CS558 Programming Languages

Fall 2015
Lecture 2a

Andrew Tolmach
Portland State University

Review: Expressions

- ☐ They are usually tree-structured
- ☐ They can be defined over many value domains
  - ☐ numbers, booleans, strings, lists, sets, etc.
- ☐ They abstract away from evaluation order and use of temporaries (contrast with, e.g., stack machine)
- ☐ They may have unevaluated subexpressions (e.g. if)
- ✗ They may not be well-defined on all dynamic values
- ✗ They may be statically ill-formed

Today: Names (and Functions)

- Part of being a "high-level" language is letting the programmer name things:

variables	constants	types
functions	classes	modules
fields	operators	...

- Generically, we call names identifiers
- An identifier binding makes an association between the identifier and the thing it names
- An identifier use refers to the thing named
- The scope of a binding is the part of the program where it can be used

Scala Example

```
object Printer {
 def print(expr: Expr) : String = unparse(expr).toString()

  def unparse(expr: Expr) : SExpr = expr match {
    case Num(n) => SNum(n)
    case Add(l,r) => SList(SSym("+")::unparse(l)::unparse(r)::Nil)
    case Mul(l,r) => SList(SSym("*")::unparse(l)::unparse(r)::Nil)
    case Div(l,r) => SList(SSym("/")::unparse(l)::unparse(r)::Nil)
  }
}
```

binding use keyword

- Identifier syntax is language-specific
  - Usually unbounded sequence of alpha|numeric|symbol(?)
  - Further rules/conventions for different categories
- Identifiers are distinct from keywords! Some identifiers are pre-defined

Names for values

- Most languages let us bind names to values computed by expressions.
  - typically (maybe confusingly) called "variables"
- Why are variables useful?
  - ✗ In imperative languages, they are used to refer to memory cells that can be read or updated
  - They let us share expressions
    - to save repeated writing and, maybe, evaluation
  - They are needed to parameterize functions

Local Value Bindings

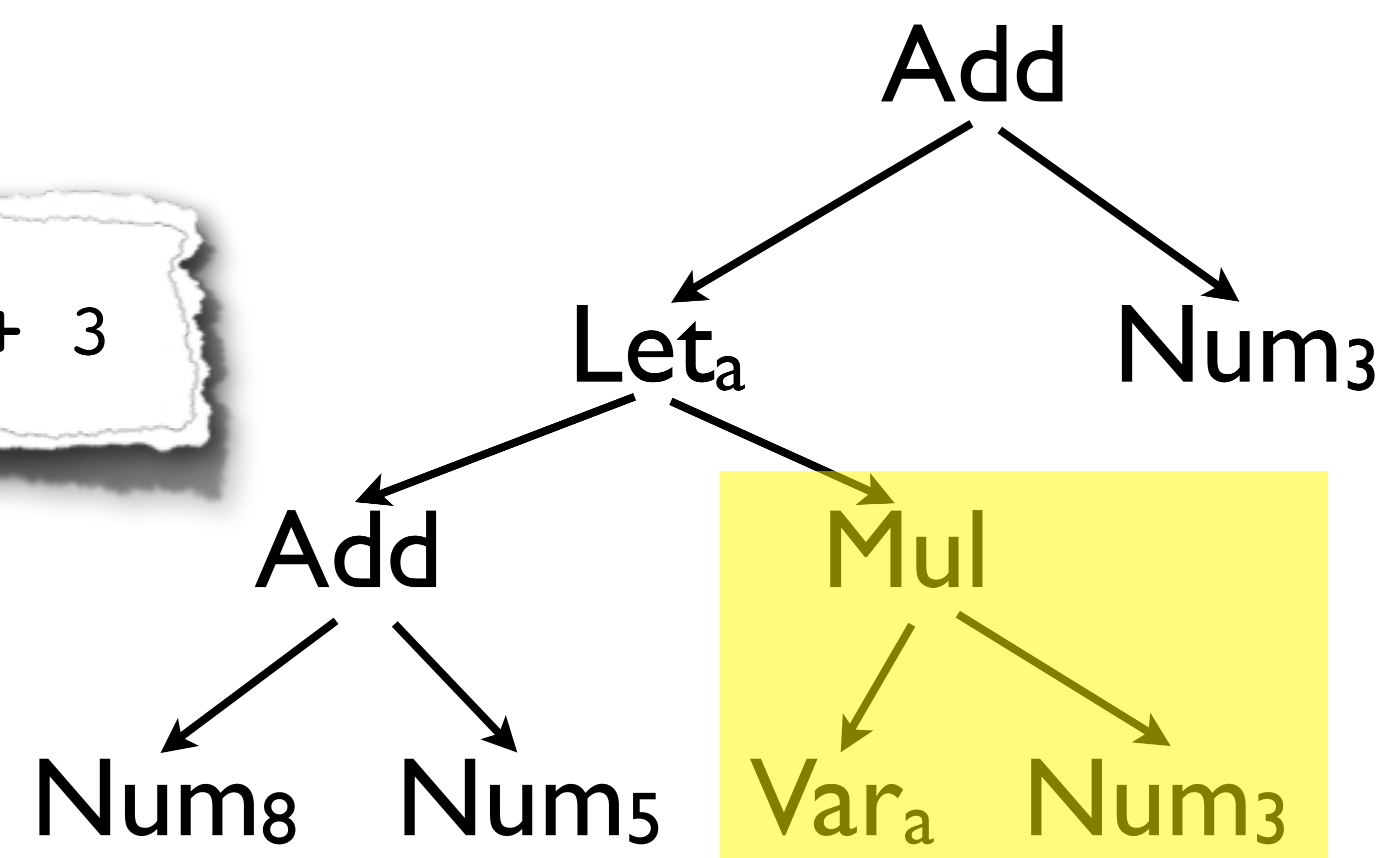
```
expr ::= num | expr + expr | ... |(expr)|
         id  | let id = expr in expr
```

scope

```
(let a = 8 + 5 in a * 3) + 3
```

binding

use

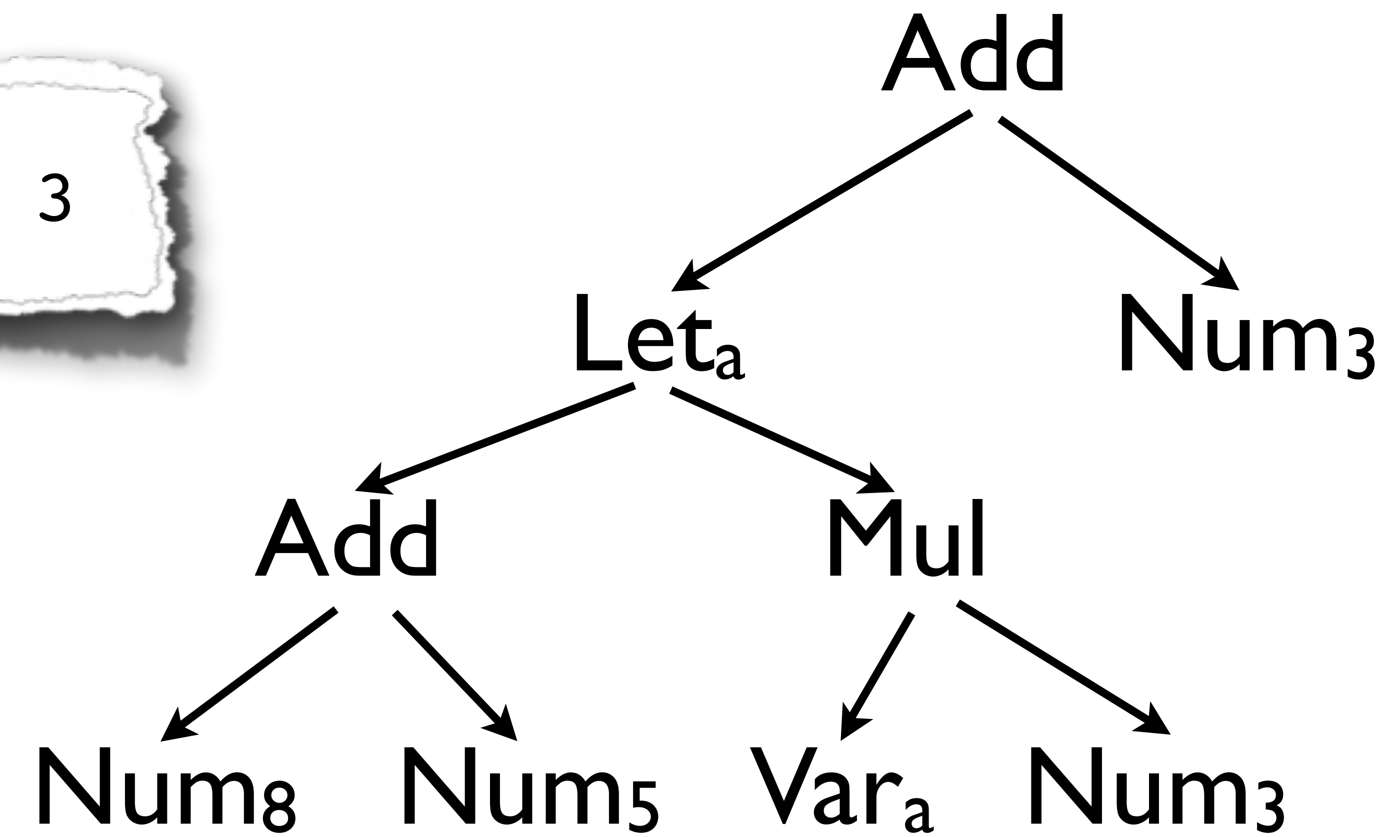


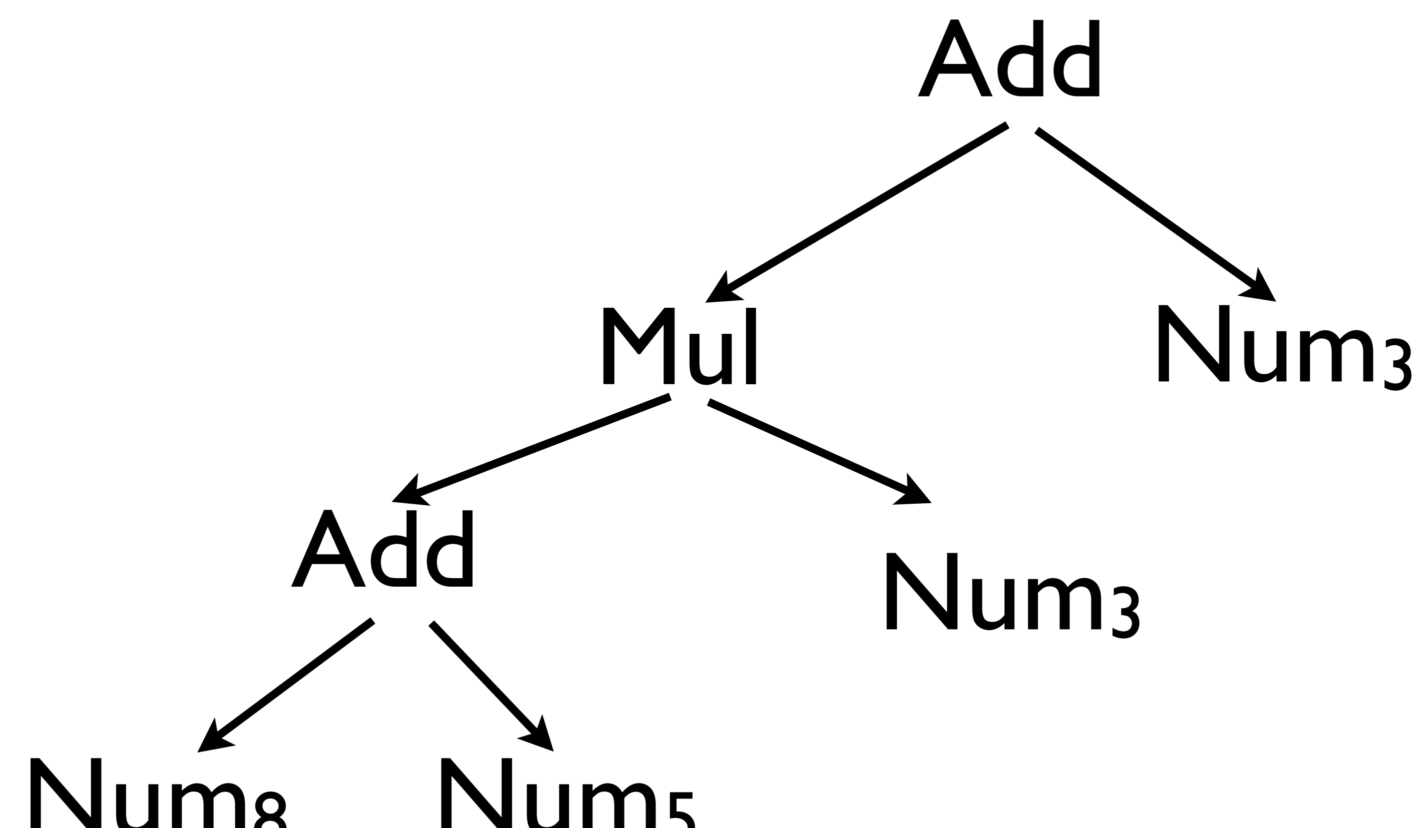
Semantics via Substitution

```
(let a = 8 + 5 in a * 3) + 3
```

“Rewrite the program text”

```
((8 + 5) * 3) + 3
```





Bound vs. Free

- A variable use *x* is **bound** if it appears in the scope of a binding for *x*
- Otherwise, it is **free**
- Bound and free are relative to an enclosing subexpression, e.g.

`a` is bound in `(let a = 8 + 5 in a * 3)`

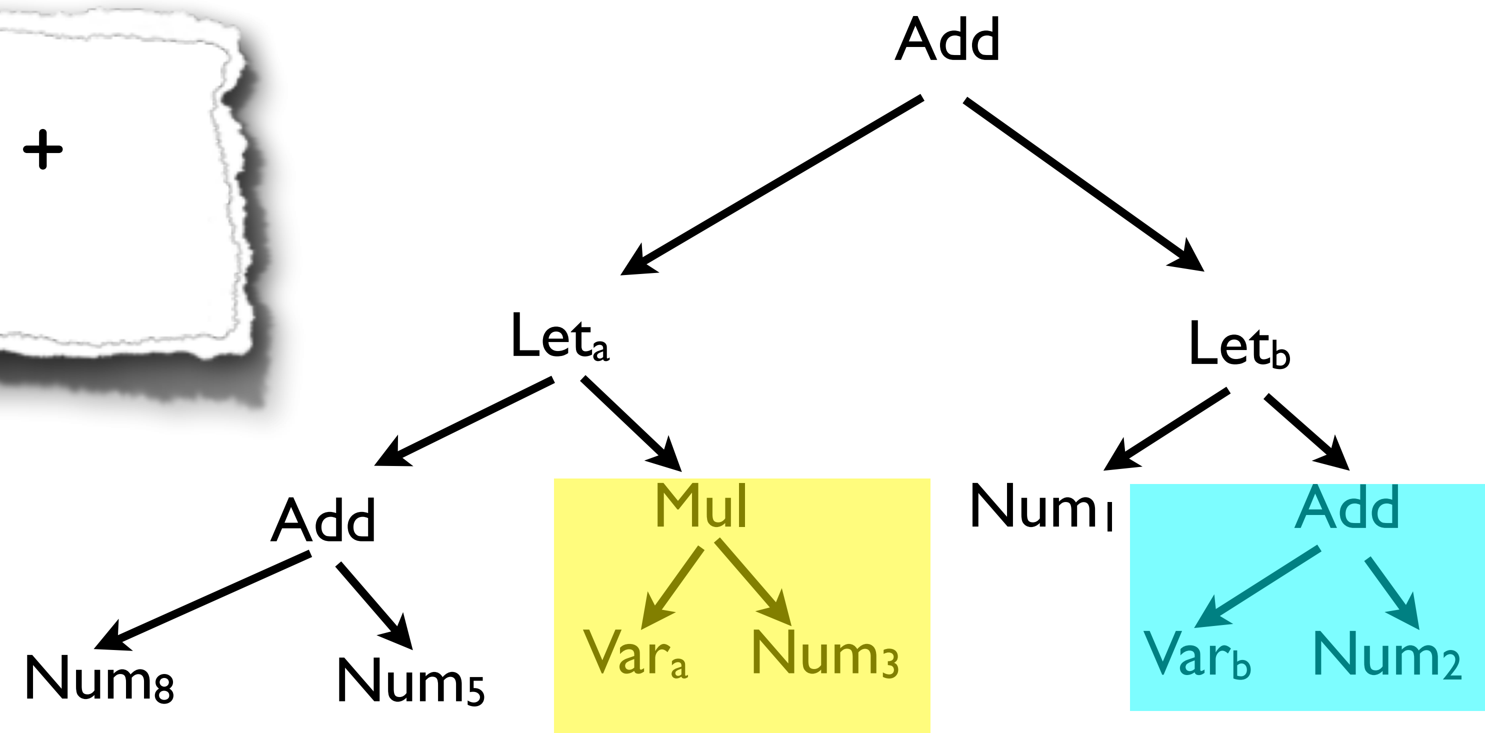
but free in `a * 3`

- We cannot evaluate a free variable

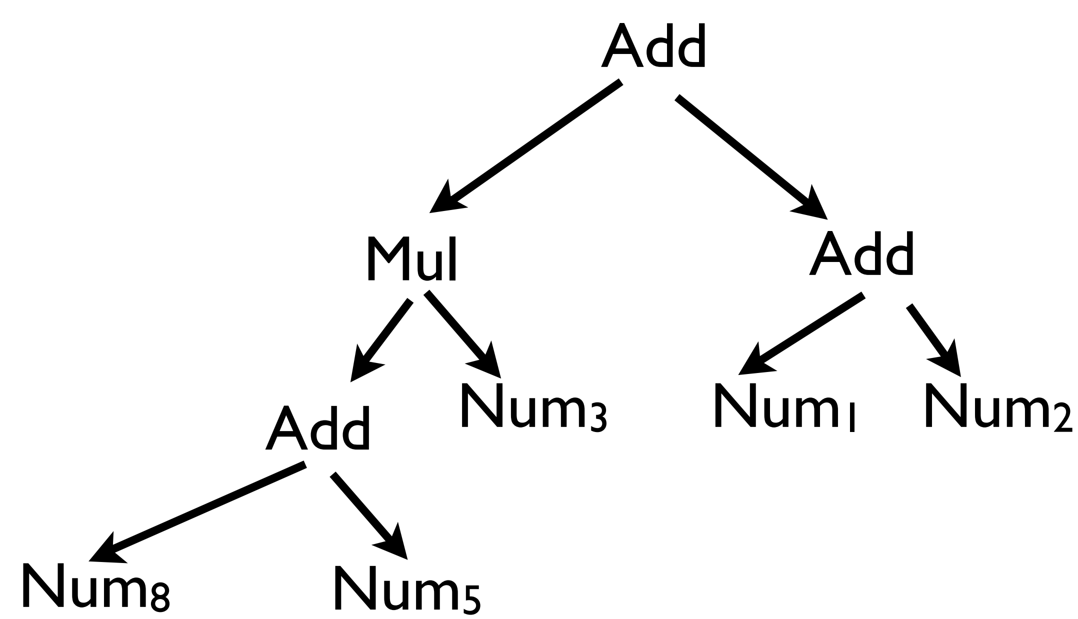
scope$_a$ scope$_b$

Parallel Scopes

```
(let a = 8 + 5 in a * 3) +
(let b = 1 in b + 2)
```



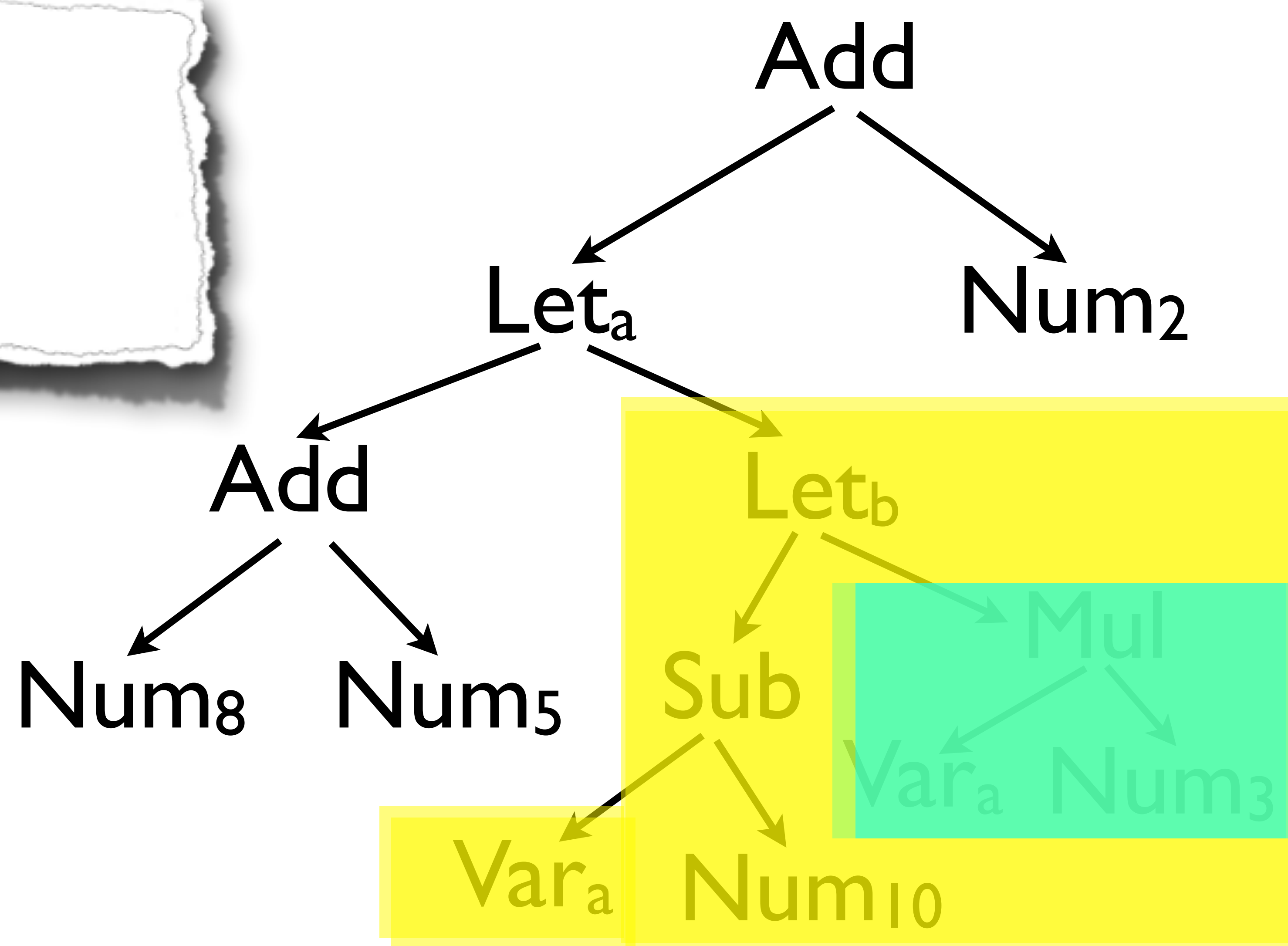

```
((8 + 5) * 3) +
(1 + 2)
```



What if both `let`'s bind a ?

Nested Scopes

```
(let a = 8 + 5 in
   let b = a - 10 in
      a * b) + 2
```



scope$_a$ scope$_b$

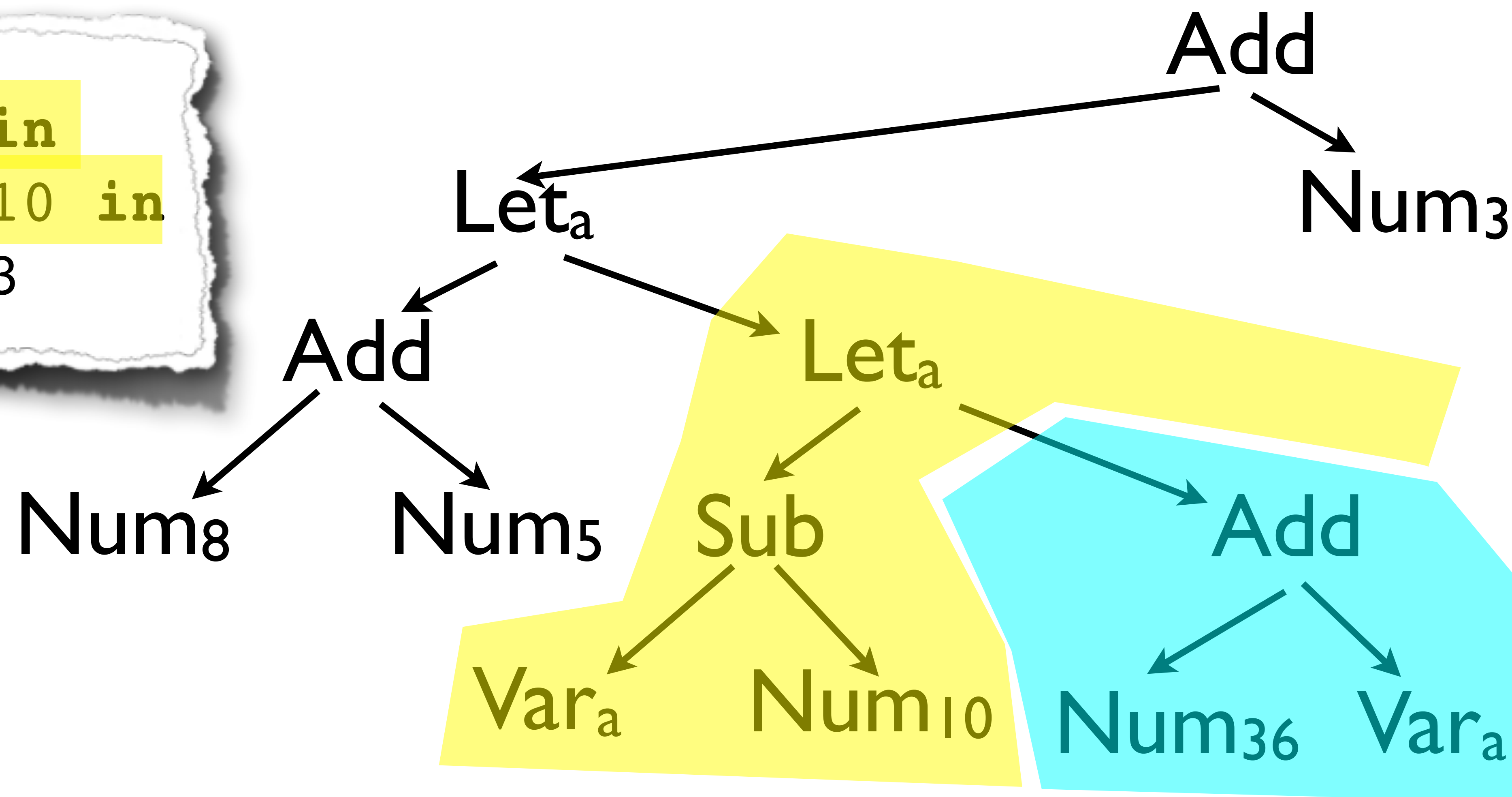
scope$_a$ scope$_{a\&b}$

Shadowing

scope$_a$

```
(let a = 8 + 5 in
   let a = a - 10 in
     36 + a) + 3
```

scope$_a$



Need more careful definition of substitution:

Don't substitute for variable *x* inside a nested let-binding for *x*

And that is still not quite good enough...see homework

Substitution Reconsidered

```
(let a = 8 + 5 in a * a) + 3
```

"Rewrite the program text"

```
((8 + 5) * (8 + 5)) + 3
```

Is this a good idea?

- It gives the expected answer
- But it doesn't reflect desired sharing of computations

Eager Evaluation Semantics

```
(let a = 8 + 5 in a * a) + 3
```

“Reduce body of let before substitution”

```
(let a = 13 in a * a) + 3
```

```
(13 * 13) + 3
```

Note that this isn’t always a win...

```
let a = do_giant_computation() in 42
```

Environments

- Substitutions are useful for giving high-level semantics
  - And they can be used to build interpreters
  - But these don't have a very "realistic" flavor
  - In conventional implementations, the program itself does not change during execution
- An alternative to substitution is to maintain an environment: a map from variables to values
  - Evaluating a let binding extends the env.
  - Evaluating a variable use looks up in the env.
  - Can think of this as a "deferred substitution"

Environment-based Interpreter

```
object Interp {
  type Env = Map[String,Value] // immutable map
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr,env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l,env),interpE(r,env)) …
    case Let(x,d,b) => {
      val v = interpE(d,env); interpE(b,env + (x -> v))
    }
    case Var(x) => (env get x) match {
      case Some(v) => v
      case None =>
        throw InterpException("Undefined variable:" + x)
    }
  // evaluate root of expression tree
  val v = interpE(expr,emptyEnv)
}
```

Environment-based Semantics

- Behavior of this interpreter relies on semantics of Scala's immutable maps
  - `Map[String,Value]()` creates a fresh empty map
  - `env + (x -> v)` creates a new map that is just like `env`, except that `x` is bound to `v`
  - `env get x` returns either `Some(v)` where `v` is the value bound to `x` or `None` if `v` is not bound
- This gives us eager evaluation and nestable local scopes with shadowing!

Procedures and Functions

- Procedures have a long history as an essential programming tool
  - Low-level view: subroutines give a way to avoid duplicating frequently-used code
  - High-level view: procedural abstraction lets us divide large programs into smaller pieces with hidden internals
- Procedures can be parameterized over values, types,…
- A function is just a procedure that returns a value
  - Or, conversely, a procedure is just a function whose result is uninteresting

Function parameterization

- Consider adding functions to our toy expression language
- To be useful in that context, a function must have one or more value parameters (Why?)
- We need value identifiers to name these parameters
  - The scope of a parameter is the function body
  - The value of each parameter is provided at the function call (or "application") site

Semantics via Substitution

Given a function declaration AST:

```
(f x (+ x 3))
```

function name

formal parameter

body

To evaluate a function application:

```
(@ f (* 13 3))
```

function name

actual parameter

We substitute a copy of the body for the application and then substitute the actual for the formal in that copy:

```
(+ (* 13 3) 3)
```

Call-by-name

- In this substitution semantics, the actual parameter is re-evaluated each time it used:

```
(f x (+ x x))
```

```
(@ f (* 2 3)) → (+ (* 2 3) (* 2 3))
```

- This semantics is known as call-by-name evaluation
- It duplicates work if a parameter is used twice
- But it saves work if a parameter is not used at all
- Even more useful is a variant called lazy evaluation, which evaluates each parameter at most once

Call-by-value

- Let’s switch back to a semantics based on value environments
  - Gives better match to conventional implementations
- Idea: to evaluate an application:
  - put bindings from actual parameters to formal parameters into the environment
  - then evaluate the function body in that environment
- But our environments map variables to values!
  - So we must evaluate the actual parameters to values first, before we add the new bindings to the environment
- This semantics is known as call-by-value evaluation

Hardware Calls

- A function is normally compiled to a machine-code subroutine
  - A single sequence of code that can be invoked from multiple places
  - Hardware gives support for remembering the return address to jump to when function is done
- Parameter values are typically passed in machine registers or on the stack
  - Fairly close match to environment model
- Call-by-value is most efficient choice at hardware level

Environment-based Interpreter

```
object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr,env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l,env),interpE(r,env)) …
    case App(f,a) => (functions get f) match {
      case Some((param,body)) => {
        val v = interpE(a,env)
        interpE(body,initialEnv + (param -> v))
      }
      case None => throw InterpException(…)
    }
    case Var(x) => (env get x) match …
  }
  // evaluate root of expression tree
  val v = interpE(expr,emptyEnv)
}
```

Environment-based Interpreter

```
object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr,env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l,env),interpE(r,env)) …
    case App(f,a) => (functions get f) match {
      case Some((param,body)) => {
        val v = interpE(a,env)  ??
        interpE(body,initialEnv + (param -> v))
      }
      case None => throw InterpException(…)
    }
    case Var(x) => (env get x) match …
  }
  // evaluate root of expression tree
  val v = interpE(expr,emptyEnv)
}
```

Environment-based Interpreter

```
object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr,env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l,env),interpE(r,env)) …
    case App(f,a) => (functions get f) match {
      case Some((param,body)) => {
        val v = interpE(a,env)    dangerous choice!
        interpE(body,env + (param -> v))
      }
      case None => throw InterpException(…)
    }
    case Var(x) => (env get x) match …
  }
  // evaluate root of expression tree
  val v = interpE(expr,emptyEnv)
}
```

"Dynamic scope"

- What should happen in the following program?

```
(f x (+ x y))
```

```
(@ f 42)
```

- How about this one?

```
(f x (+ x y))
```

```
(let y 2 (@ f 42))
```

- One possible answer: let the value of `y` "leak" into `f`
- But this is a bad idea! Why?

Environment-based Interpreter

```
object Interp {
  val emptyEnv = Map[String,Value]()
  def interpE(expr:Expr,env:Env) : Value = expr match {
    case Num(n) => NumV(n)
    case Add(l,r) => (interpE(l,env),interpE(r,env)) …
    case App(f,a) => (functions get f) match {
      case Some((param,body)) => {
        val v = interpE(a,env)    better choice!
        interpE(body,emptyEnv + (param -> v))
      }
      case None => throw InterpException(…)
    }
    case Var(x) => (env get x) match …
  }
  // evaluate root of expression tree
  val v = interpE(expr,emptyEnv)
}
```

“Static scope”/“Lexical scope”

- This program remains erroneous

```
(f x (+ x y))
```

```
(let y 2 (@ f 42))
```

- Looking at a function declaration, we can always determine if and where a variable is bound without considering the dynamic execution of the program!
- Some scripting languages still use dynamic scope, but as programs get larger, its dangers become obvious