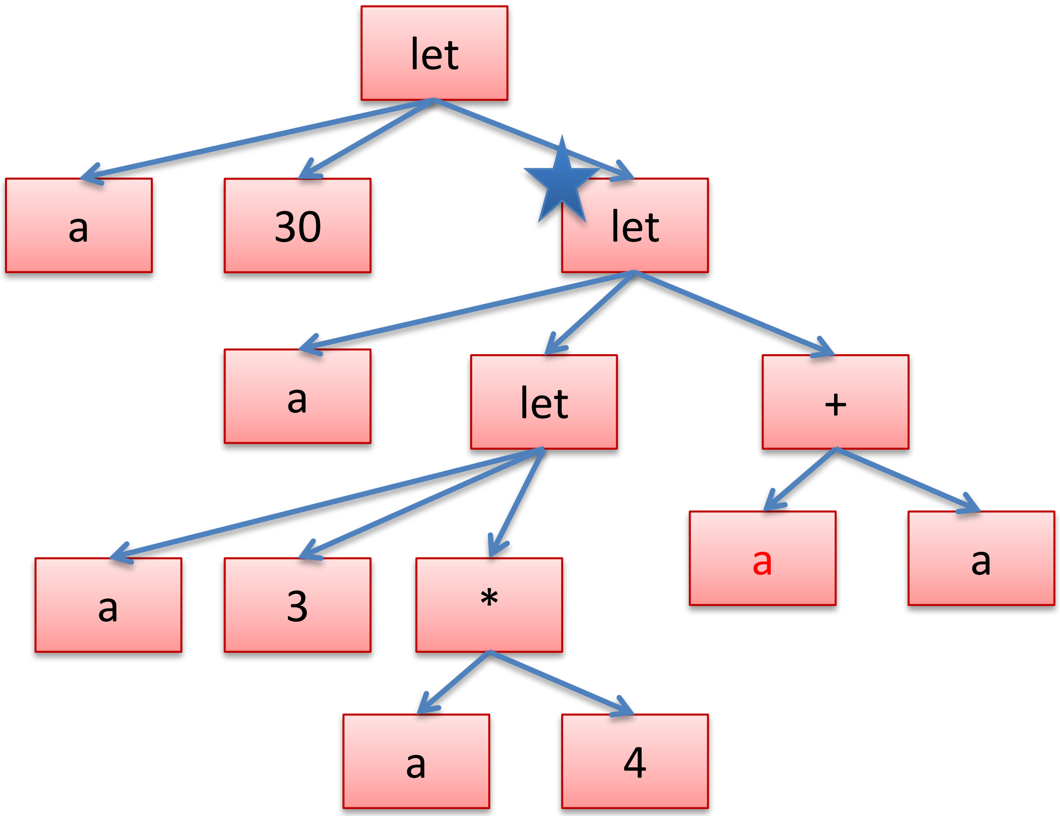
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- and see if the “let” binds the variable – if so, we’ve found the nearest enclosing definition. If not, we keep going up.

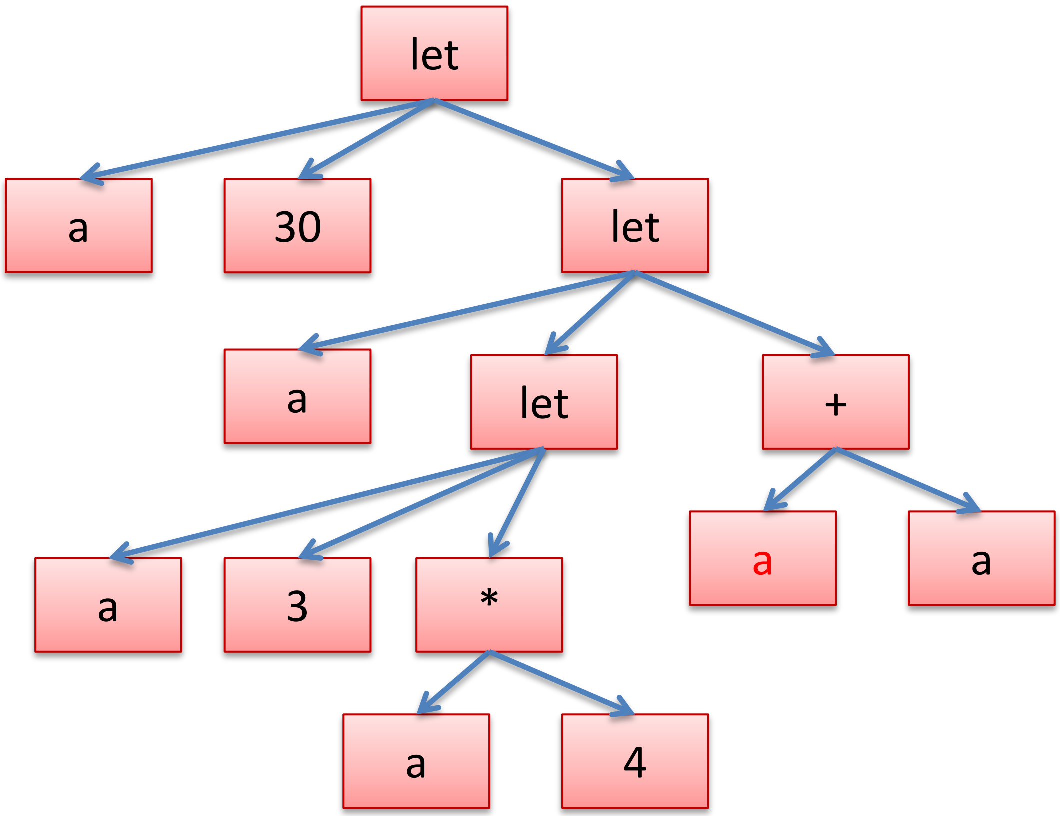
```
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```



Abstract Syntax Trees

- Now we can also systematically rename the variables so that it's not so confusing. Systematic renaming is called *alpha-conversion*

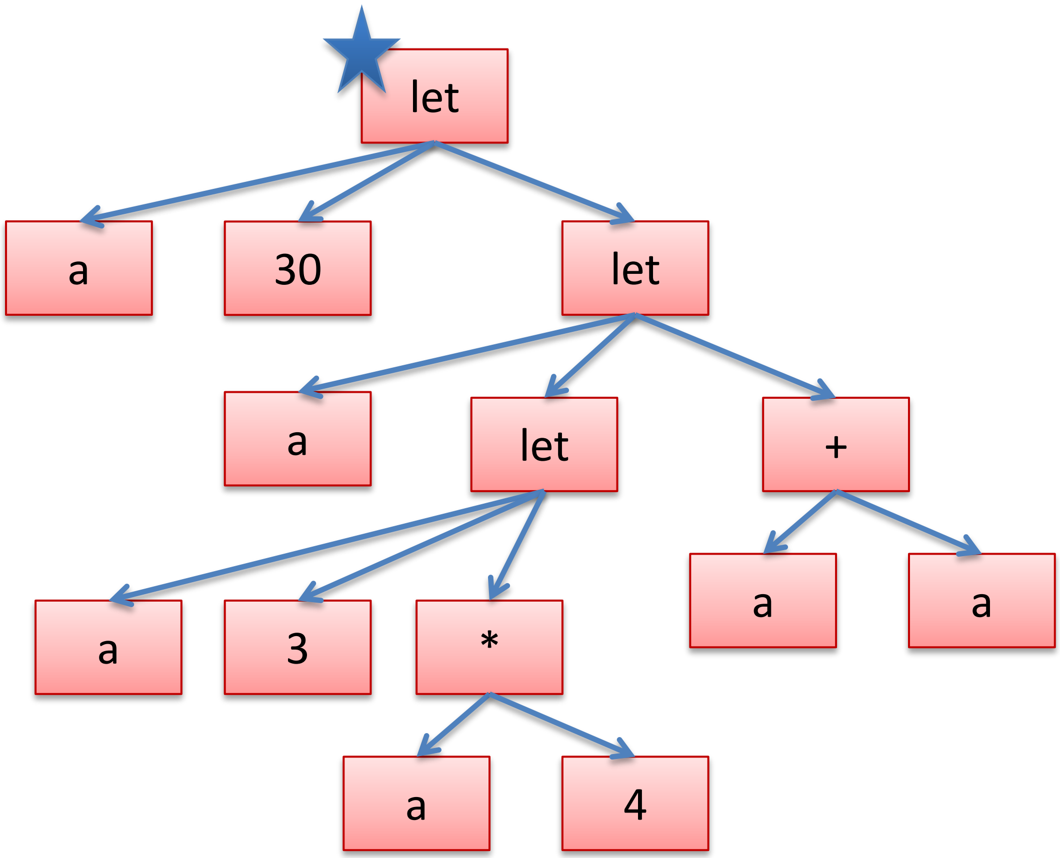
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- Start with a let, and pick a fresh variable name, say “x”

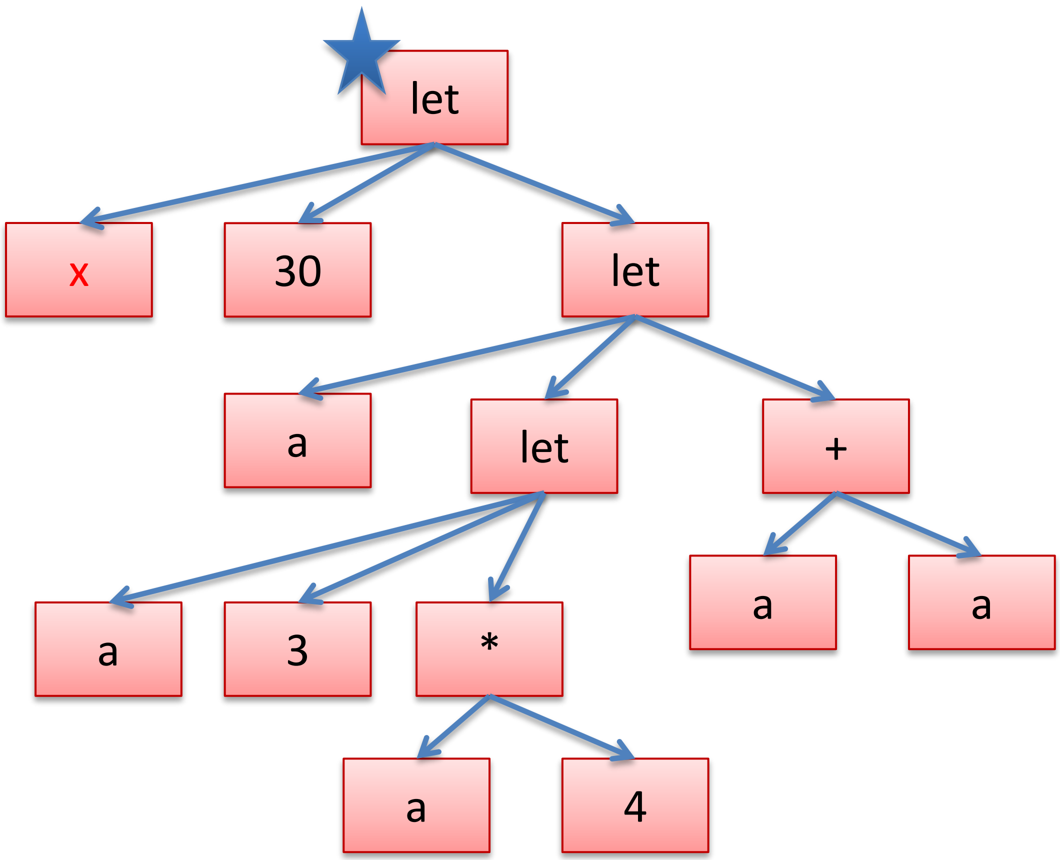
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- Rename the binding occurrence from “a” to “x”.

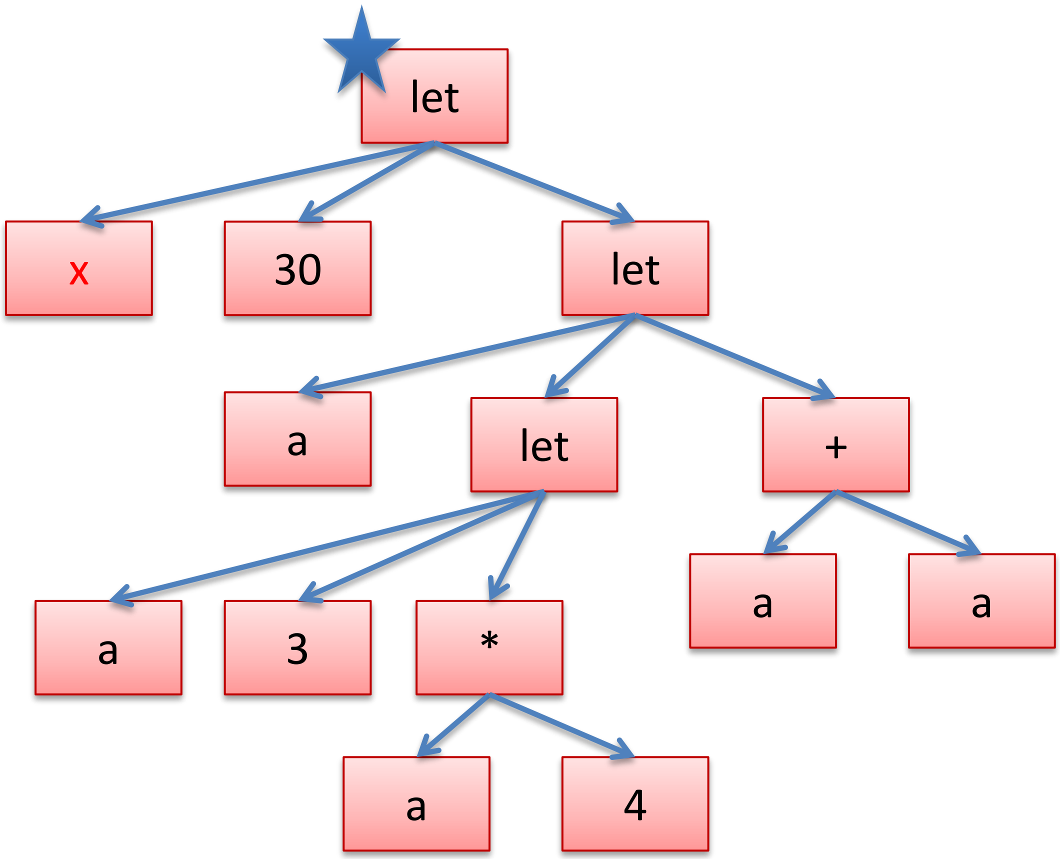
```
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```



Abstract Syntax Trees

- Then rename all of the free occurrences of the variables that this let binds.

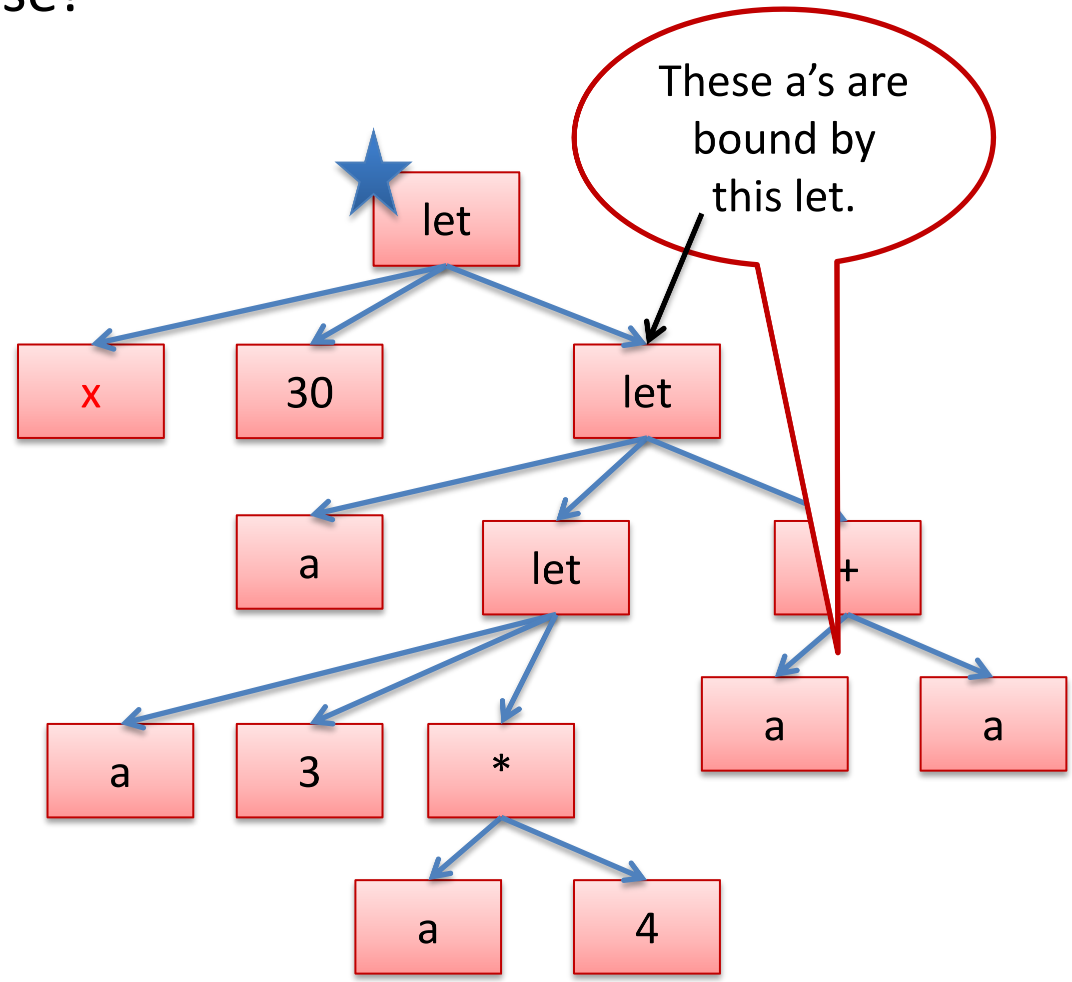
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- There are none in this case!

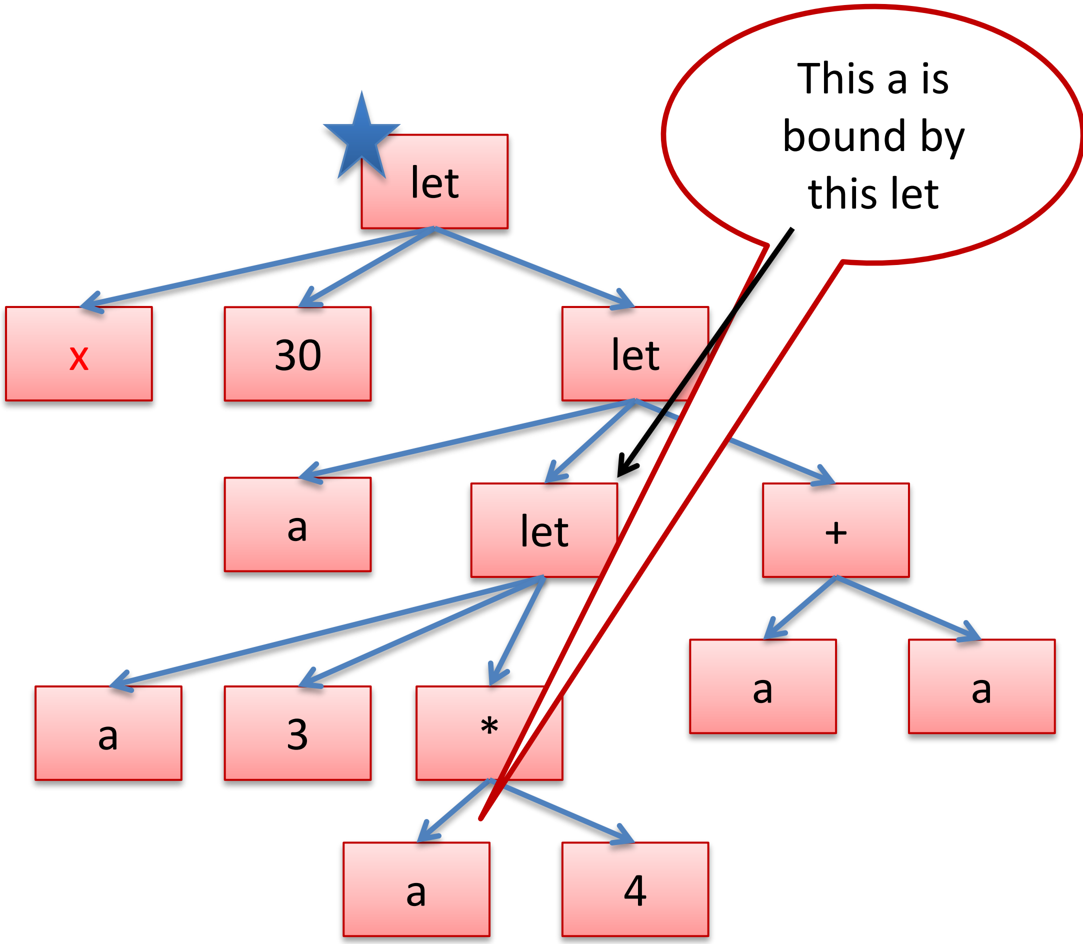
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- There are none in this case!

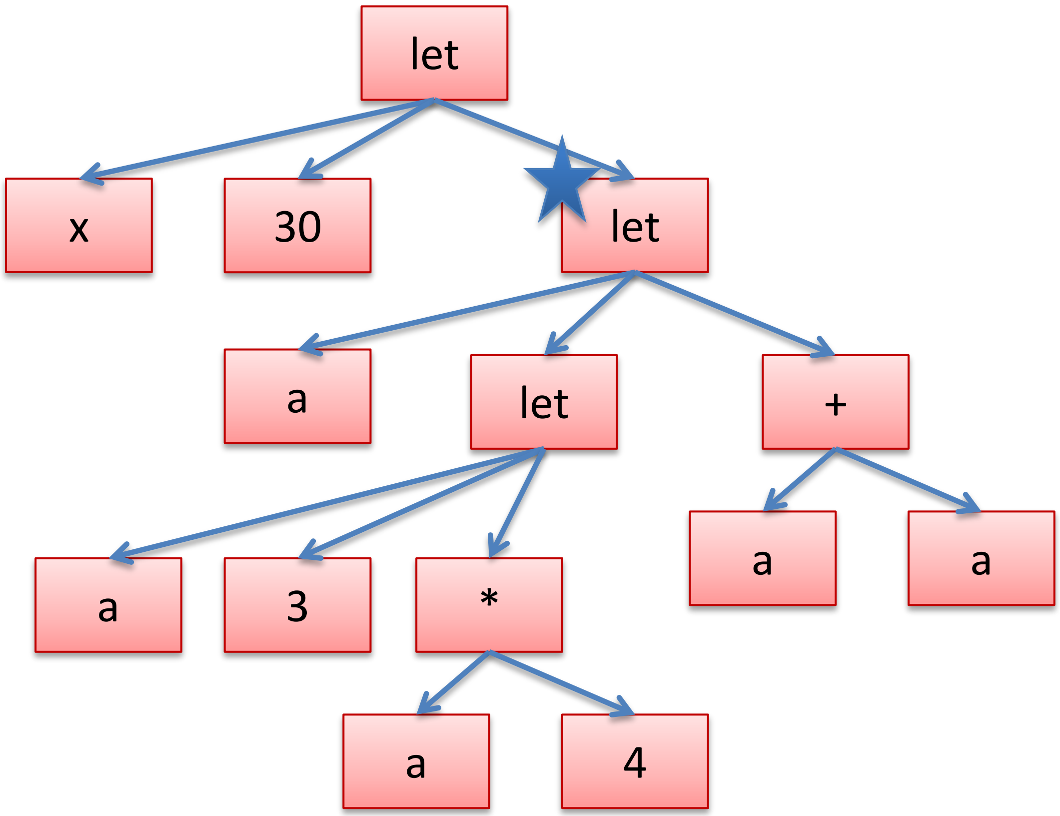
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

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

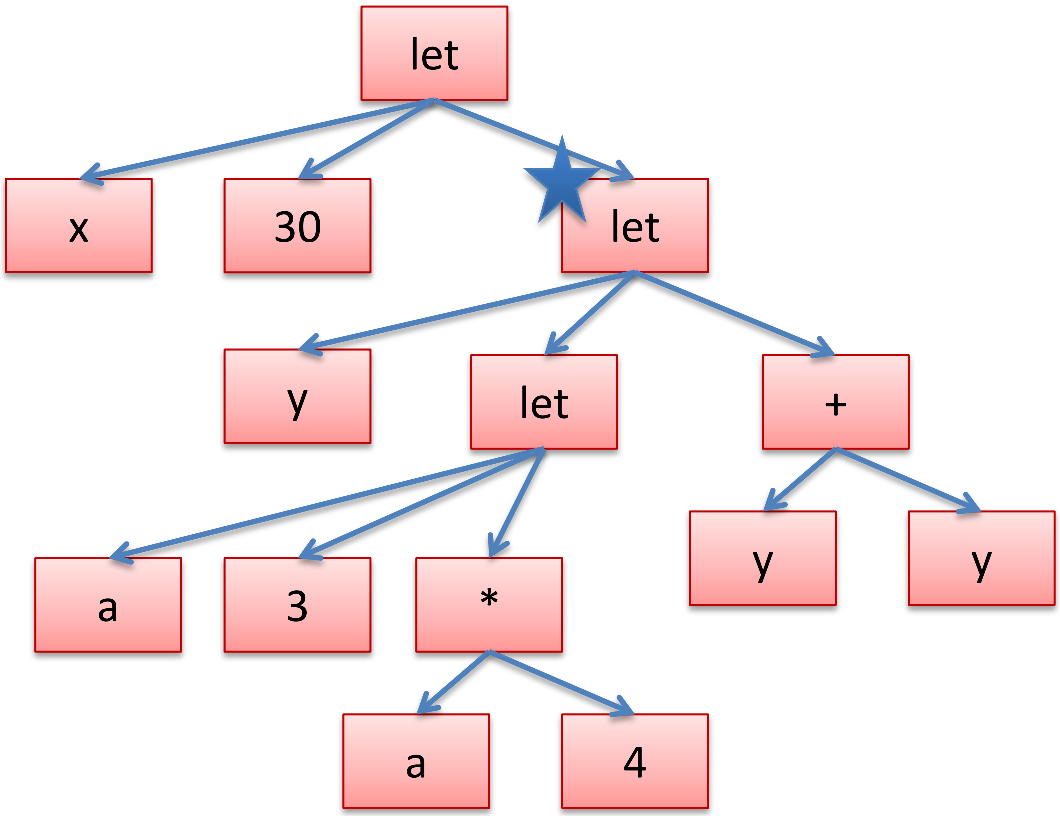
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;



Abstract Syntax Trees

- Let's do another let, renaming "a" to "y".

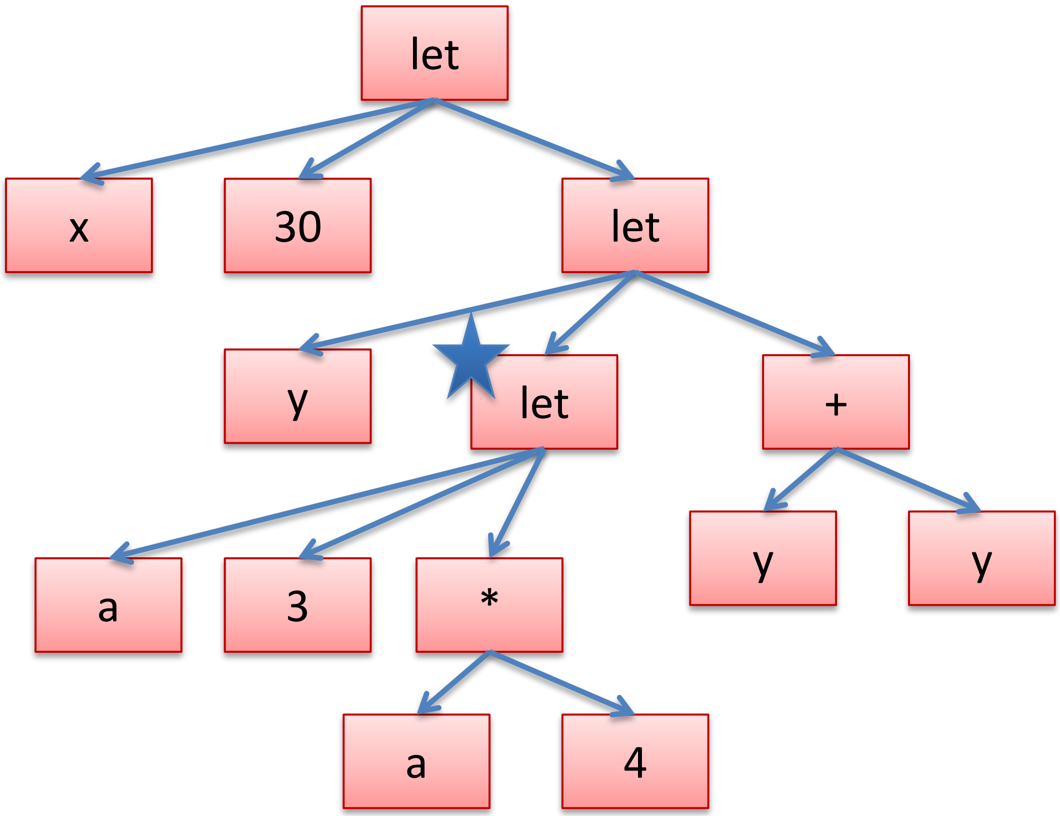
let x = 30 in
let y = (let a = 3 in a*4) in
y+y;;



Abstract Syntax Trees

- And if we rename the other let to “z”:

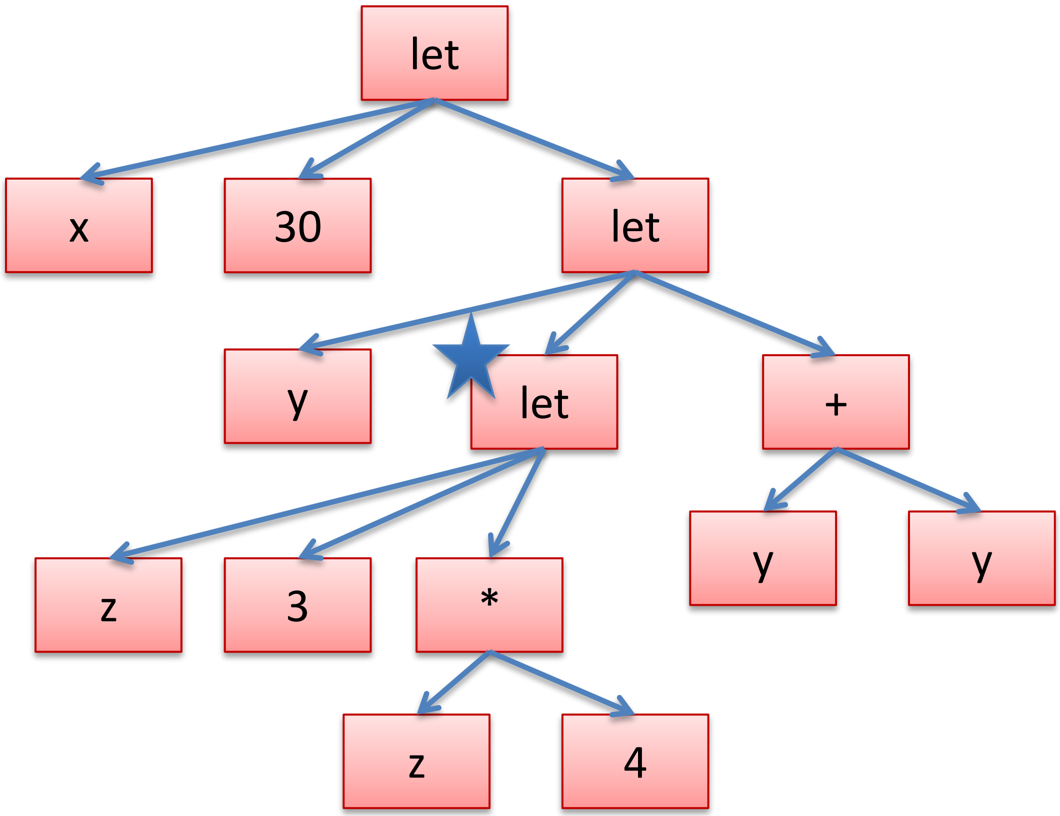
let x = 30 in
let y = (let a = 3 in a*4) in
y+y;;



Abstract Syntax Trees

- And if we rename the other let to "z":

```
let x = 30 in
let y = (let z = 3 in z*4) in
y+y;;
```



AN O'CAML DEFINITION OF O'CAML EVALUATION

Making These Ideas Precise

- We can define a datatype for Ocaml expressions

```
type variable = string ;;
type operand = 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 ;;
```

Making These Ideas Precise

- We can define a datatype for Ocaml expressions

```
type variable = string ;;
type operand = 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 ;;

let three = Int_e 3 ;;
let three_plus_one =
      Op_e (Int_e 1, Plus, Int_e 3) ;;
```

Making These Ideas Precise

We can represent the Ocaml program:

```
let x = 30 in
let y =
   (let z = 3 in
    z*4)
in
y+y;;
```

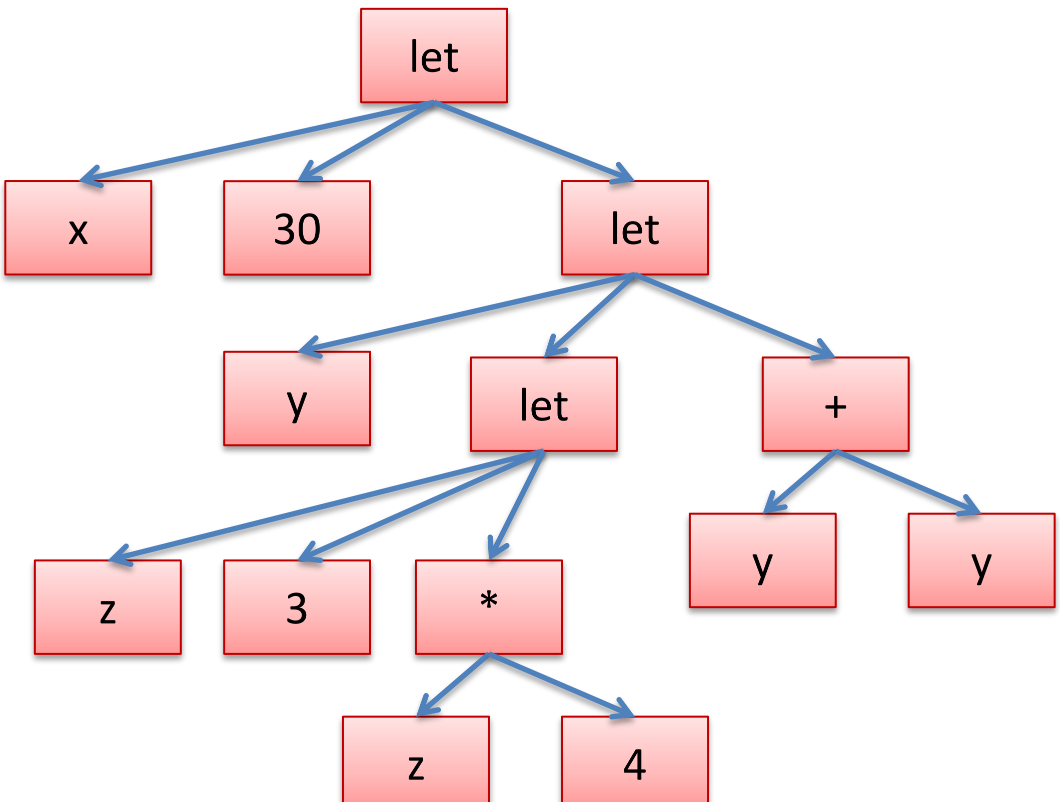
as an exp value:

```
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”)
```

Making These Ideas Precise

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")
```



Free versus Bound Variables

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

This is a free occurrence of a variable

Free versus Bound Variables

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

This is a free occurrence of a variable

This is a binding occurrence of a variable

A Simple Evaluator

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

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) ->
        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 e = substitute v1 x e2 in
     eval e
```

Even Simpler

```
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)
```

Oops! We Missed a Case:

```
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 Use Options:

```
let eval_op v1 op v2 = …
let substitute v x e = …

let rec eval (e:exp) : exp option =
  match e with
  | Int_e i -> Some(Int_e i)
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> None
```

But this isn't quite right – we need to match on the recursive calls to eval to make sure we get Some value!

Exceptions

```
exception UnboundVariable of variable ;;

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

Instead, we can throw an exception.

Exceptions

```
exception UnboundVariable of variable ;;

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

Note that an exception declaration is a lot like a datatype declaration. Really, we are extending one big datatype (exn) with a new constructor (UnboundVariable).

Exceptions

```
exception UnboundVariable of variable ;;

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

Later on, we'll see how to catch an exception.

Back to our Evaluator

```
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 -> raise (UnboundVariable x) ;;
```

Evaluating the Primitive Operations

```
let eval_op (v1:exp) (op:operand) (v2:exp) : exp =
  match v1, op, v2 with
  | Int_e i, Plus, Int_e j -> Int_e (i+j)
  | Int_e i, Minus, Int_e j -> Int_e (i-j)
  | Int_e i, Times, Int_e j -> Int_e (i*j)
  …;;

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 -> raise (UnboundVariable x) ;;
```

Substitution

```
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
;;

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

```
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
  in
  subst 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 -> raise (UnboundVariable x) ;;
```

We want to replace x (and only x) with v.

Substitution

```
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 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 -> raise (UnboundVariable x) ;;
```

If x and y are the same variable, then y *shadows* x.

Let us Scale 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 ;;
```

Let us Scale 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 ;;
```

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

Let us Scale 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 ;;
```

A function call (e.g., fact 3) is represented using a FunCall_e

Let us Scale 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) : boo
  match e with
  | Int_e
```

fact 3 is represented as:
FunCall_e(Var_e “fact”, Int_e 3)

append x y is the same as (append x) y
and represented as:
FunCall_e (
    FunCall_e (Var_e “append”, Var_e “x”),
    Var_e “y”)

Let us Scale 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!

Let us Scale 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 us Scale 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)
```

Let us Scale 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.

Let us Scale 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.

Let us Scale 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.

Let us Scale 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
       | Fun_e (x,e) -> eval (substitute (eval e2) x e)
       | _ -> raise TypeError)
```

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
       | Fun_e (x,e) -> eval (substitute (eval e2) x e)
       | _ -> 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

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 (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)
```

Alternatively:

```
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_e (Fun_e(x,e2)) e1)
  | 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)
```

Or we can turn let's into function calls!

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
  | LetRec_e of variable * exp * exp ;;
```

Example:

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

→ (rewrite)

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

→ (implement)

```
LetRec_e (“f”,
  Fun_e (“x”,
     FunCall_e (Var_e “f”,
       Op_e (Var_e “x”, Plus, Int_e 1)
     )
  ),
  FunCall (Var_e “f”, 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
  | LetRec_e of variable * 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 (_,_) | LetRec_e of (_,_,_)) -> false ;;
```

Evaluating Letrec

Start out with a recursive let:

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

Unwind/unroll the recursion one time:

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

Replace the recursive let with a normal let:

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

Evaluate the normal let definition:

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

Evaluating Letrec

```
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) ->
      let e1_unwound =
             substitute (LetRec_e (x,e1,Var x)) x e1 in
       eval (Let_e (x, e1_unwound, e2))
```

Evaluating Letrec

```
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) ->
      let e1_unwound =
             substitute (LetRec_e (x,e1,Var x)) x e1 in
        eval (Let_e (x, e1_unwound, e2))
```

We're unrolling the recursive function once by substituting LetRec_e (x,e1,var x) for x.

Evaluating Letrec

```
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) ->
      let e1_unwound =
             substitute (LetRec_e (x,e1,Var x)) x e1 in
        eval (Let_e (x, e1_unwound, e2))
```

Once it's unwound, it's no longer recursive. So we can evaluate it like a normal let.

Another Example of Unwinding Letrec

```
let rec fact n = if n < 1 then 1 else n * fact(n-1) in
fact 3
==
let rec fact =
   fun n -> if n < 1 then 1 else n * fact(n-1) in
fact 3
==>
let fact =
   fun n -> if n < 1 then 1 else n *
     (let rec fact = fun n -> if … in fact) (n-1) in
fact 3
==>
if 3 < 1 then 1 else 3 *
  (let rec fact = fun n -> if … in fact) (3-1)
```

A MATHEMATICAL DEFINITION* OF O'CAML EVALUATION

* it's a partial definition and this is a big topic; for more, see COS 441

From Code to Abstract Specification

- the O'Caml code is one kind of specification of a language
  - 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, relatively 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
    - eg: Standard ML (of which O'Caml is a descent) has a formal definition given in this notation

Notation for Substitution

- Programming languages are defined using substitution so often that PL researchers have a special notation for it
  - (actually, we have like 6 notations for it; I'll just give you one)

- To substitute, value v for free variable x in expression e:

e [v/x]

- Examples:

(x + y) [7/y]	is	(x + y)
(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)

Rules

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

- We are going to do so using a set of (inductive) rules
- 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 concerning evaluation

As a rule:

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

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) -> eval_op (eval e1) op (eval e2)
```

An example rule concerning evaluation

As a rule:

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

asserts i is an integer

In English:

"If the expression is an integer, 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:

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

In English:

> "If e1 evaluates to v1
> 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) -> eval (substitute (eval e1) x e2)
 ...
```

An example rule concerning evaluation

As a rule:

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

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

In English:

"A function 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
          | Fun_e (x,e) -> eval (Let_e (x,e2,e))
          | _ -> raise TypeError)
...
```

An example rule concerning evaluation

As a rule:

$$\frac{eu == e1[letrec\ x = e1\ in\ x\ /\ x] \qquad let\ x = eu\ in\ e2\ \text{-->}\ v}{letrec\ x = e1\ in\ e2\ \text{-->}\ v}$$

In English:

"uggh"

In code:

```
let rec eval (e:exp) : exp =
  match e with
 ..
 | LetRec_e (x,e1,e2) ->
           let e1_unwound =
                  substitute (LetRec_e (x,e1,Var x)) x e1 in
           eval (Let_e (x, e1_unwound, e2))
```

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) ->
      let e1_unwound =
            substitute (LetRec_e (x,e1,Var x)) x e1 in
       eval (Let_e (x, e1_unwound, e2))
```

complete set of rules:

$$\frac{i \in \mathbb{Z}}{i \ \texttt{-->}\ i}$$

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

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

$$\frac{}{\lambda x.e \ \texttt{-->}\ \lambda x.e}$$

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

$$\frac{eu == e1[\text{letrec } x = e1 \text{ in } x \ / \ x] \qquad \text{let } x = eu \text{ in } e2 \ \texttt{-->}\ v}{\text{letrec } x = e1 \text{ in } e2 \ \texttt{-->}\ v}$$

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) ->
      let e1_unwound =
            substitute (LetRec_e (x,e1,Var x)) x e1 in
       eval (Let_e (x, e1_unwound, e2))
```

complete set of rules:

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

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

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

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

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

$$\frac{eu == e1[\text{letrec } x = e1 \text{ in } x\ /\ x] \qquad \text{let } x = eu \text{ in } e2 \dashrightarrow v}{\text{letrec } x = e1 \text{ in } e2 \dashrightarrow v}$$

- There's no 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

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
  - let-rec rule: "let rec x = e1 in e2" : unwind e1 once, then evaluate using the same rule as let.
    - To unwind: substitute (let rec x = e1 in x) for x in e1
- Substitution is tricky
  - follow the rule of lexical scope
  - substitute for only those free occurrences that correspond to the bound variable. (i.e., respect shadowing)
- 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 441

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 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 program execution time or space

Some Exercises

Complete the following expressions so they evaluate to 42 or explain why this is impossible, appealing to the substitution model.

```
let x = ??? in
let x = 43 in
x ;;
```

```
let x = ??? in
let y = (let x = 21 in x+x) in
x ;;
```

```
let x = ??? in
let y = [42] in
x y ;;
```

```
let x = fun x -> x*2 in
let x = ??? 21 in
x ;;
```

END