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 lecture, 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 OCaml 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 OCaml 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 to 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 --> 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 -->   ????
```

- 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

This must be already a value

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

-->

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

-->

```
30 + 12
```

-->

```
42
```

Let Expressions: Evaluate using Substitution

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

This must be already a value

otherwise, first evaluate inside the bound expression

```
let x = 15*2 in
let y = 12 in
x+y
```

-->

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

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 f = (fun x -> x + 1) in f) (30+11)
```

-->

```
(fun x -> x + 1) (30 + 11)
```

-->

```
(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

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

not a value; must reduce before substituting for inc

```
-->
```

```
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;;
```

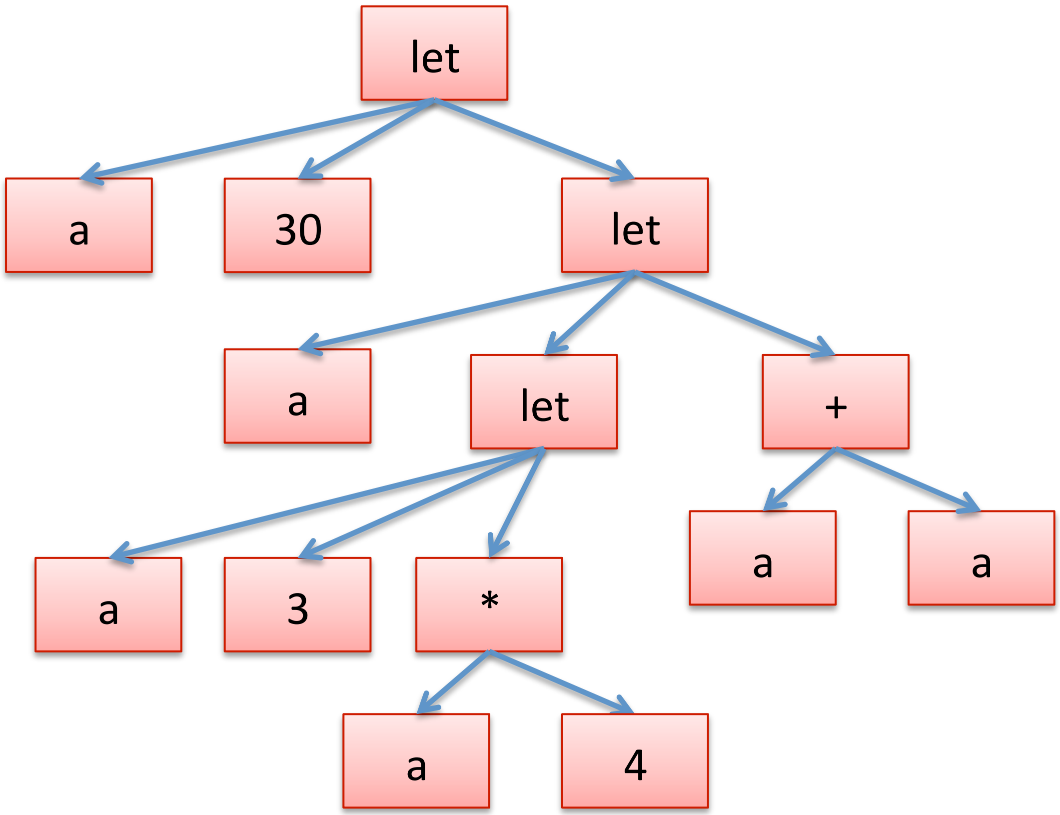
Abstract Syntax Trees

- 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;;
```

==

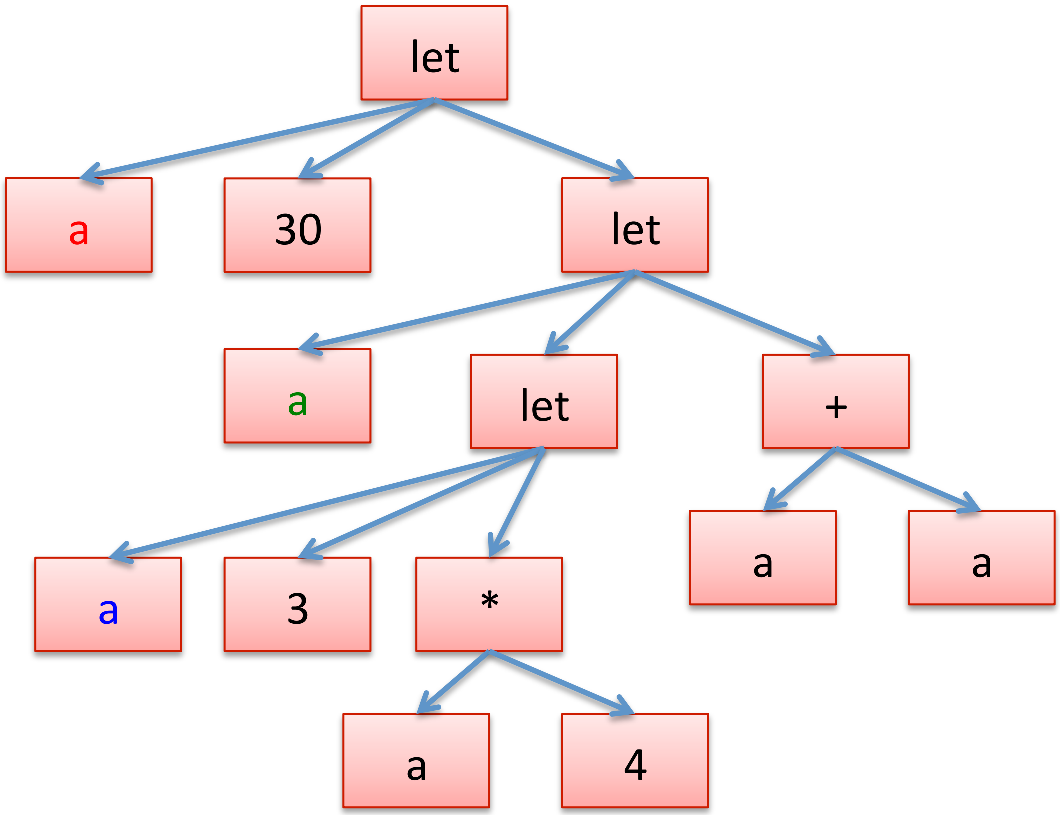
```
(let a = (30) in
 (let a =
    (let a = (3) in (a*4))
  in
  (a+a)))
```



Binding Occurrences

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

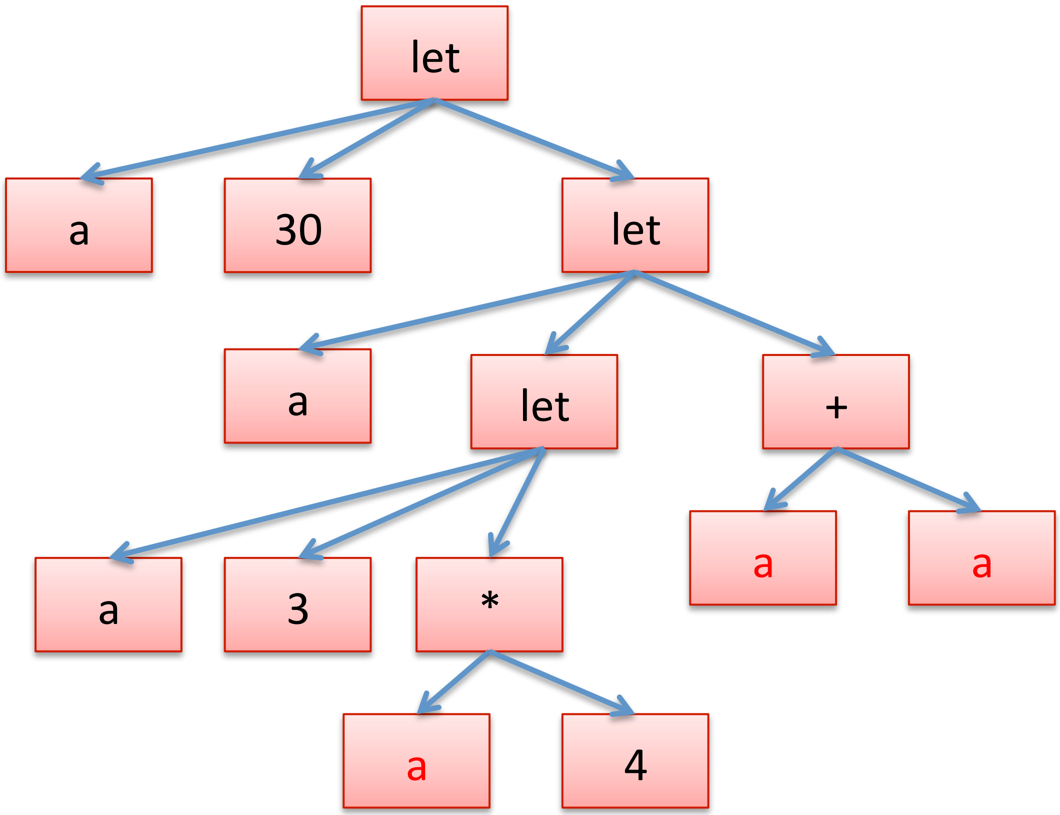
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Free Occurrences

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

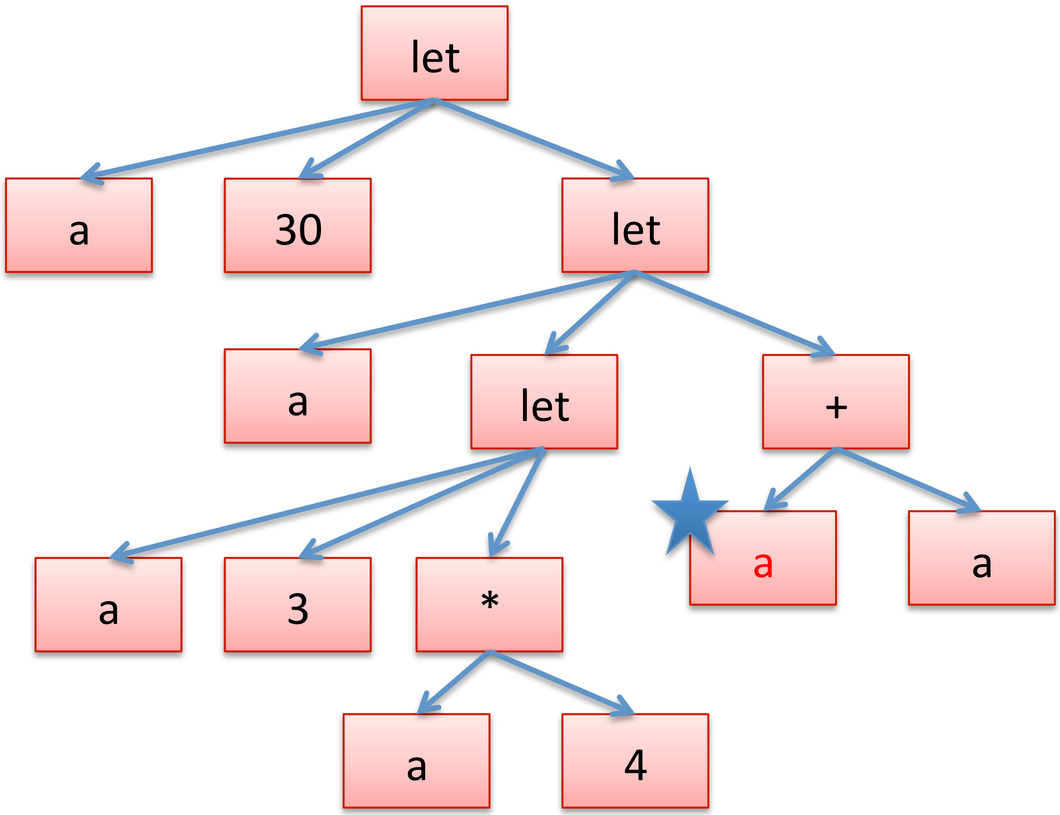
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

Given a variable occurrence, we can find where it is bound by ...

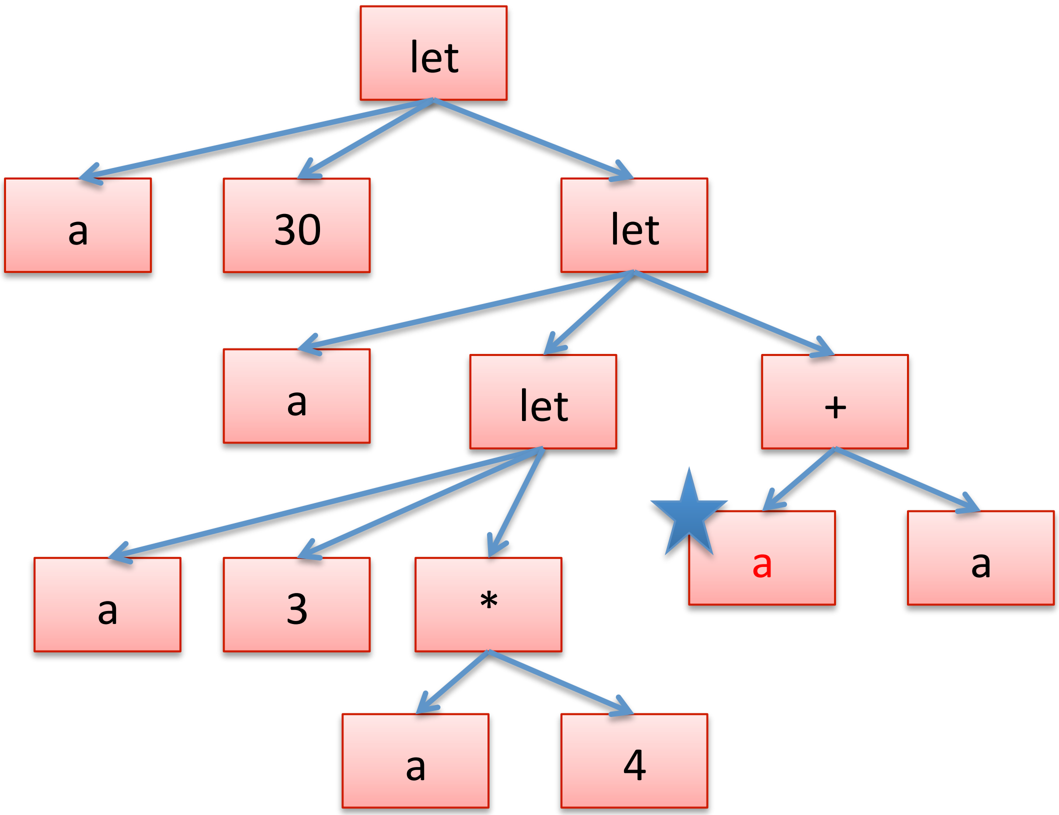
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

crawling up the tree to the nearest enclosing let…

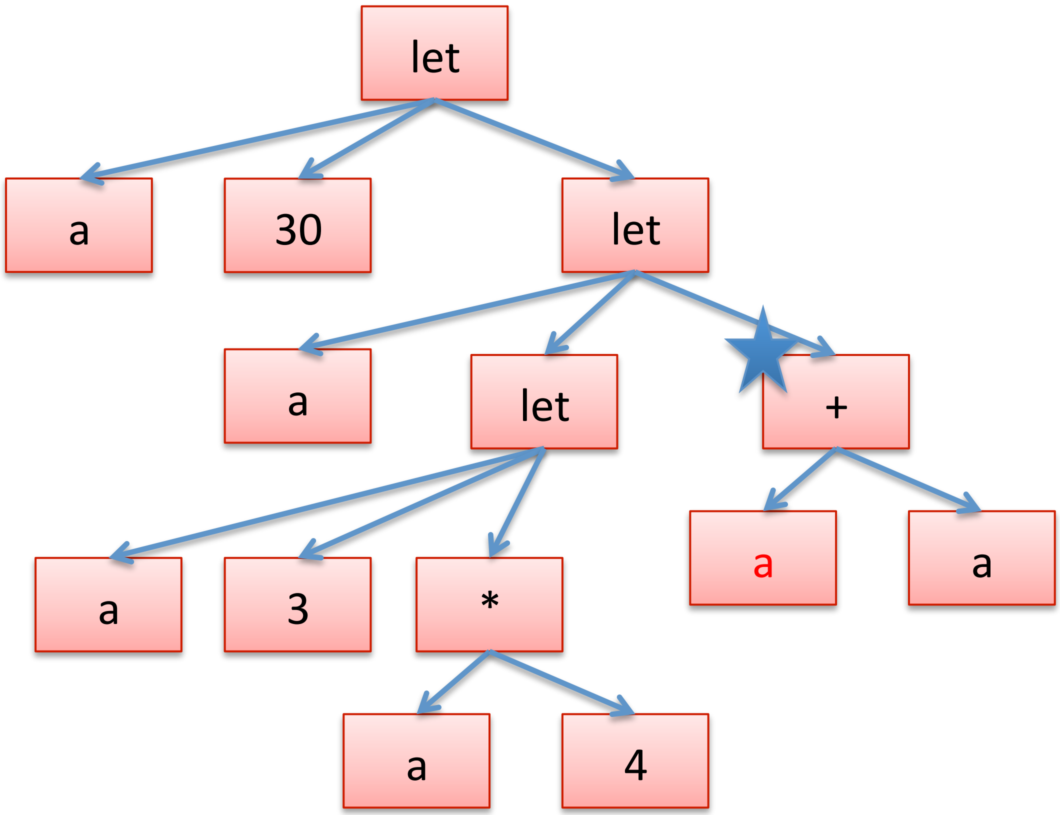
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

crawling up the tree to the nearest enclosing let…

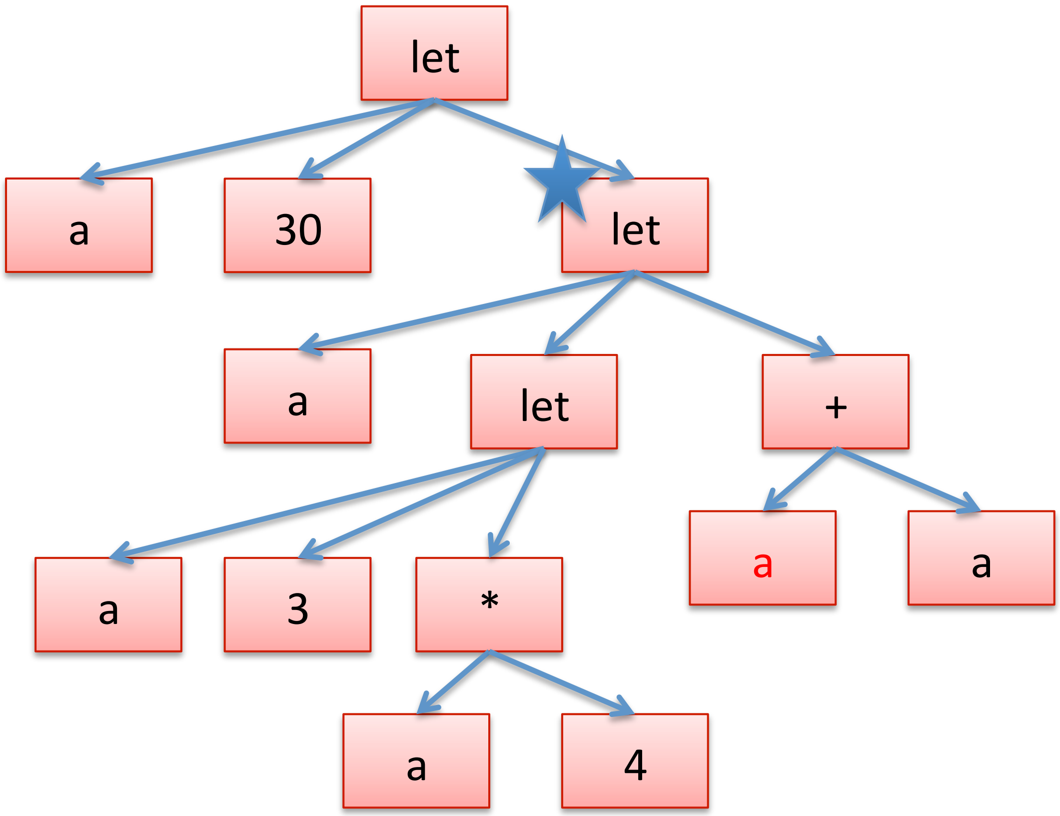
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

crawling up the tree to the nearest enclosing let…

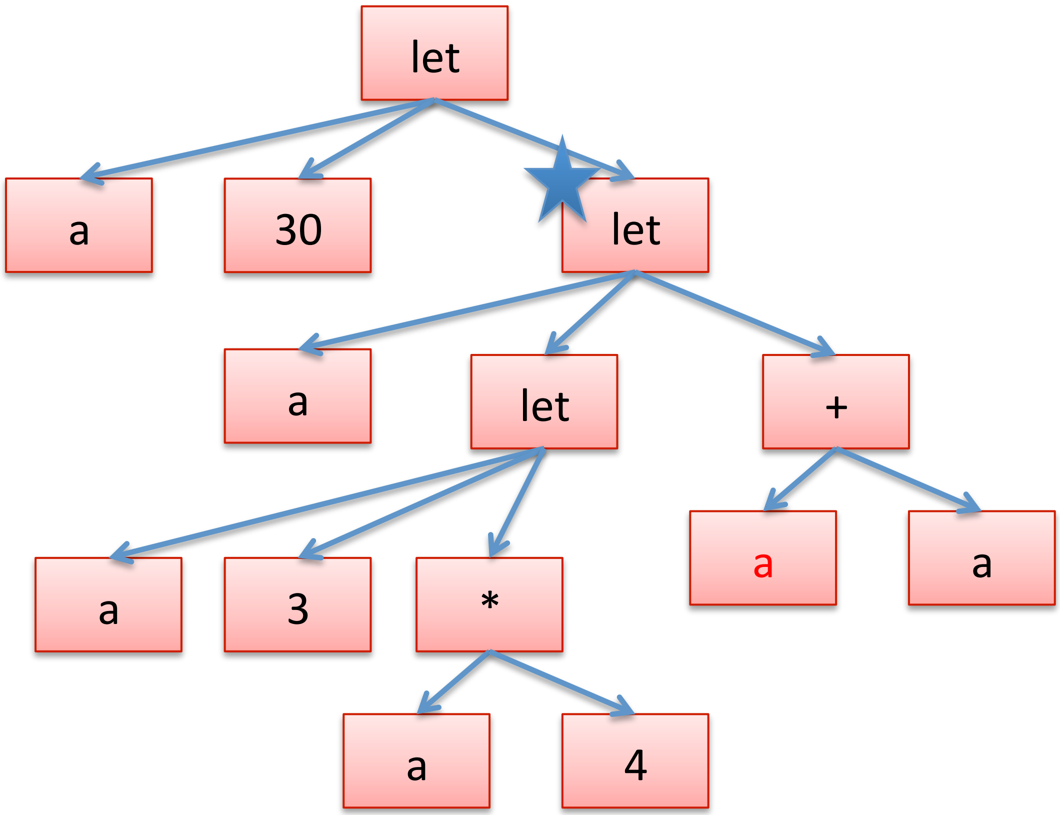
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

and checking if the "let" binds the variable – if so, we've found the nearest enclosing definition. If not, we keep going up.

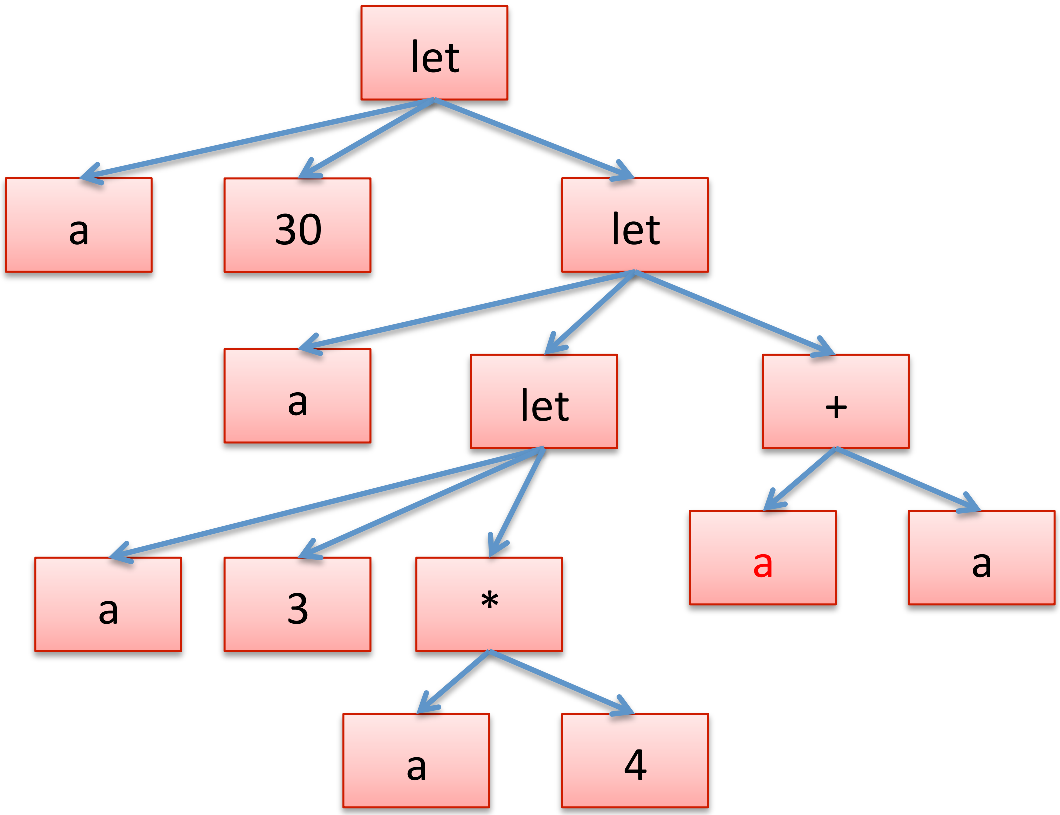
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

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

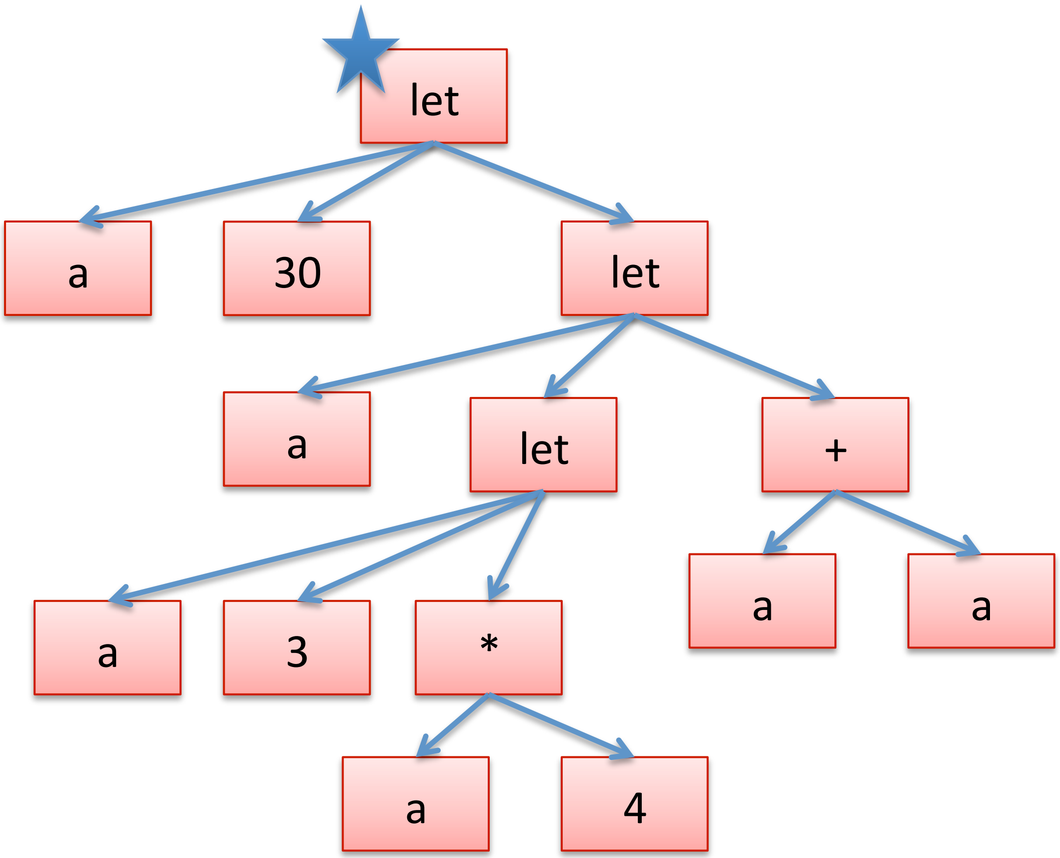
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

Start with a let, and pick a fresh variable name, say "x"

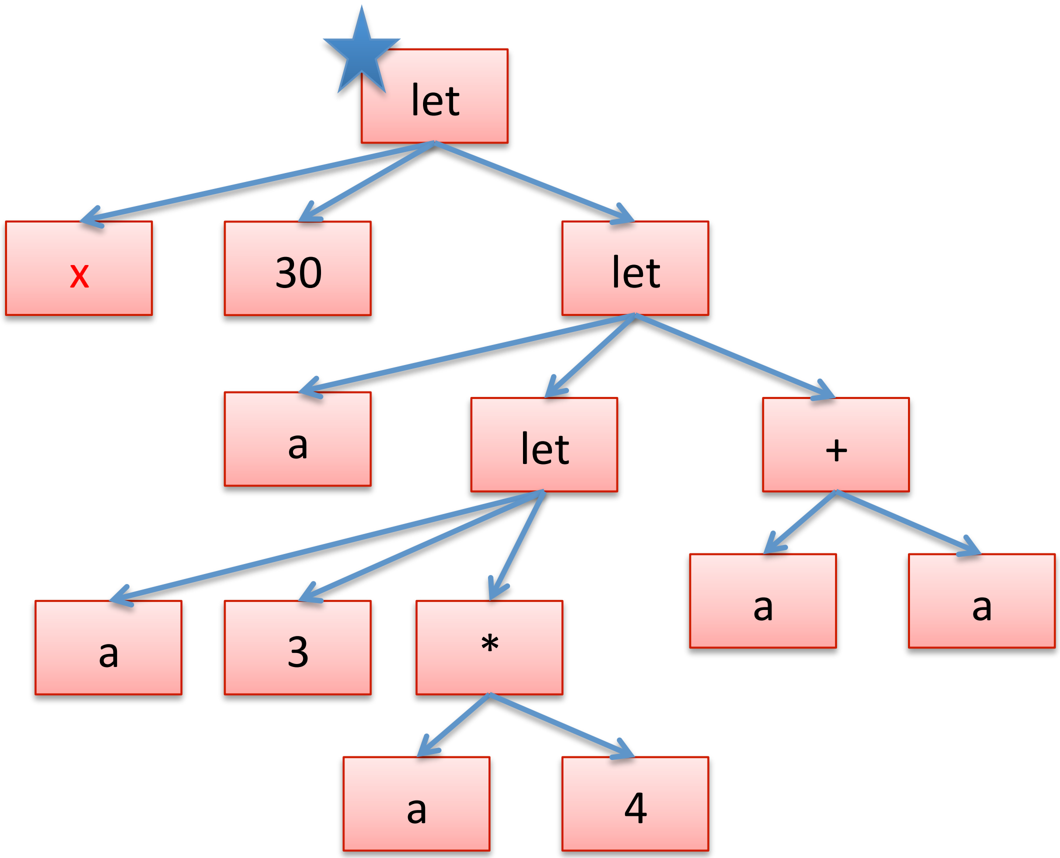
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

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

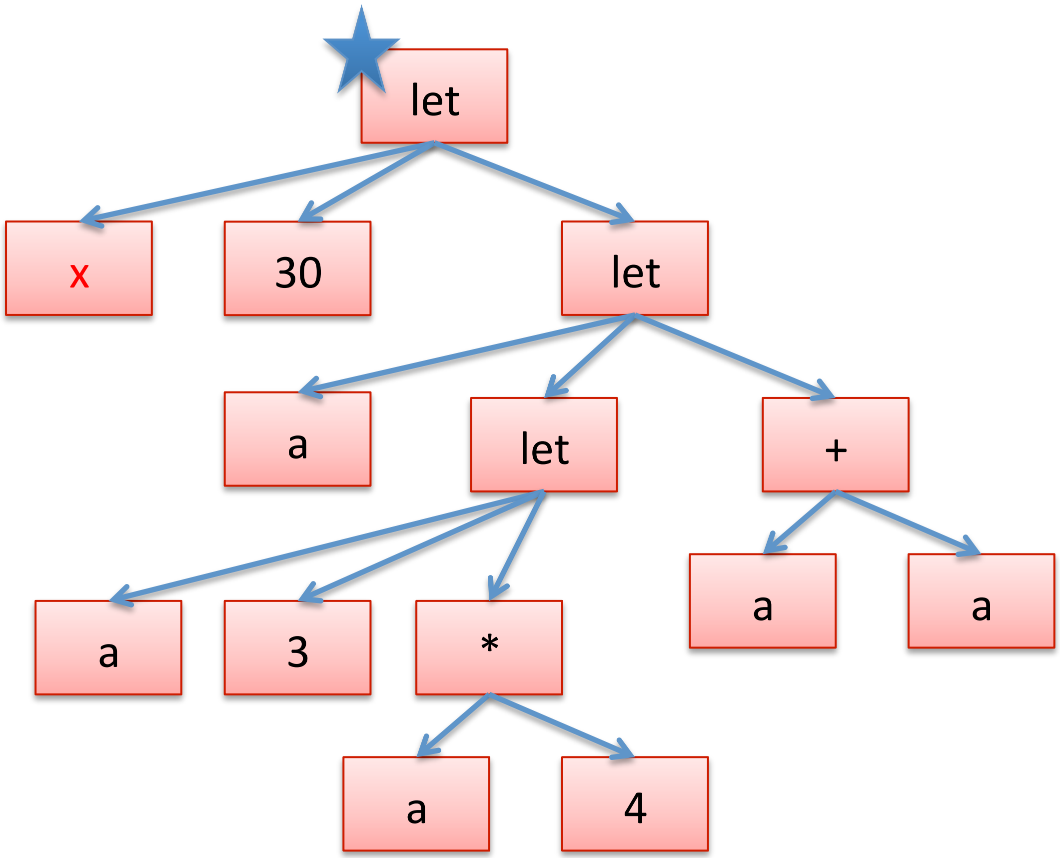
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

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

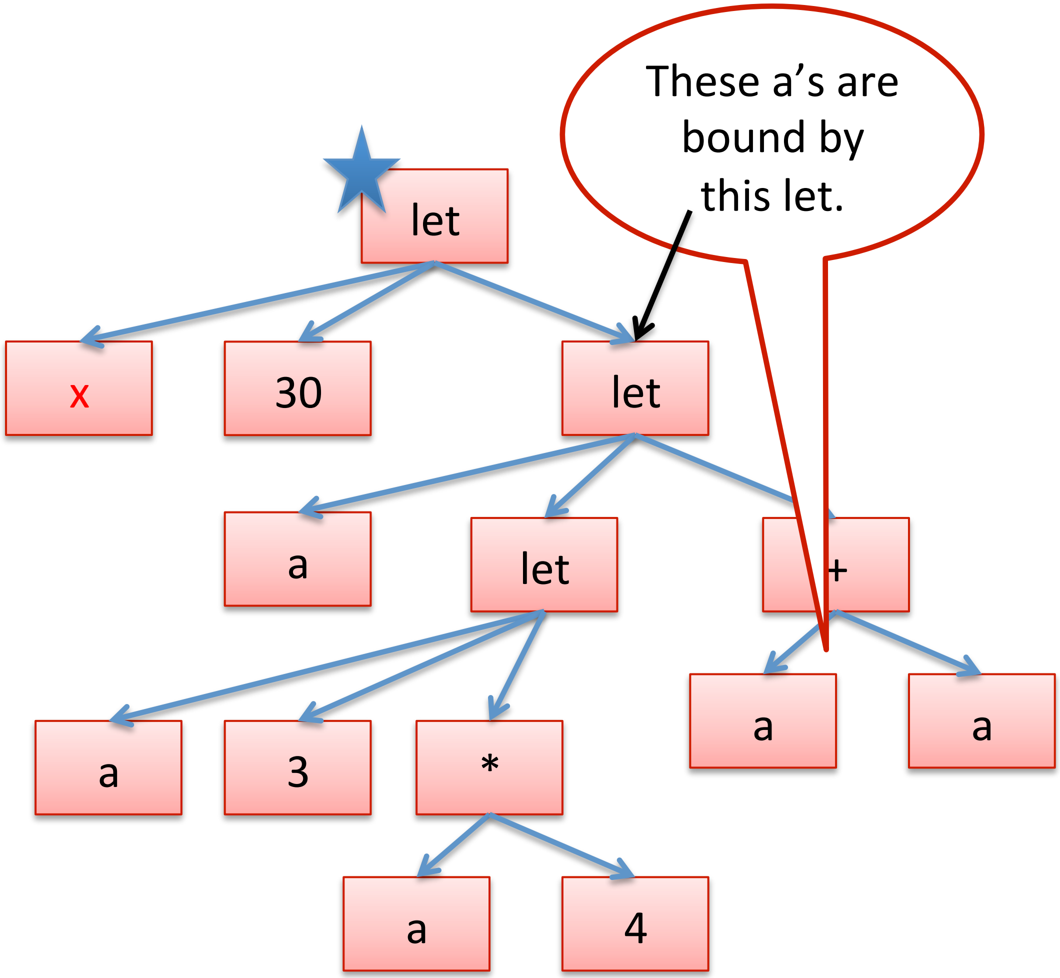
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

There are none in this case!

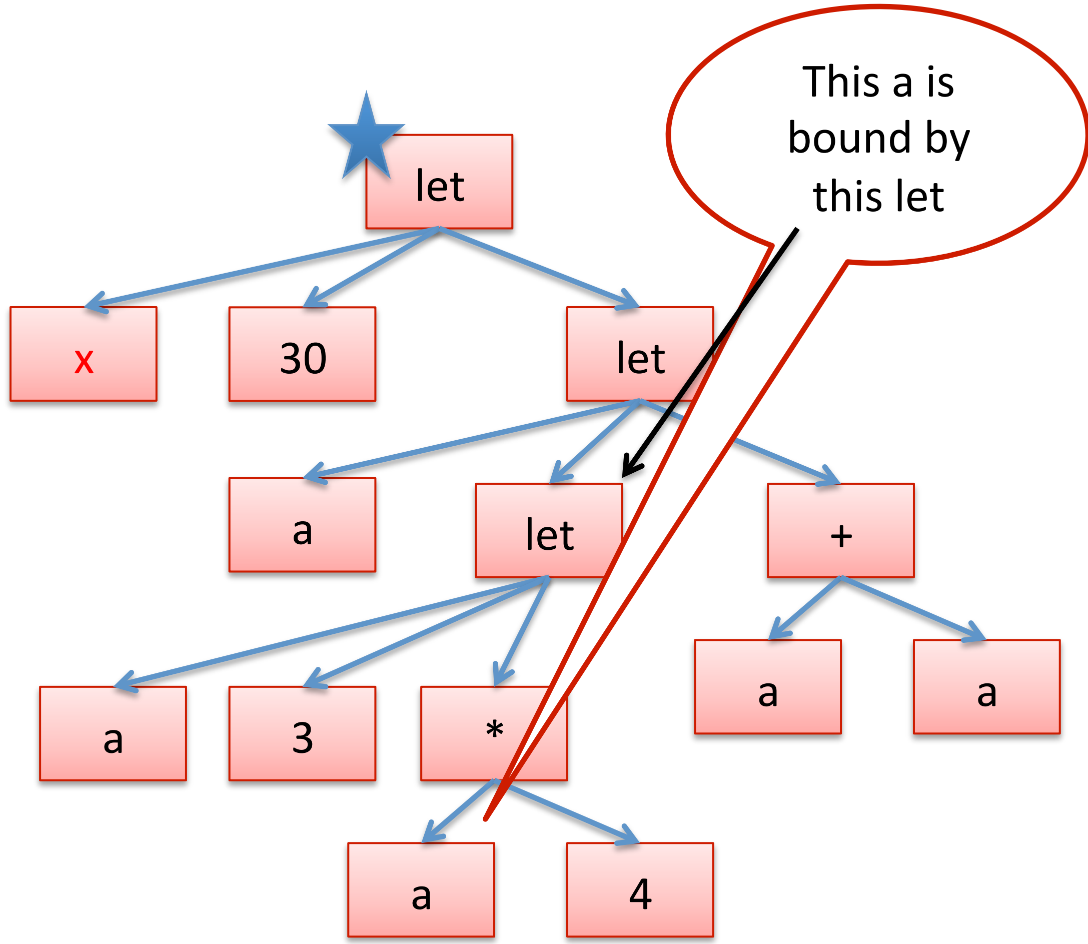
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

There are none in this case!

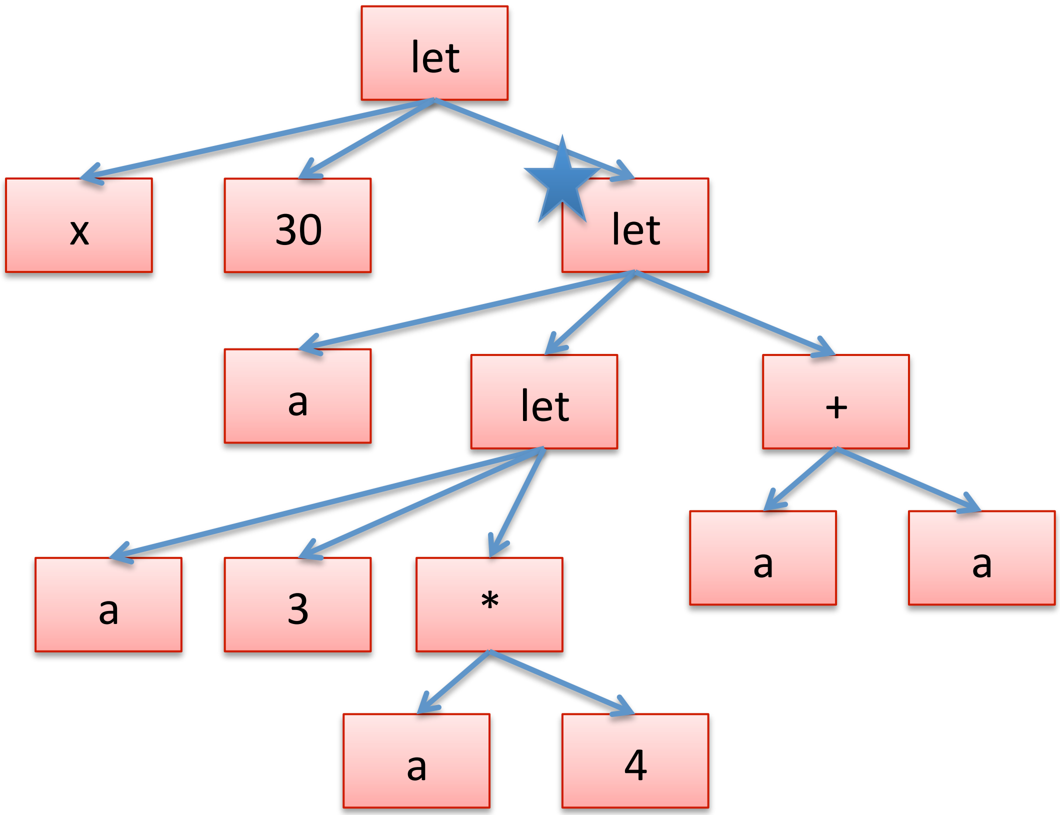
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

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

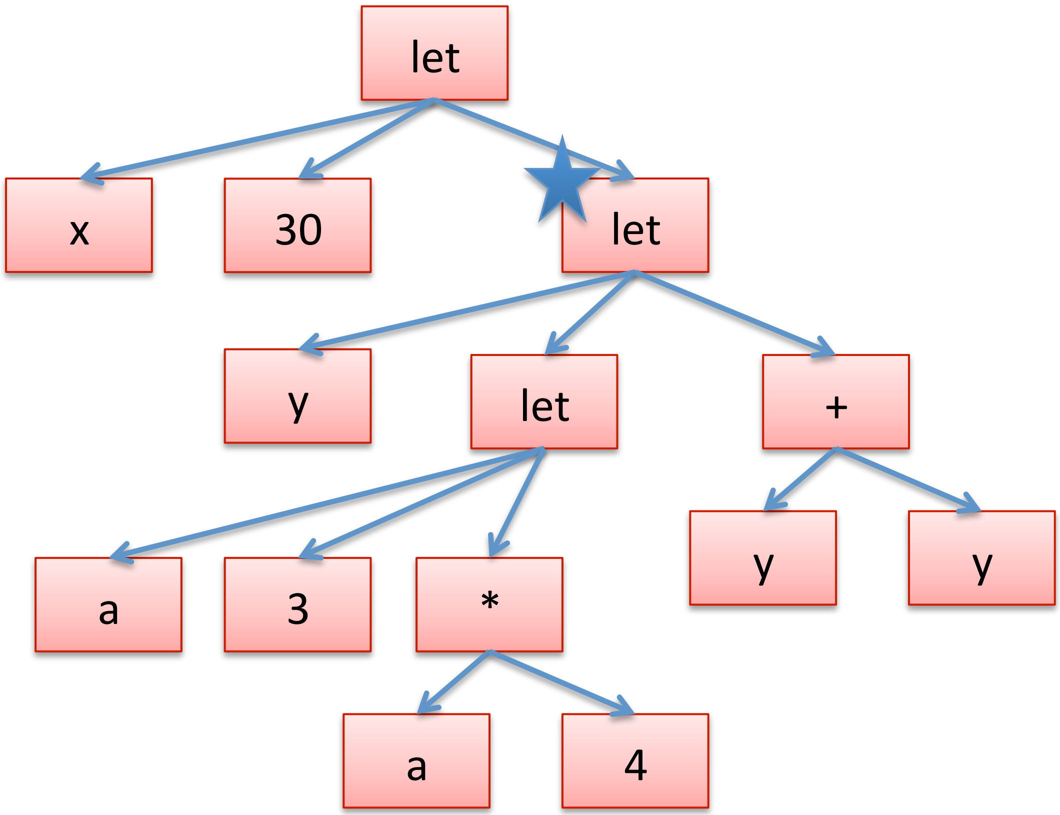
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```



Abstract Syntax Trees

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

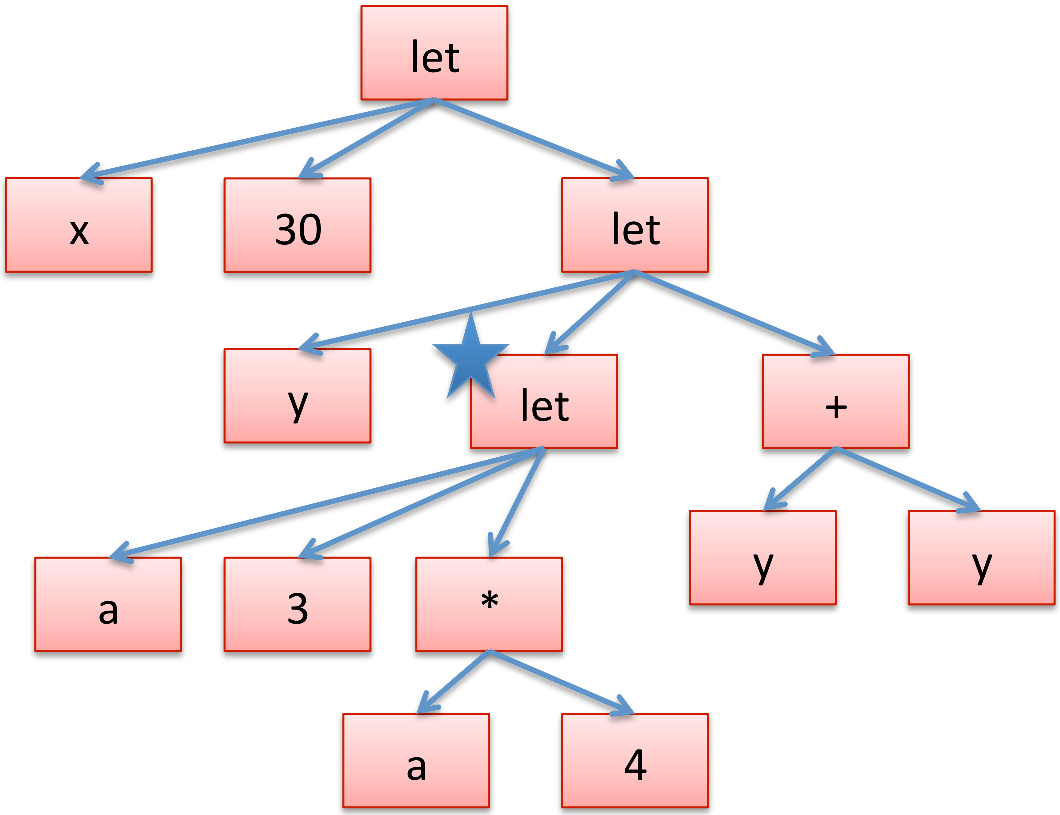
```
let x = 30 in
let y =
  (let a = 3 in a*4)
in
y+y;;
```



Abstract Syntax Trees

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

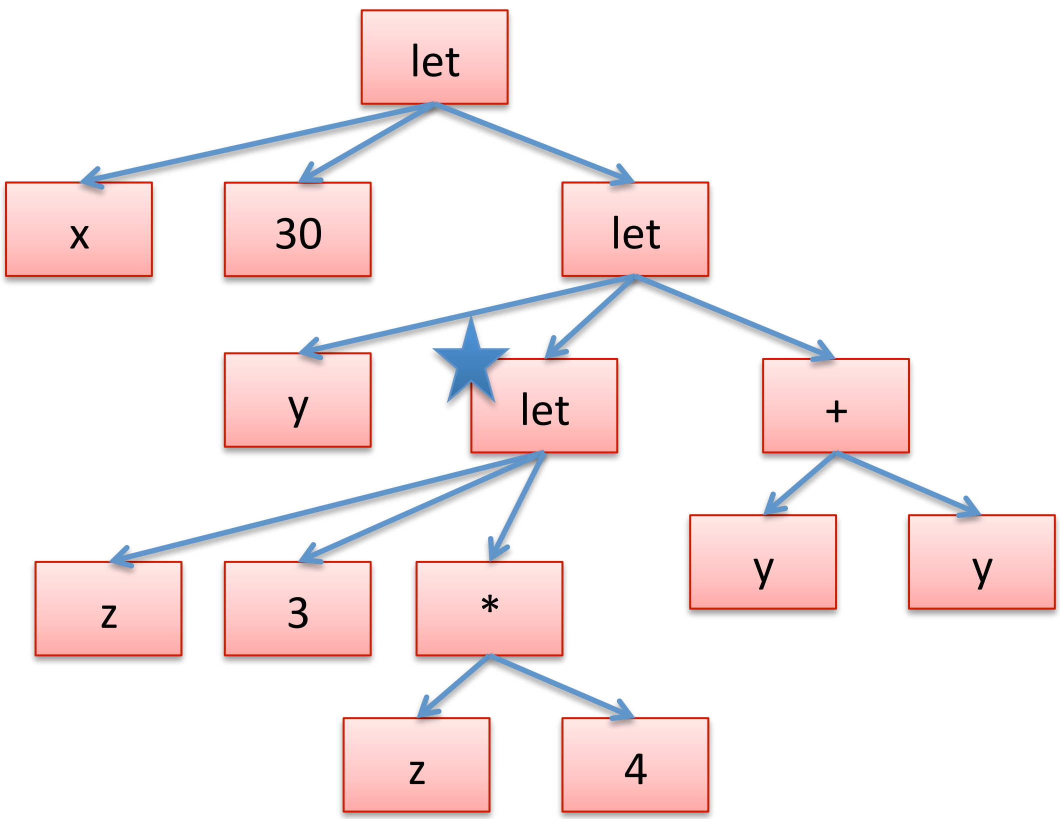
```
let x = 30 in
let y =
  (let z = 3 in z*4)
in
y+y;;
```



Abstract Syntax Trees

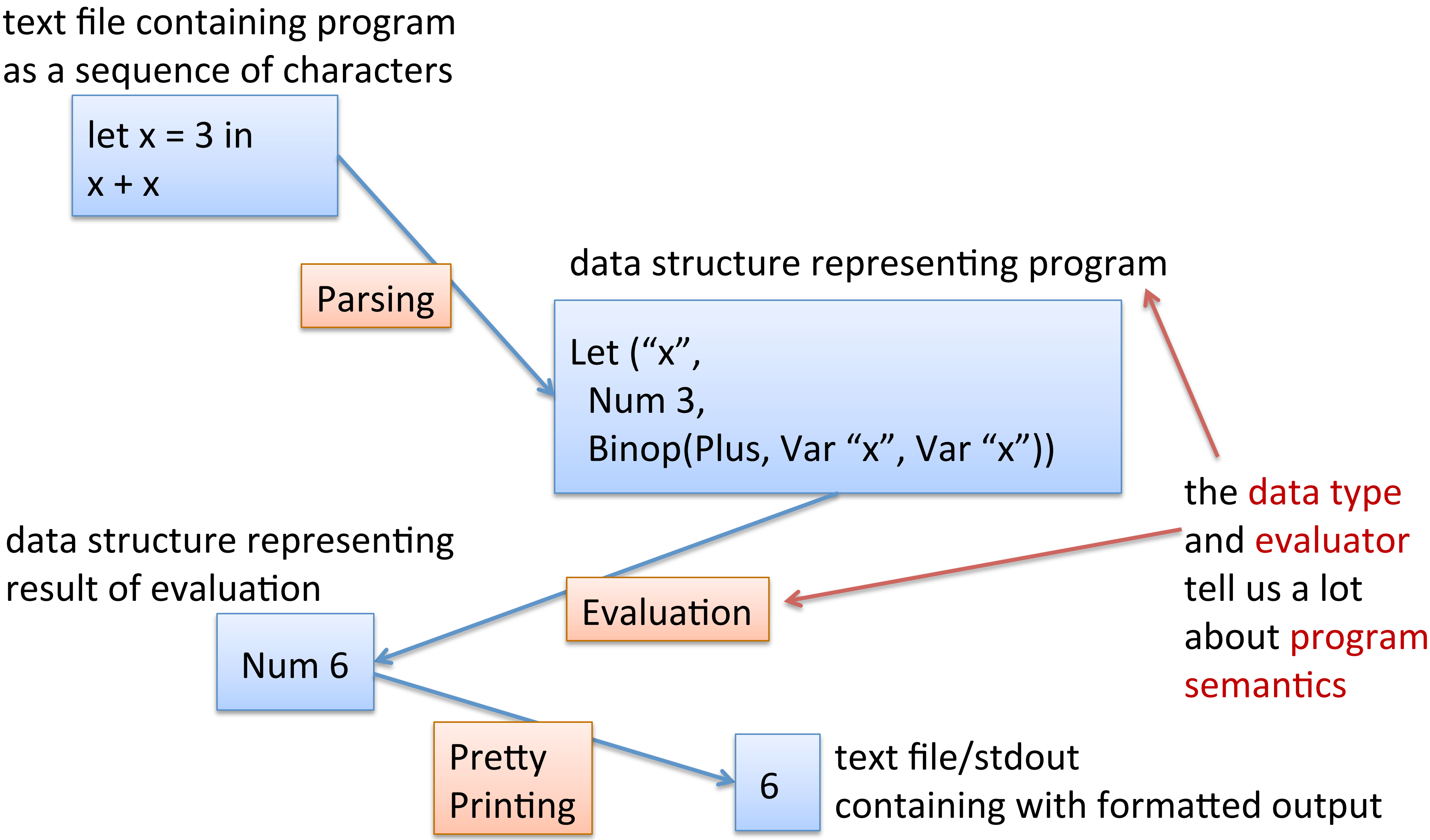
And if we rename the other let to “z”:

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



AN OCAML DEFINITION OF OCAML EVALUATION

Implementing an Interpreter



Making These Ideas Precise

We can define a datatype for simple OCaml expressions:

```
type variable = string ;;
type op = Plus | Minus | Times | … ;;
type exp =
   | Int_e of int
   | Op_e of exp * op * exp
   | Var_e of variable
   | Let_e of variable * exp * exp ;;
```

Making These Ideas Precise

We can define a datatype for simple OCaml expressions:

```
type variable = string ;;
type op = Plus | Minus | Times | … ;;
type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of variable
  | Let_e of variable * exp * exp ;;

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;;
```

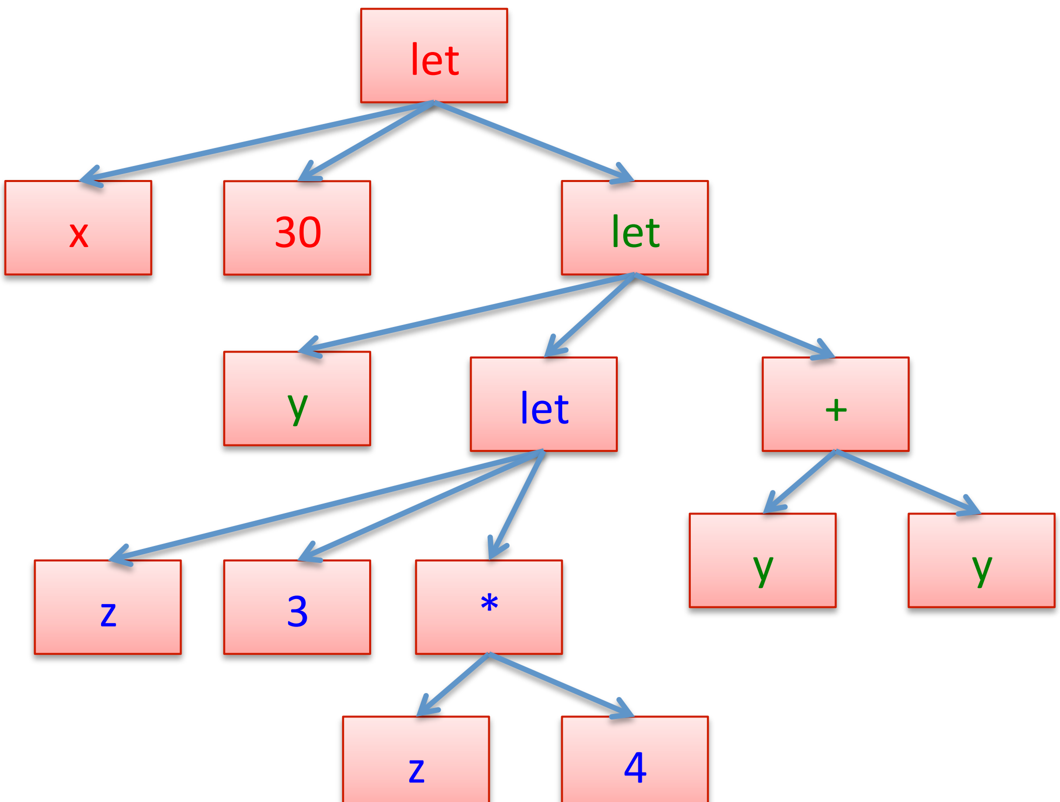
as an exp value:

```
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