Implementing OCaml in OCaml

Part 1: Representing Program Syntax

Speaker: David Walker

COS 326

Princeton University

Defining Programming Language Semantics

To write a program, you have to know how the language works.

Semantics: The study of "how a programming language works"

Defining Programming Language Semantics

To write a program, you have to know how the language works.

Semantics: The study of "how a programming language works"

Methods for defining program semantics:

- Operational: show how to rewrite program expressions step-by-step until you end up with a value
  - we've done some of this already
- Denotational: how interpret a program in a different language that is well understood
  - we aren't going to do much of this – see COS 510
- Equational: specify the equal programs
  - we'll do more of this later & use this semantics to prove things about our programs
- Axiomatic: provide (other kinds of) reasoning rules about programs

Defining Program Semantics

In this series of lectures, we'll focus on operational definitions

We'll use the following techniques to communicate:

1. *examples* (good for intuition, but highly incomplete)
   - this doesn't get at the corner cases
2. *an interpreter program* written in OCaml
3. *mathematical notation*

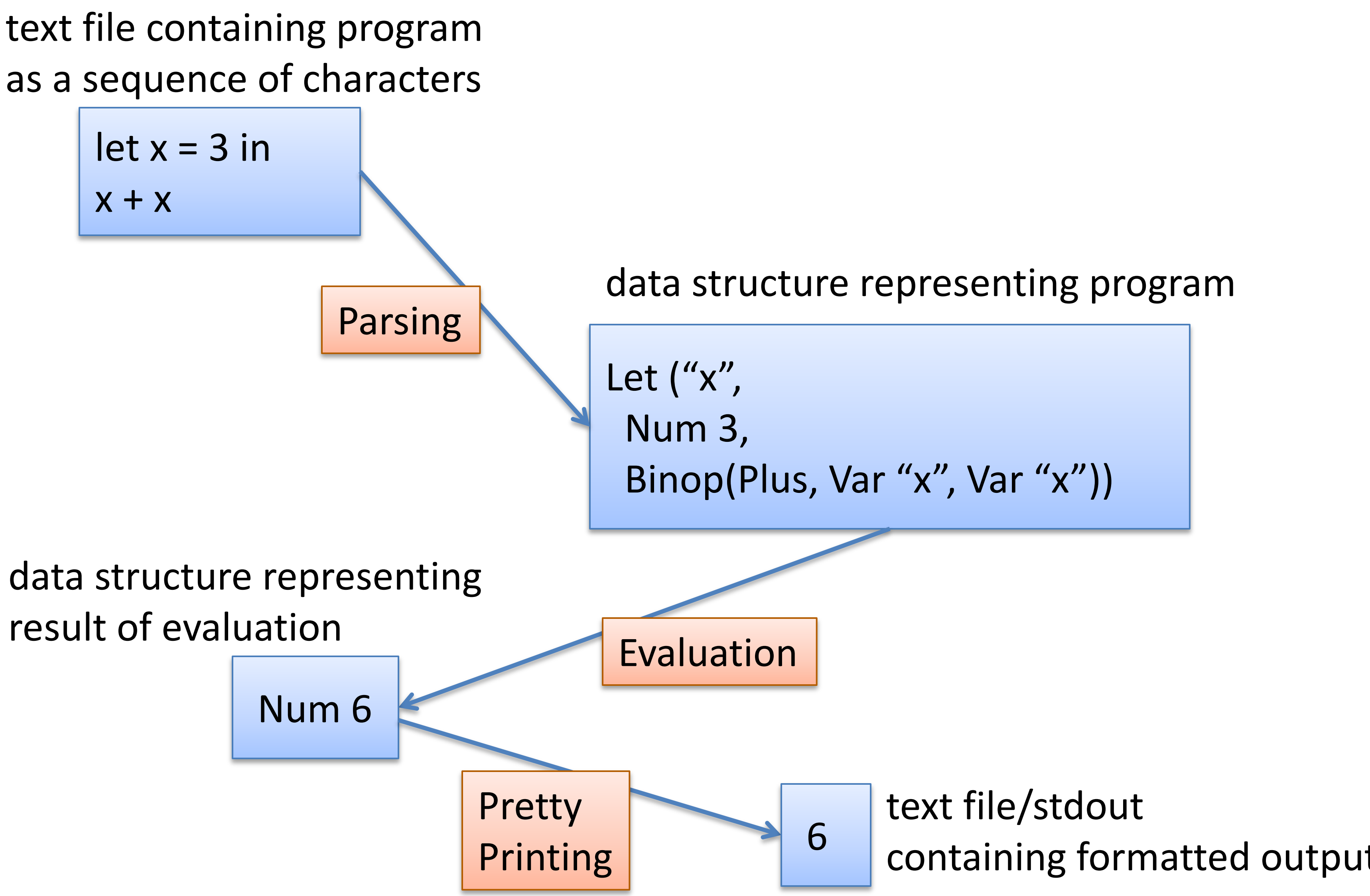
Defining Program Semantics

In this series of lectures, we'll focus on operational definitions

We'll use the following techniques to communicate:

1. *examples* (good for intuition, but highly incomplete)
   - this doesn't get at the corner cases
2. *an interpreter program* written in OCaml
3. *mathematical notation*

Implementing an Interpreter



REPRESENTING SYNTAX

Representing Syntax

Program syntax is a complicated tree-like data structure.

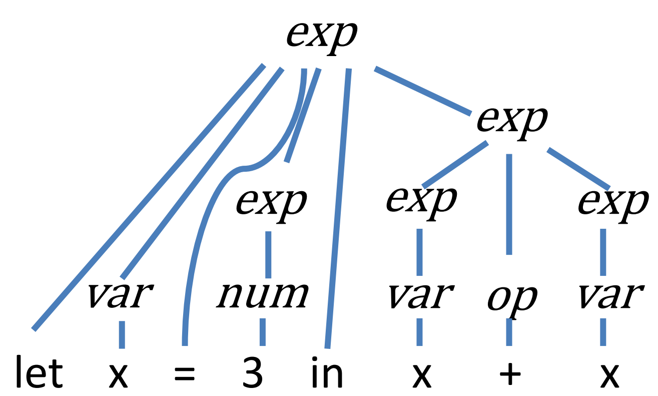
Representing Syntax

Program syntax is a complicated tree-like data structure.

```
let x = 3 in
x + x
```

Syntax Trees

let x = 3 in x + x

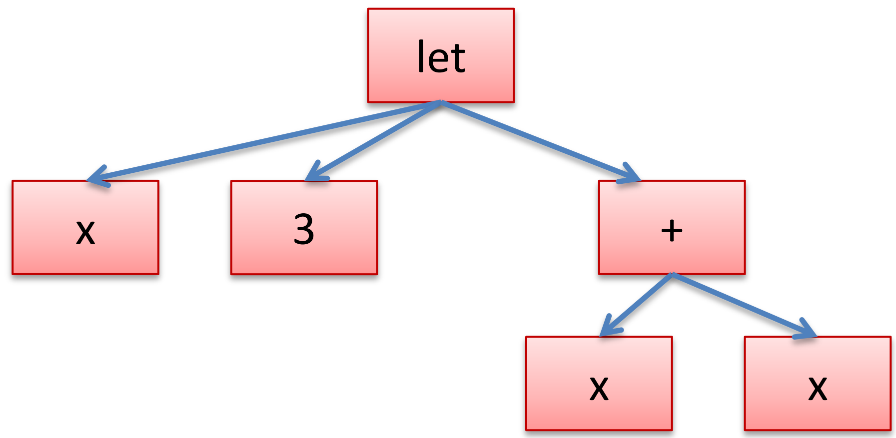


This is the *parse tree*.
Useful for some purposes, but
for the semantics it's *too much information*.

Abstract Syntax Tree (AST)

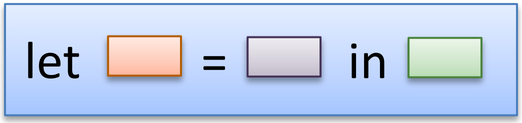
Don’t need all the “punctuation” (key words, white space).

```
let x = 3 in
x + x
```



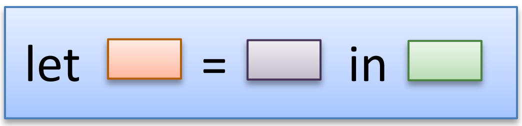
Representing Syntax

More generally each let expression has 3 parts:

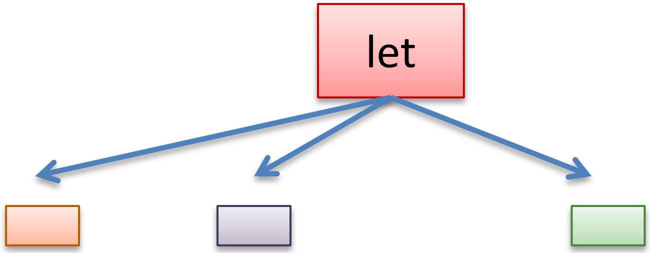


Representing Syntax

More generally each let expression has 3 parts:

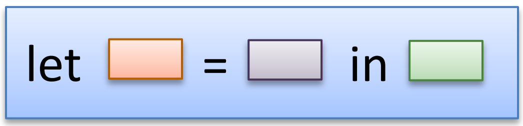


And you can represent a let expression using a tree like this:

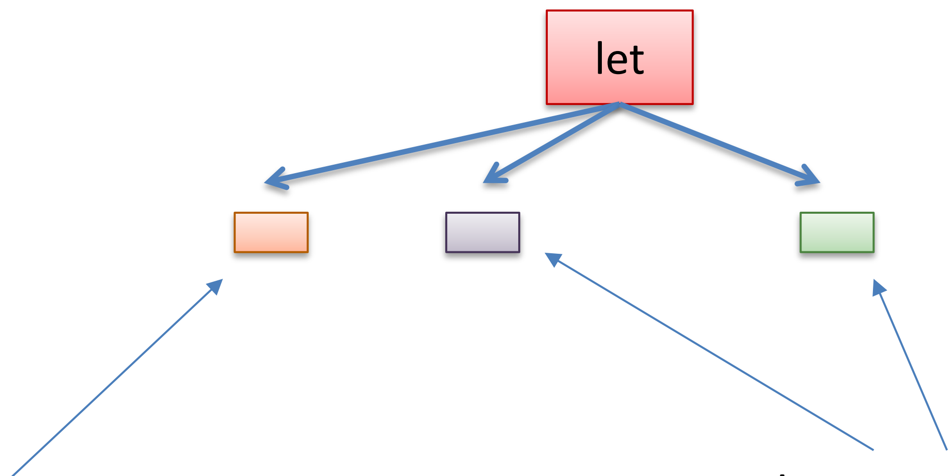


Representing Syntax

More generally each let expression has 3 parts:



And you can represent a let expression using a tree like this:

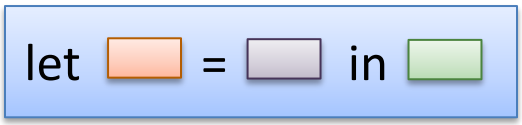


this part has to contain a variable, like x

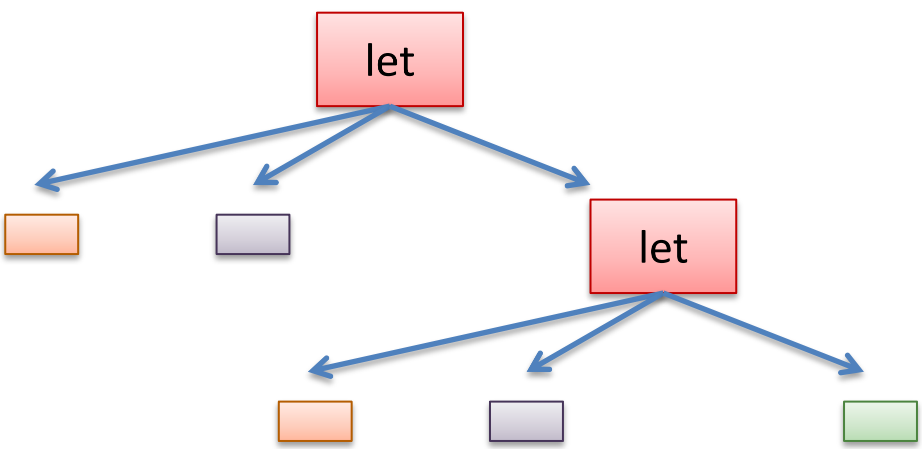
these parts contain arbitrary subexpressions

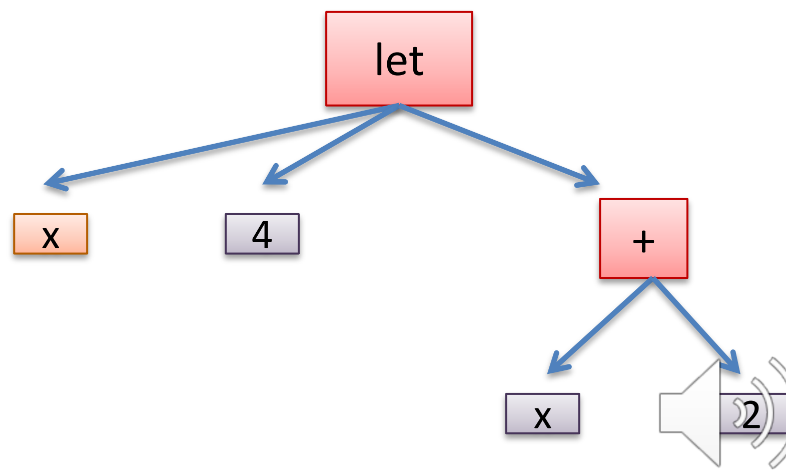
Representing Syntax

More generally each let expression has 3 parts:



And you create complicated programs by nesting let expressions (or any other expression) recursively inside one another:





OCaml for the Win

Functional programming languages have sometimes been called "domain-specific languages for compiler writers"

Datatypes are amazing for representing complicated tree-like structures and that is exactly what a program is.

Use a different constructor for every different sort of expression

- one constructor for variables
- one constructor for let expressions
- one constructor for numbers
- one constructor for binary operators, like add
- ...

Aside: Java for the loss

Languages like Java, that are based exclusively around heavy-weight class tend to be vastly more verbose when trying to represent syntax trees:

- one whole class for each different kind of syntax
- one class for variables
- one class for let expressions
- one class for numbers ...

In addition, writing traversals over the syntax is annoying, because your code is spread over N different classes (using a visitor pattern) rather than in one place.

Aside: Java for the loss

Languages like Java, that are based exclusively around heavy-weight class tend to be vastly more verbose when trying to represent syntax trees:

- one whole class for each d... ntax
- one class for
- o...
- one cla...

SCORE: OCAML 3.8, JAVA 0

(C: who cares?)

In addi... anno...
becaus... sses (using a
visit... one p... ce.

Making These Ideas Precise

A datatype for simple OCaml expressions:

```
type variable = string

type op = Plus | Minus | Times | …

type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of variable
   | Let of variable * exp * exp

type value = exp
```

Making These Ideas Precise

A datatype for simple OCaml expressions:

```
type variable = string
type op = Plus | Minus | Times | …
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of variable
   | Let of variable * exp * exp
type value = exp

let e1 = Int 3
```

Making These Ideas Precise

A datatype for simple OCaml expressions:

```
type variable = string
type op = Plus | Minus | Times | …
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of variable
   | Let of variable * exp * exp
type value = exp

let e1 = Int 3
let e2 = Int 17
```

Making These Ideas Precise

A datatype for simple OCaml expressions:

```
type variable = string
type op = Plus | Minus | Times | …
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of variable
   | Let of variable * exp * exp
type value = exp

let e1 = Int 3
let e2 = Int 17
let e3 = Op (e1, Plus, e2)
```

represents "3 + 17"

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
```

This is called
concrete syntax
(concrete syntax pertains to parsing)

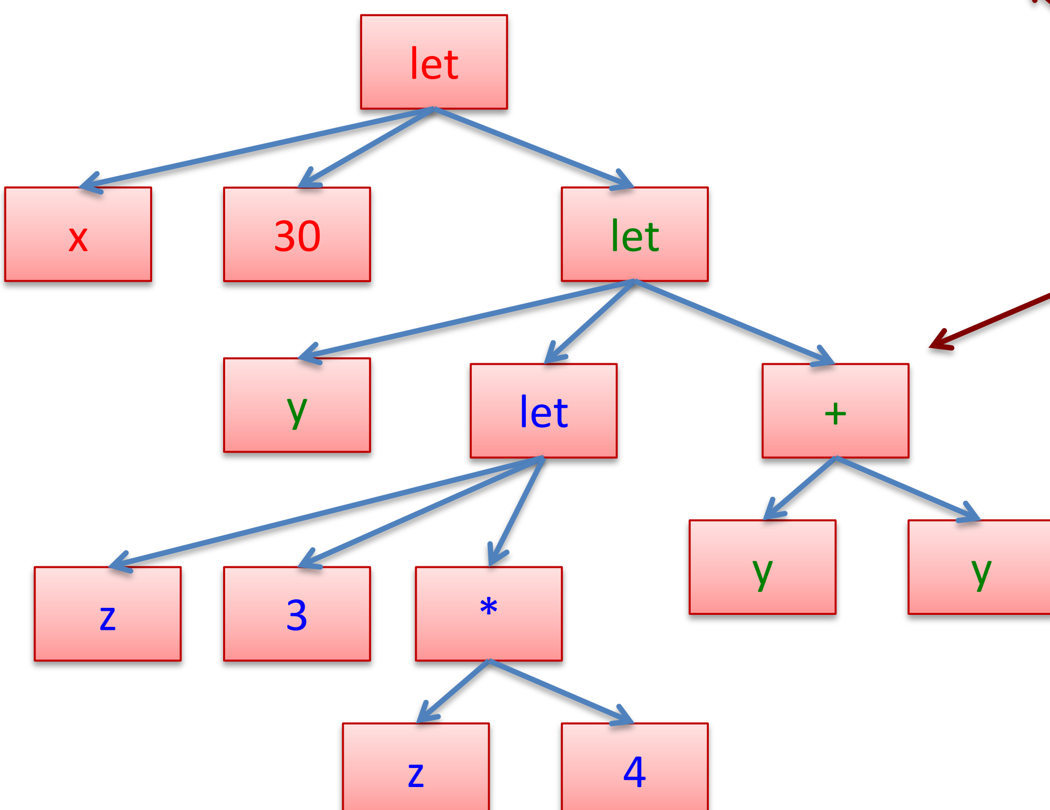
This is called an
abstract syntax tree (AST)

as an exp value:

```
Let("x", Int 30,
      Let("y",
            Let("z", Int 3,
                  Op(Var "z", Times, Int 4)),
      Op(Var "y", Plus, Var "y")
```

ASTs as ... Trees

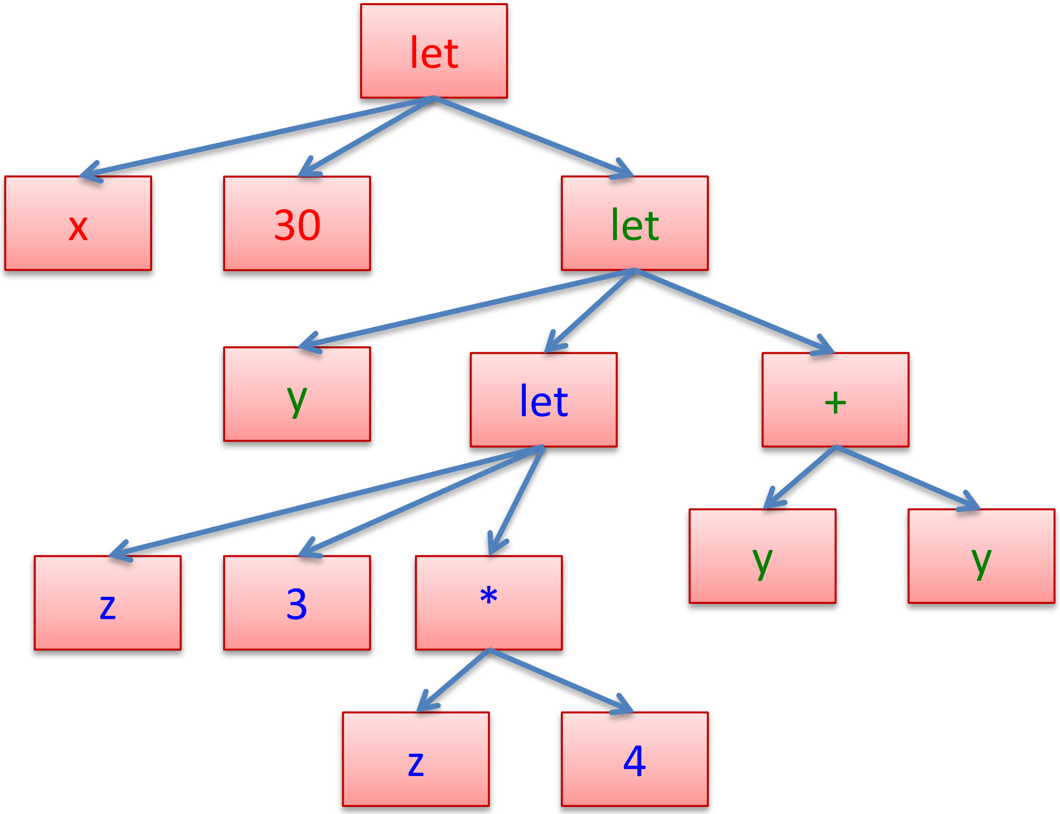
```
Let("x",Int 30,
      Let("y",Let("z",Int 3,
                      Op(Var "z", Times, Int 4)),
           Op(Var "y", Plus, Var "y")
```



Visualizing the OCaml expression as a tree

ASTs as ... Trees

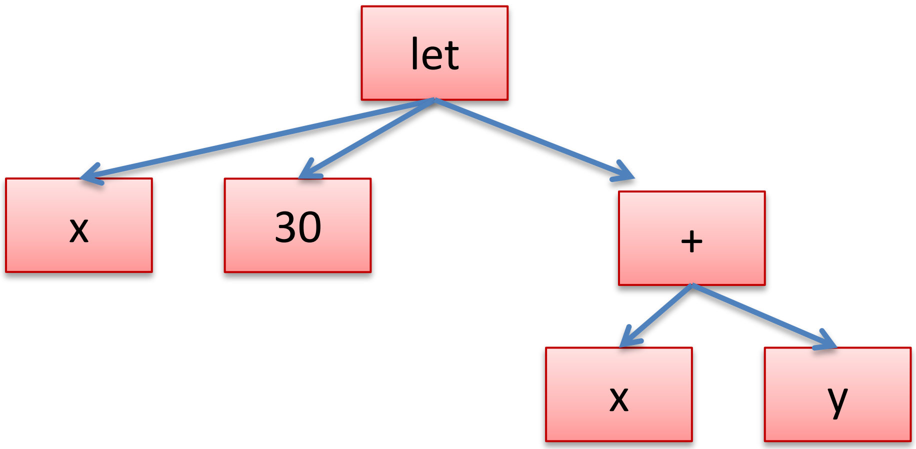

```
Let("x",Int 30,
        Let("y",Let("z",Int 3,
                     Op(Var "z", Times, Int 4)),
             Op(Var "y", Plus, Var "y")
```



Now that we have a data structure to represent programs, we can write other programs to analyze them.

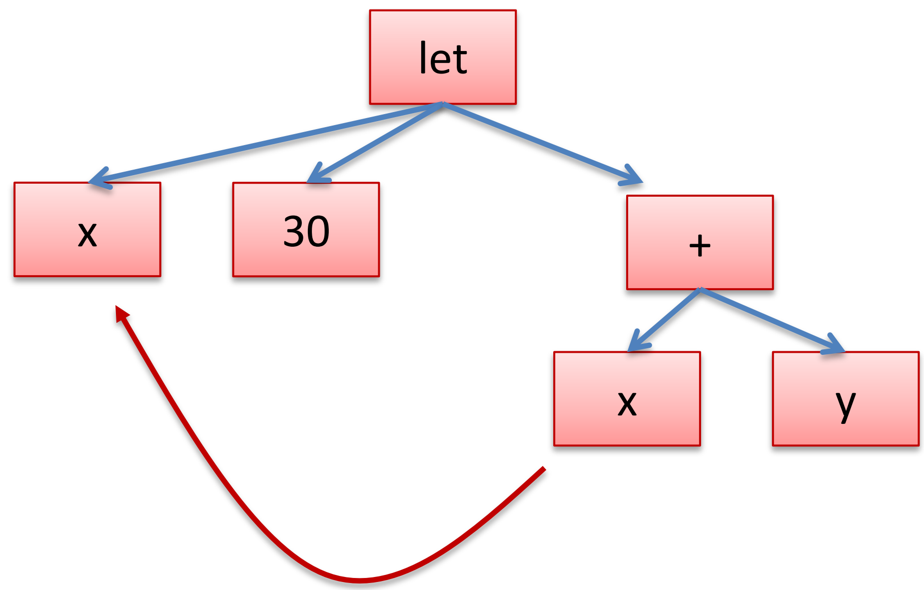
Free vs Bound Variables


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



Free vs Bound Variables

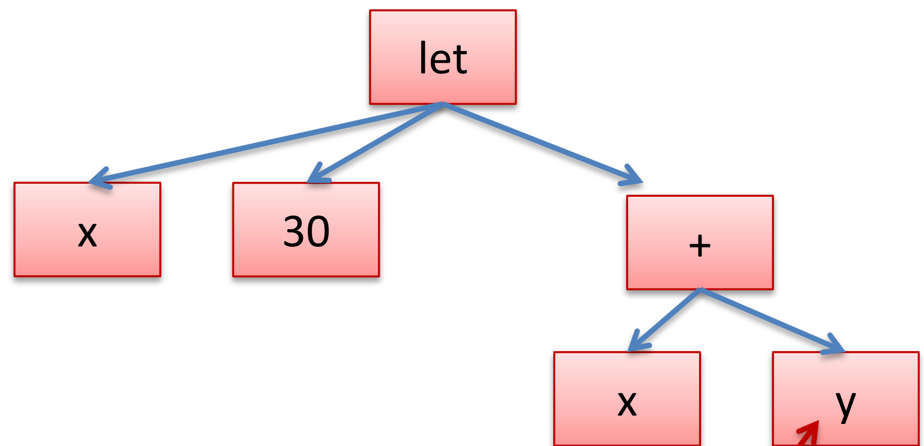
```
let x = 30 in
x+y
```



this use of x is bound here

Free vs Bound Variables

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



this use of y is free

we say: "y is a free variable in the expression (let x = 30 in x+y)"

Other Examples


```
fun z -> z + y
```

z is bound
y is a free variable

```
match x with
  (y,z) -> y + z + w
```

x, w are free variables
y, z are bound

```
let rec f x =
  match x with
    [] -> y
  | hd:tl -> hd::f tl
```

y is a free variable
f, x, hd, tl are all bound

A Few More Examples

What are the free variables of the following expressions?

```
if true then x else y
```

→ x and y

```
(fun x y ->
  match x with
    [] -> 0
  | hd::tl -> w + hd) [] z
```

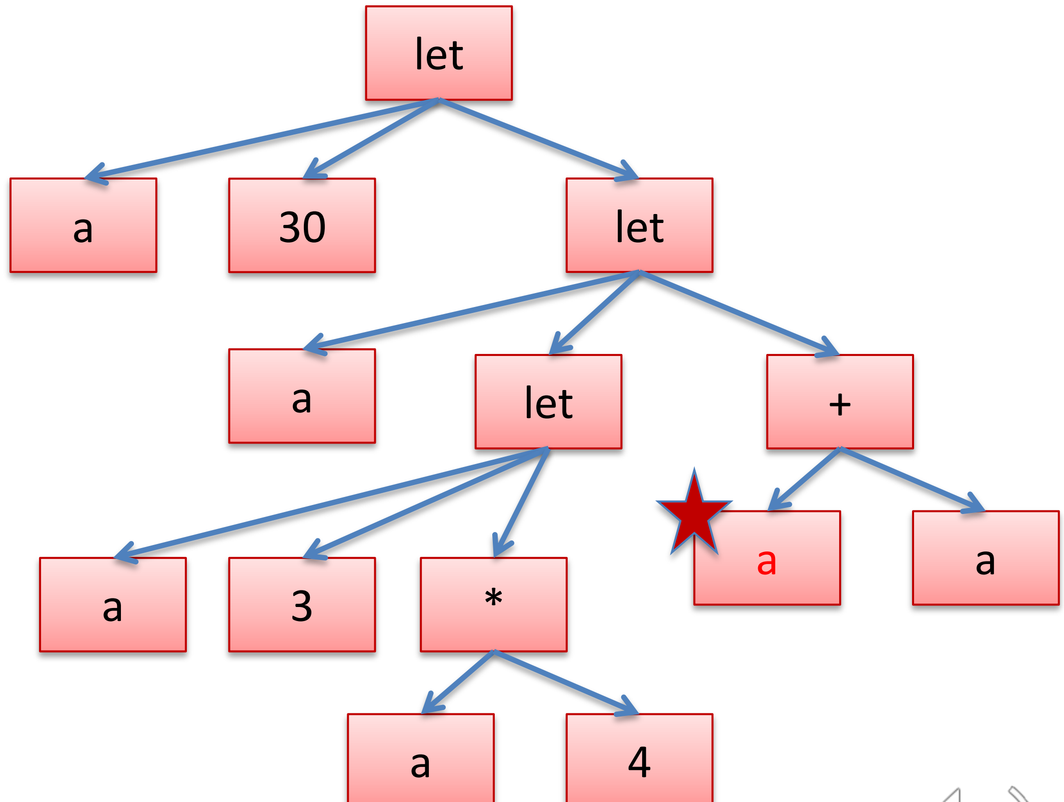
→ w and z

The free variables of an expression do not depend upon the flow of control.

Abstract Syntax Trees

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

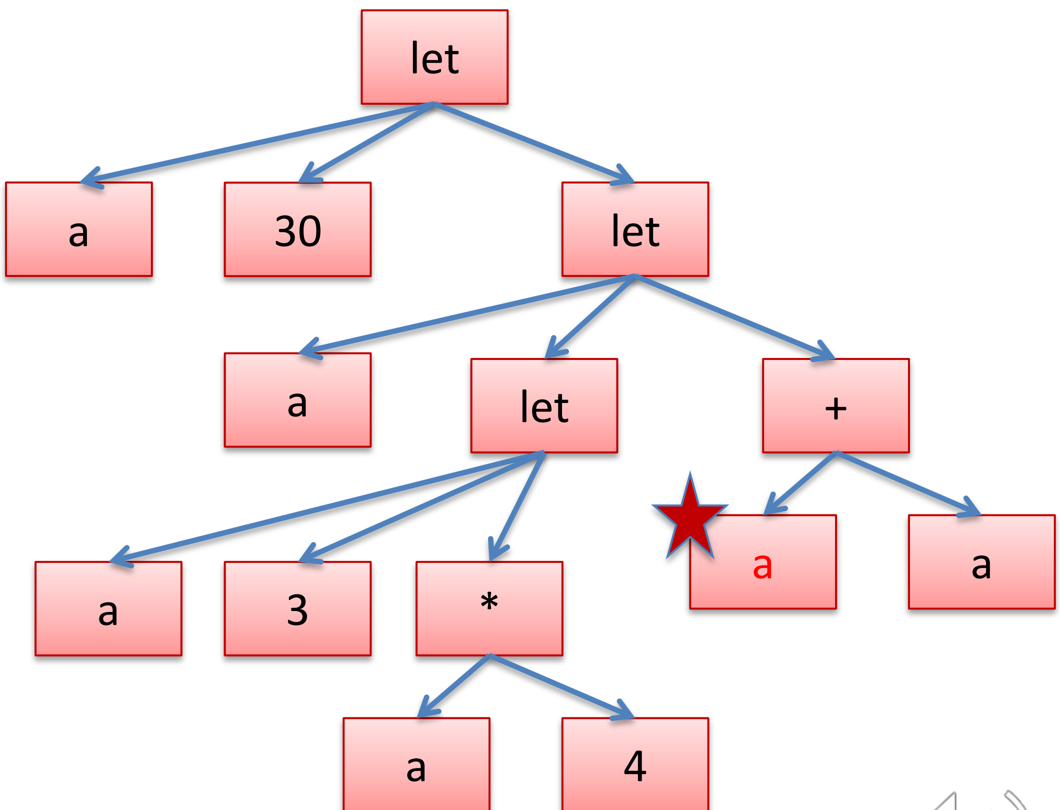
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

crawling up the tree to the nearest enclosing let…

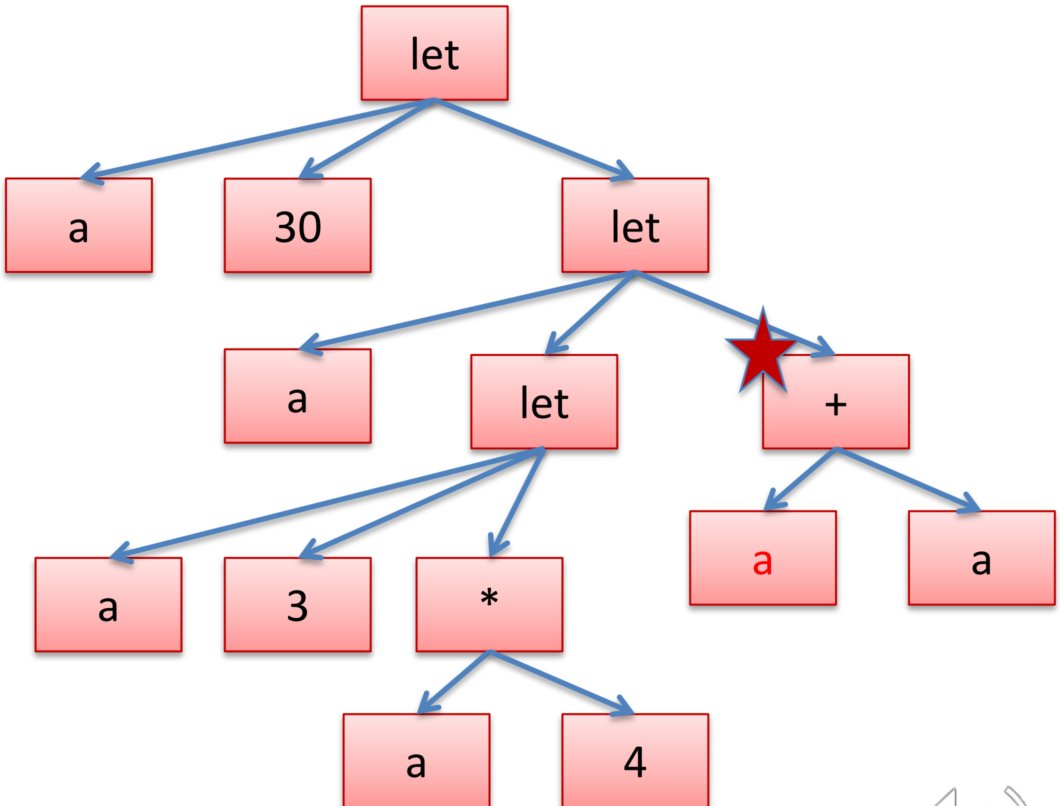
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

crawling up the tree to the nearest enclosing let…

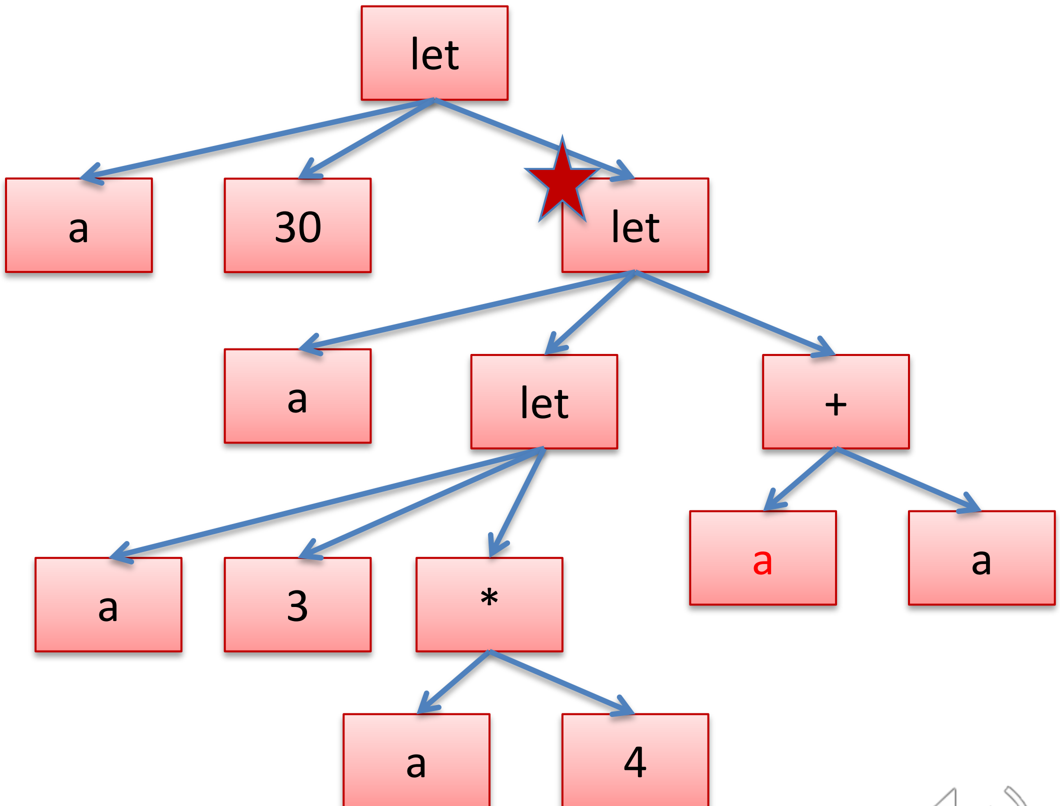
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

crawling up the tree to the nearest enclosing let...

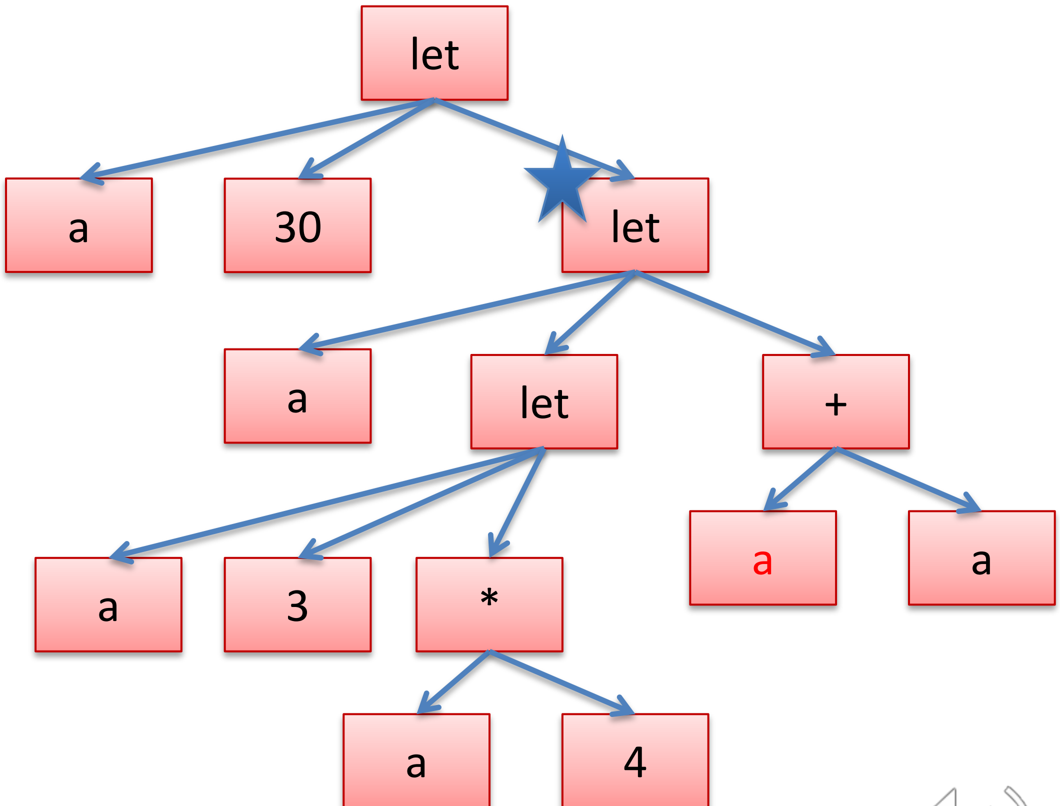
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

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

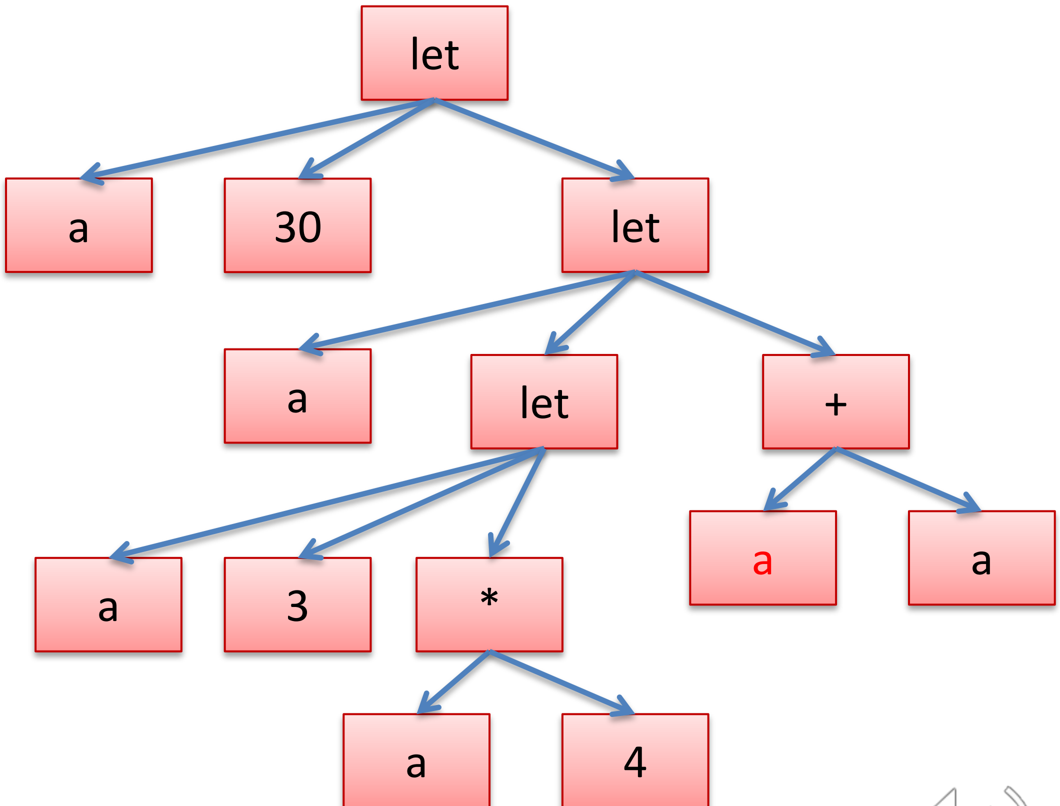
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

We can also systematically rename the variables so that it's not so confusing. Recall systematic renaming is called *alpha-conversion*

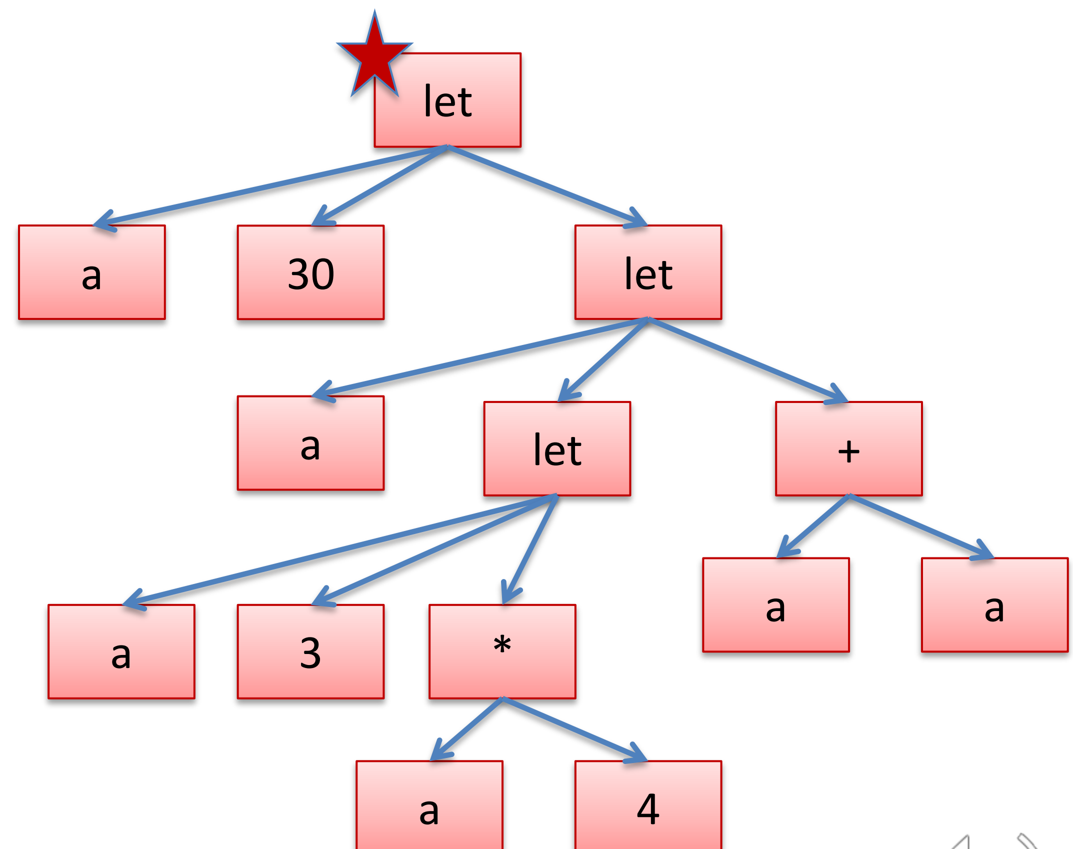
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

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

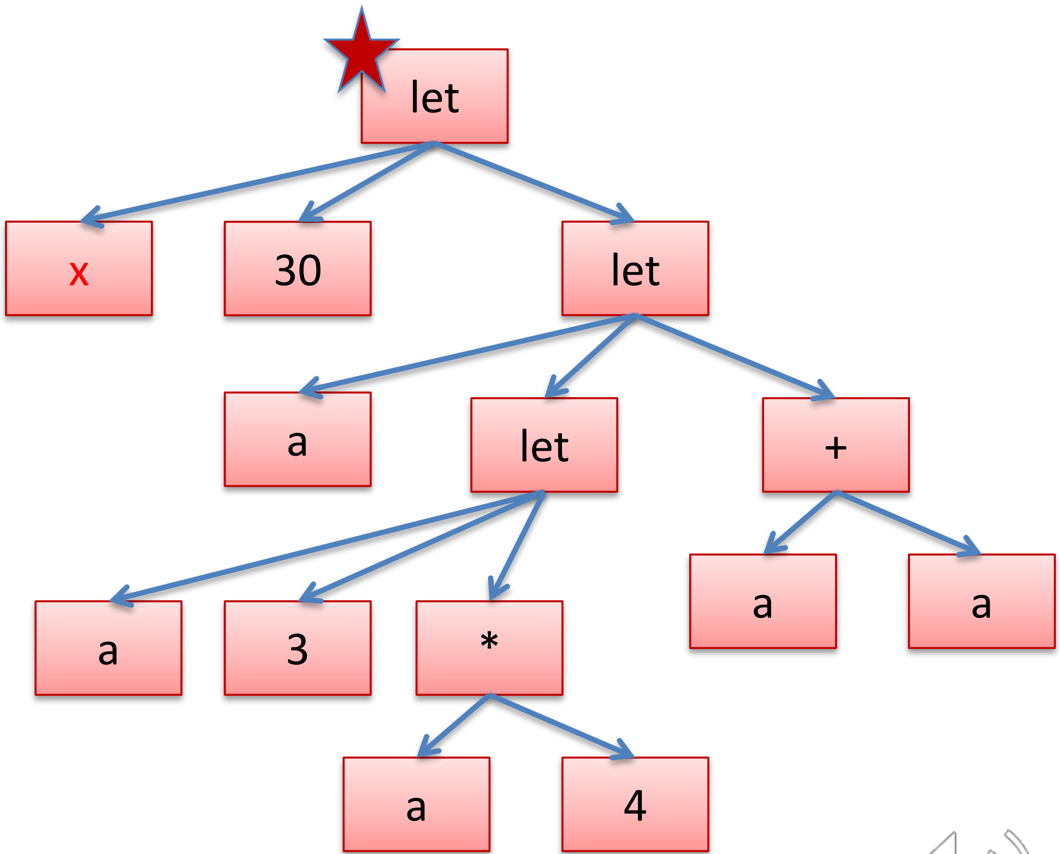
```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

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

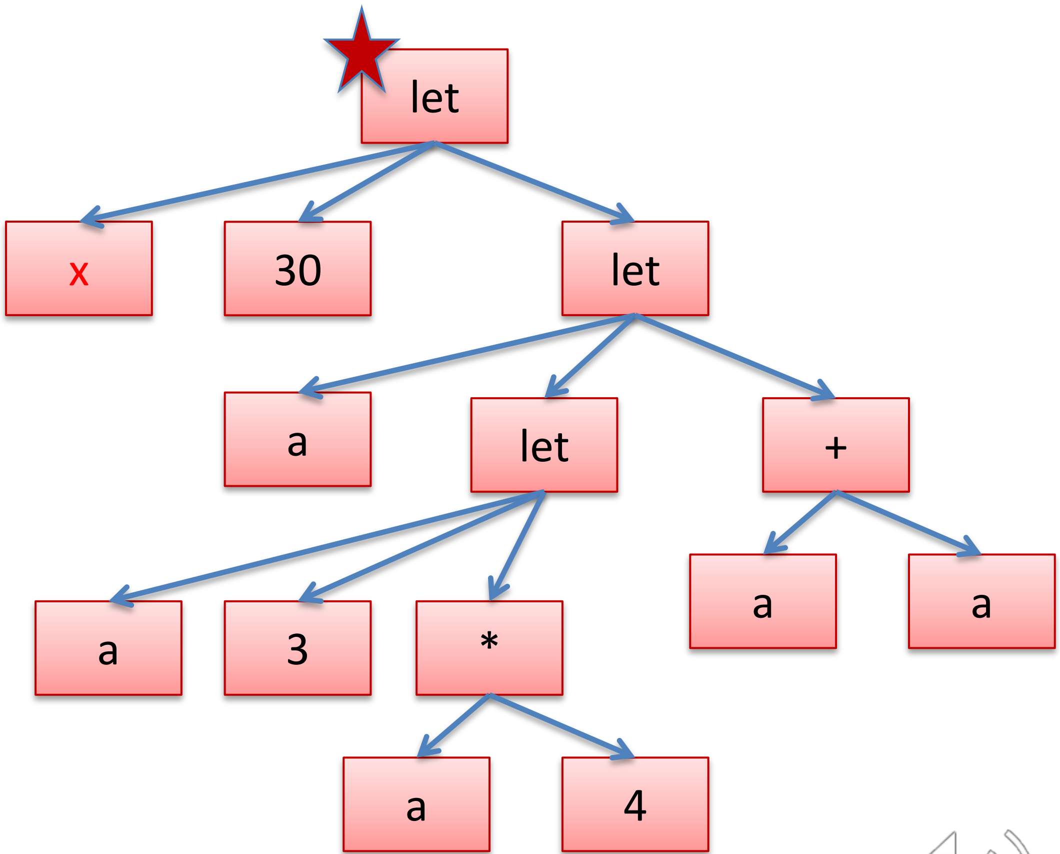
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

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

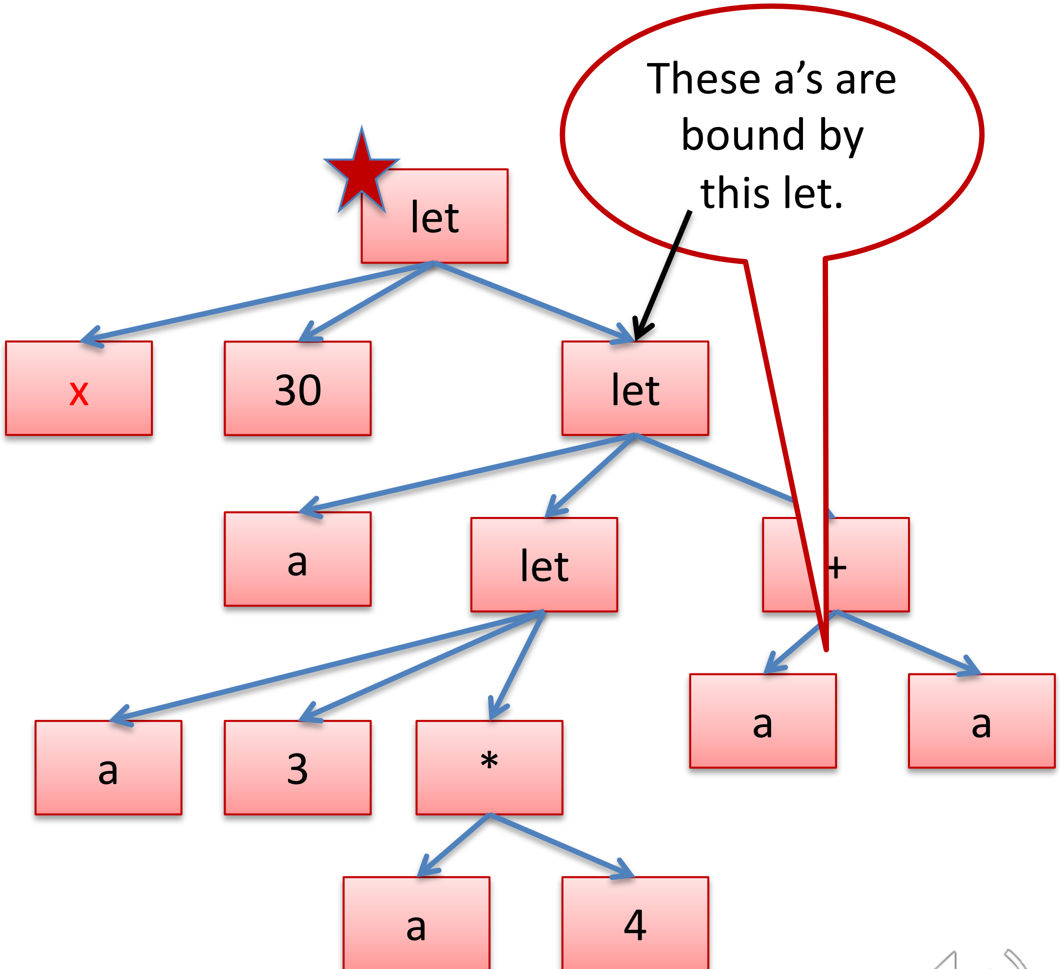
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

There are none in this case!

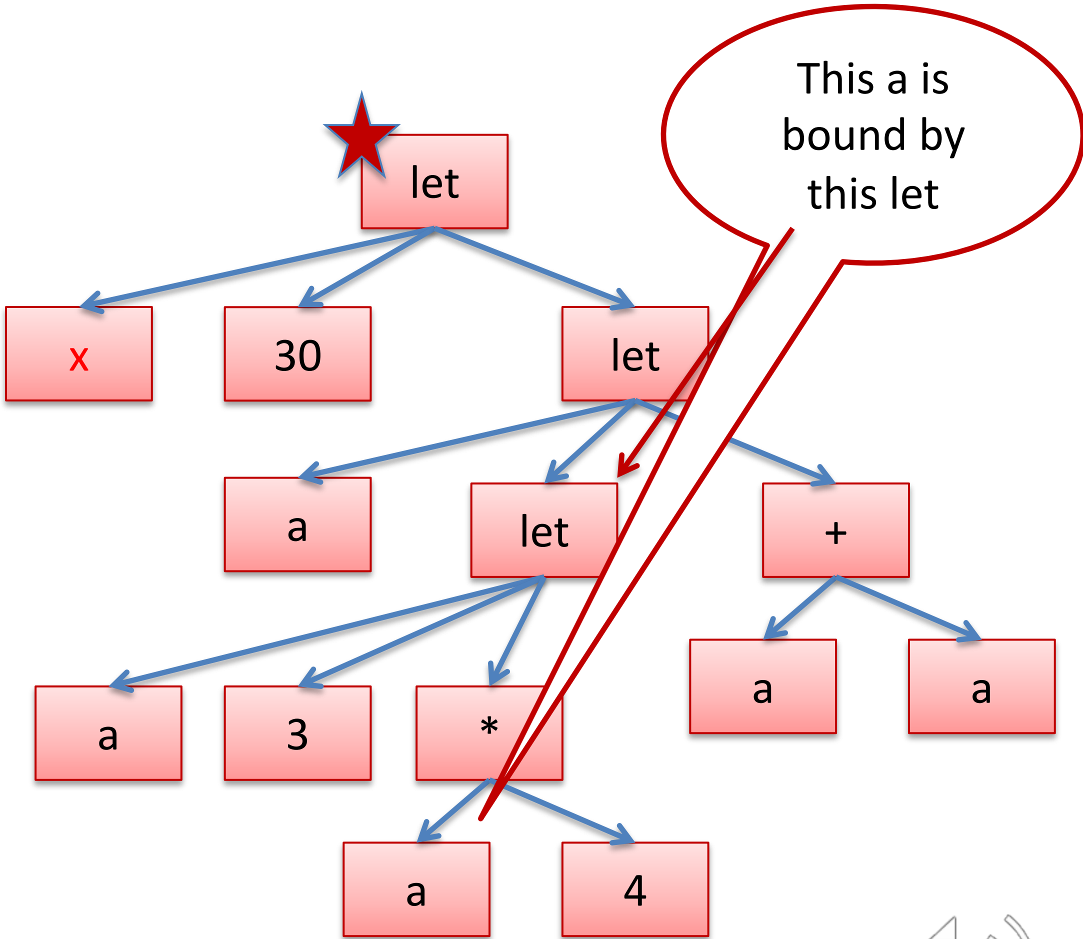
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

There are none in this case!

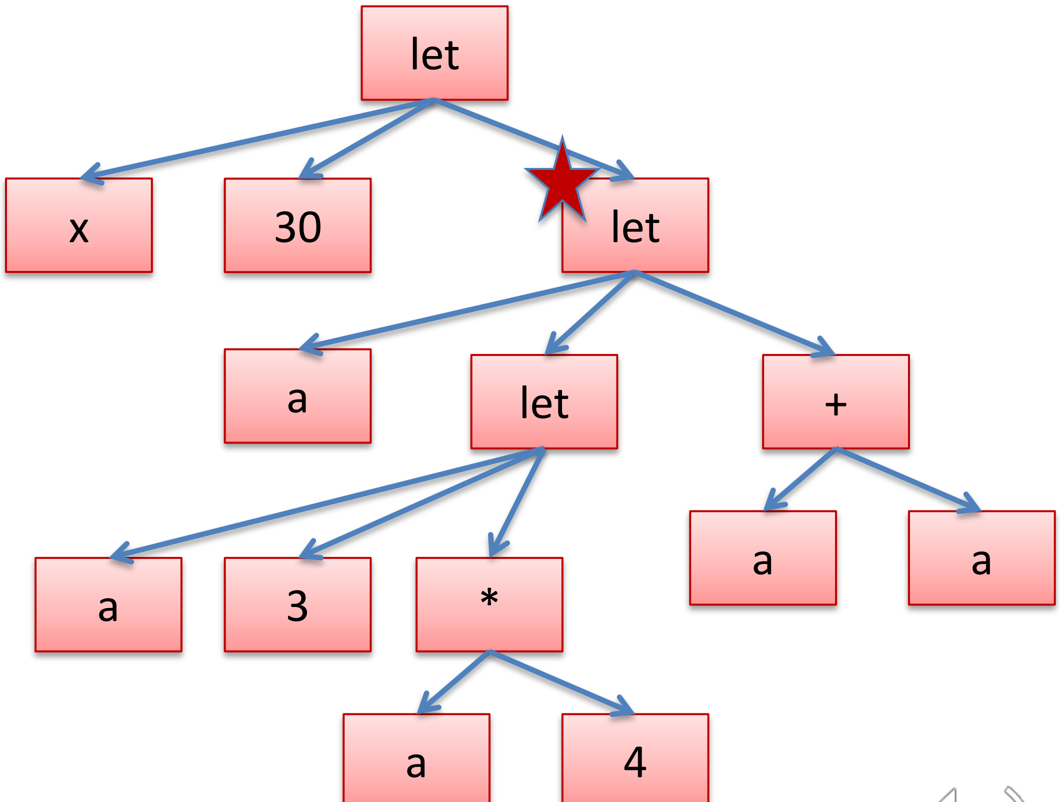
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



Abstract Syntax Trees

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

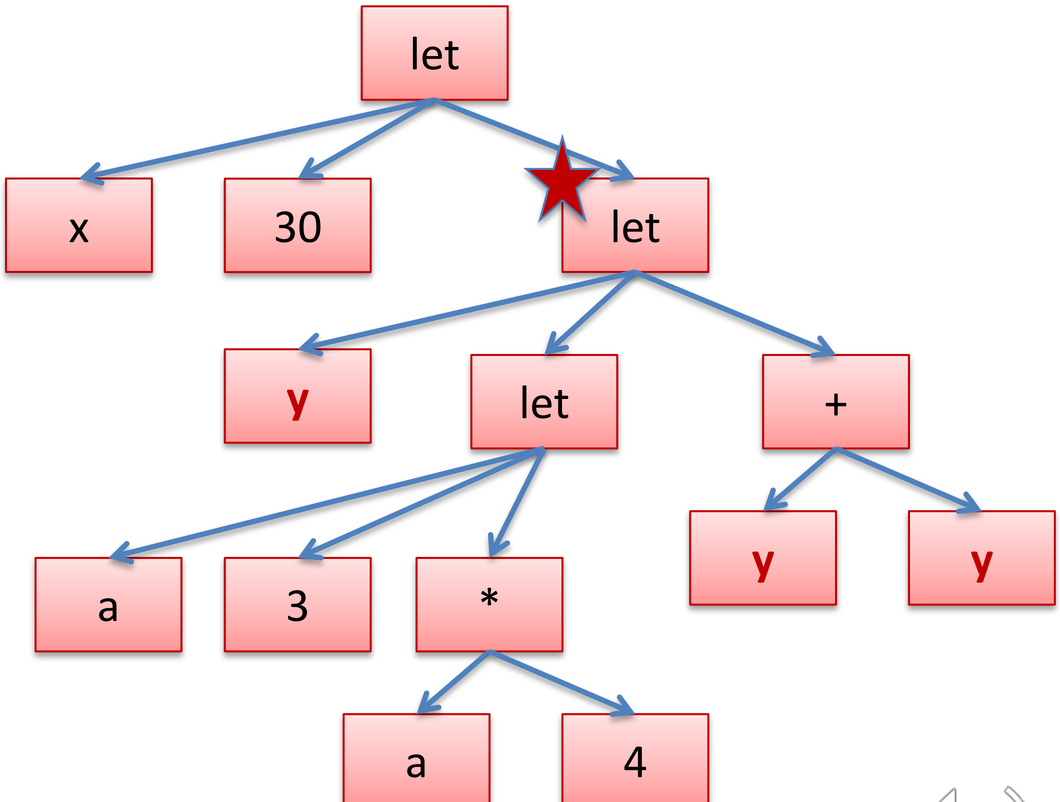
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a
```



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
```



Implementing Renaming


```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

```
let rec rename (x:var) (y:var) (e:exp) : exp =
```

Implementing Renaming


```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

```
let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
  | Op  (e1, op, e2) ->

  | Var z ->

  | Int i ->

  | Let (z,e1,e2) ->
```

Implementing Renaming

```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

```
let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
   | Op  (e1, op, e2) ->
       Op (rename x y e1, op, rename x y e2)
   | Var z ->

   | Int i ->

   | Let (z,e1,e2) ->
```

Implementing Renaming

```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

```
let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
   | Op  (e1, op, e2) ->
       Op (rename x y e1, op, rename x y e2)
   | Var z ->
       if z = x then Var y else e
   | Int i ->

   | Let (z,e1,e2) ->
```

Implementing Renaming

```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

```
let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
   | Op  (e1, op, e2) ->
       Op (rename x y e1, op, rename x y e2)
   | Var z ->
       if z = x then Var y else e
   | Int i ->
       Int i
   | Let (z,e1,e2) ->
```

Implementing Renaming


```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

```
let rec rename (x:var) (y:var) (e:exp) : exp =
  match e with
   | Op  (e1, op, e2) ->
       Op (rename x y e1, op, rename x y e2)
   | Var z ->
       if z = x then Var y else e
   | Int i ->
       Int i
   | Let (z,e1,e2) ->
       Let (z, rename x y e1,
              if z = x then e2 else rename x y e2)
```

Exercise

Here's the syntax of our little language:

```
type var = string
type op = Plus | Minus
type exp =
   | Int of int
   | Op  of exp * op * exp
   | Var of var
   | Let of var * exp * exp
```

Extending the abstract syntax of expressions. Extend the implementation of the renaming function.

- (Easy) Booleans true and false, if statements, and operations like and, or, not
- (Harder) Pairs and patterns "let (x,y) = e1 in e2"