CS558 Programming Languages

Fall 2023
Lecture 3a

Andrew Tolmach
Portland State University

Binding, Scope, Storage

- 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 a sequence of alpha|numeric|symbol(?)
  - May be further rules/conventions for different categories
- Identifiers are distinct from keywords!
  - Some identifiers are pre-defined (and can be re-defined)

Names, values, variables

- Most languages let us bind variable names to locations in the store that contain values
  - Name gives access to location for read or update
- Many languages also let us bind names directly to (immutable) values computed by expressions
  - Sometimes (confusingly) also called “variables”
  - This lets us share expressions
    - to save repeated writing and, maybe, evaluation

Scala `var` vs. `val`

Local Value Bindings

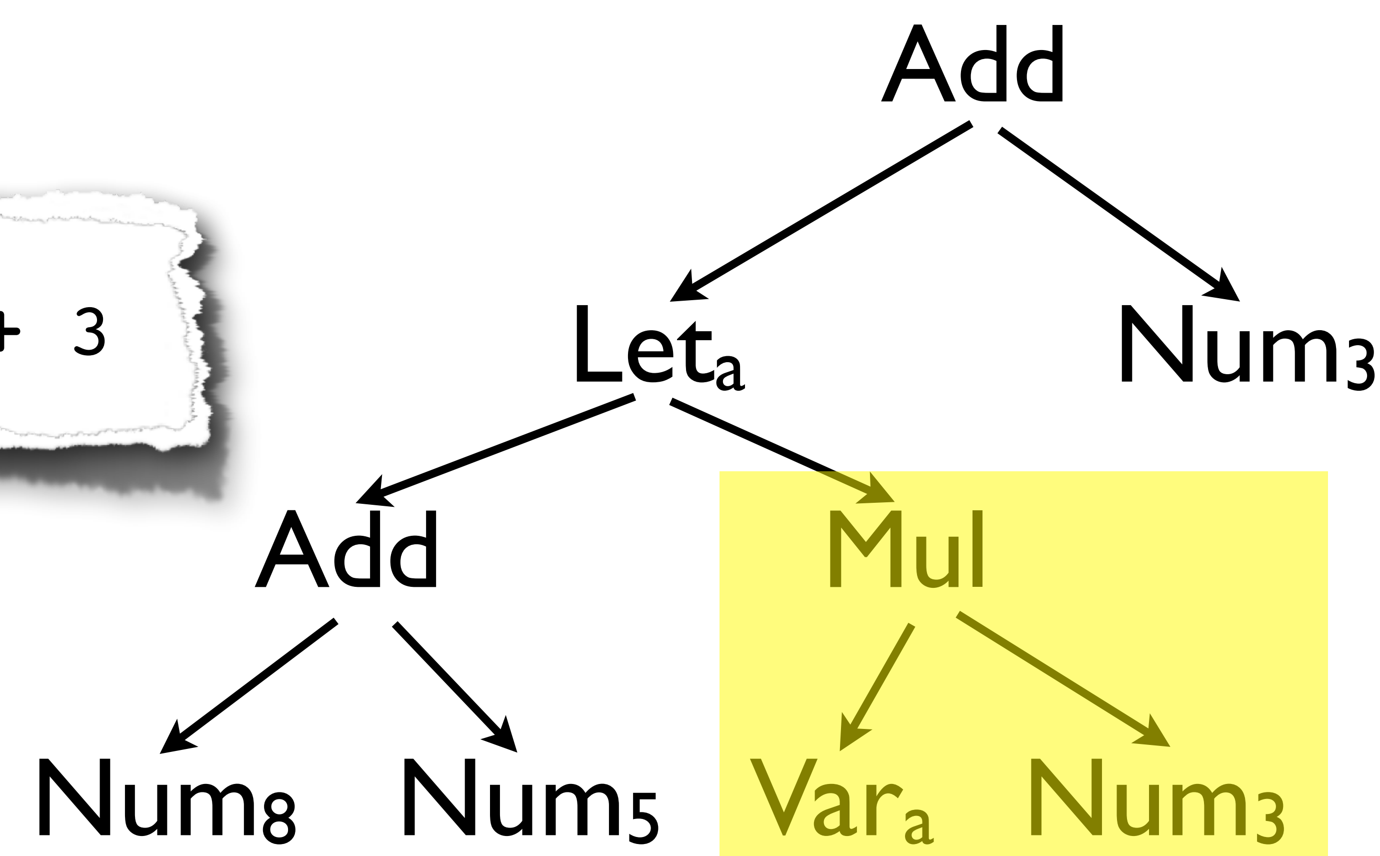
```
expr ::= num  |  expr + expr  |  ...  |(expr)|
         id   |  let id = expr in expr
```

scope

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

binding

use



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)`

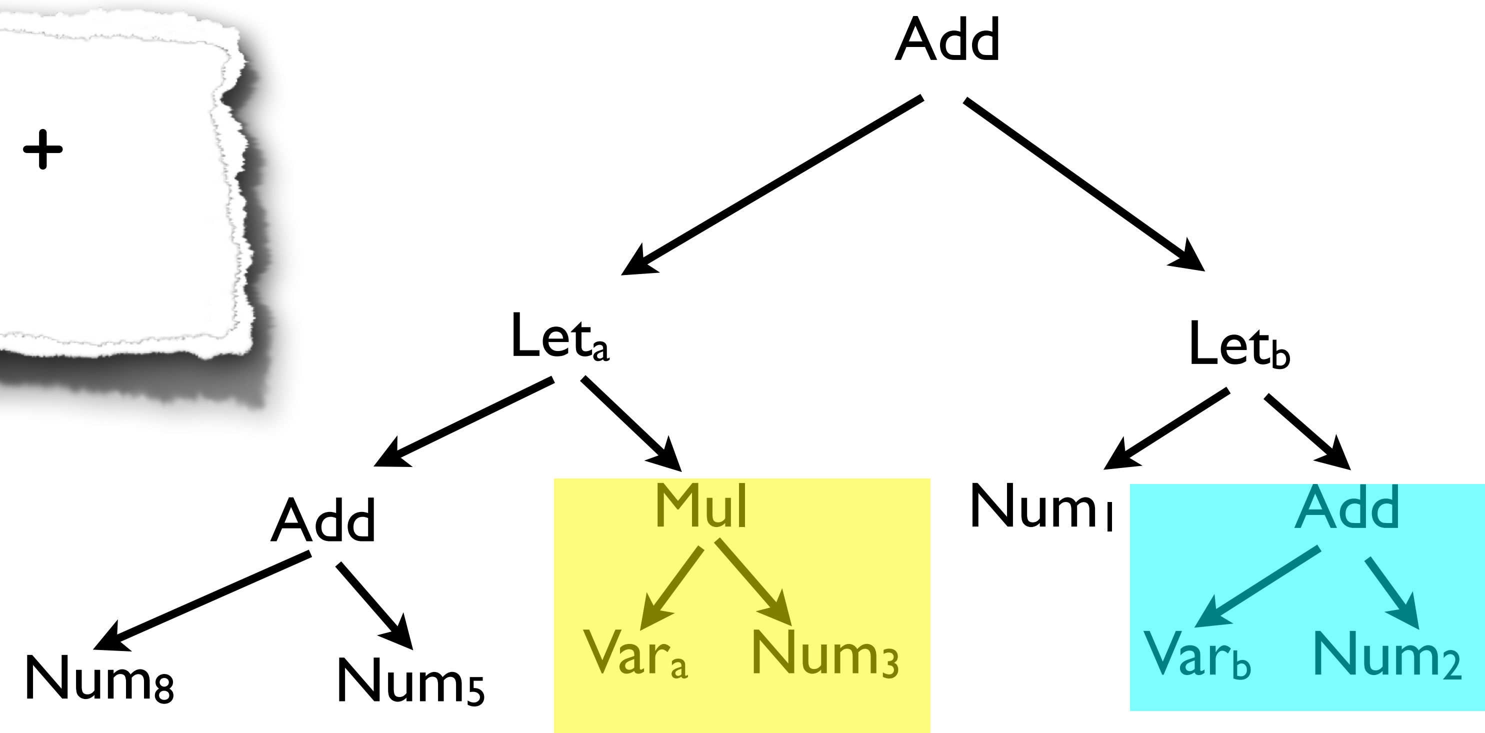
but free in `a * 3`

- 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)
```



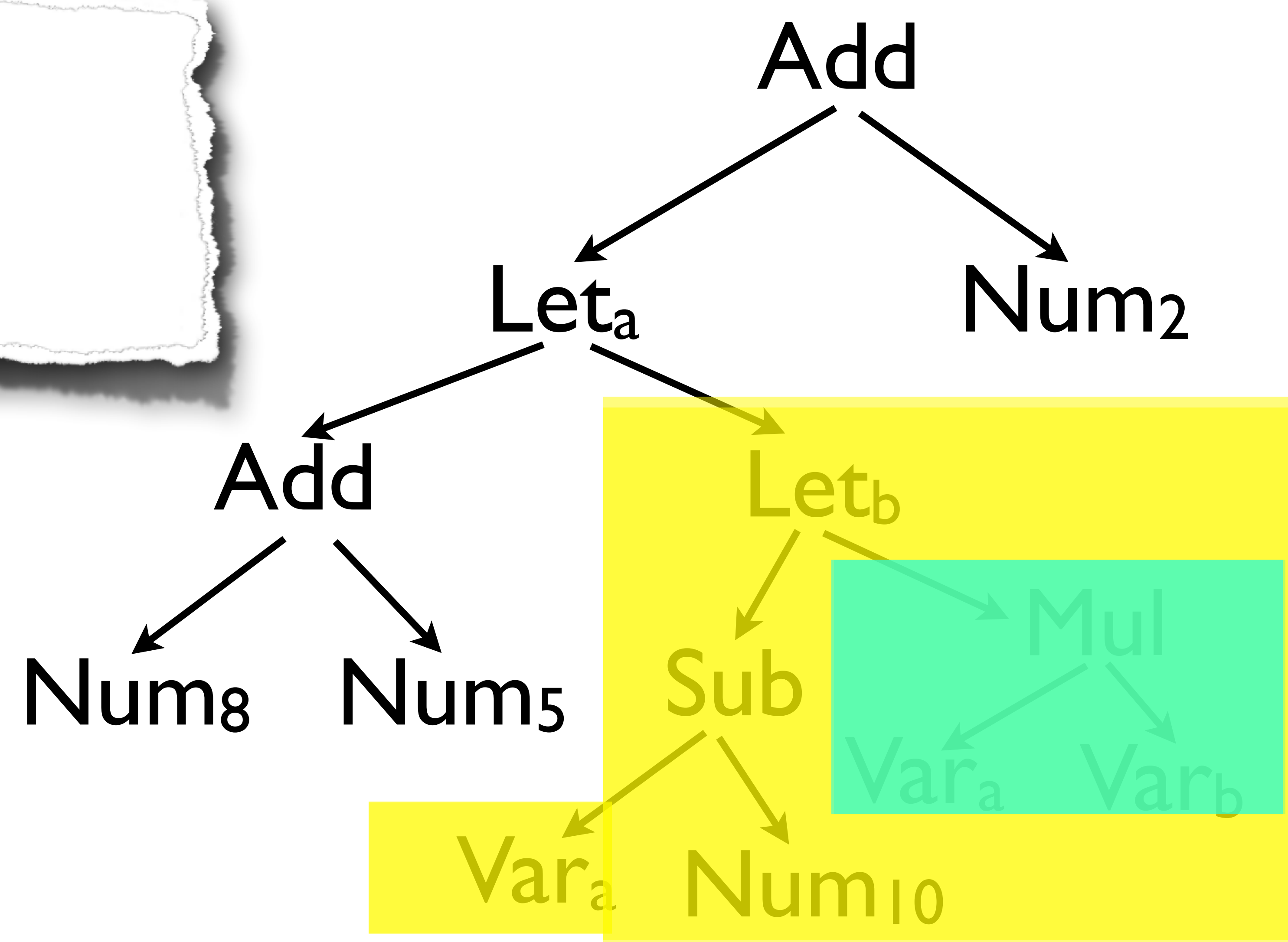
What if both `let`'s bind a ?

Nested Scopes

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

scope$_a$ scope$_b$

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

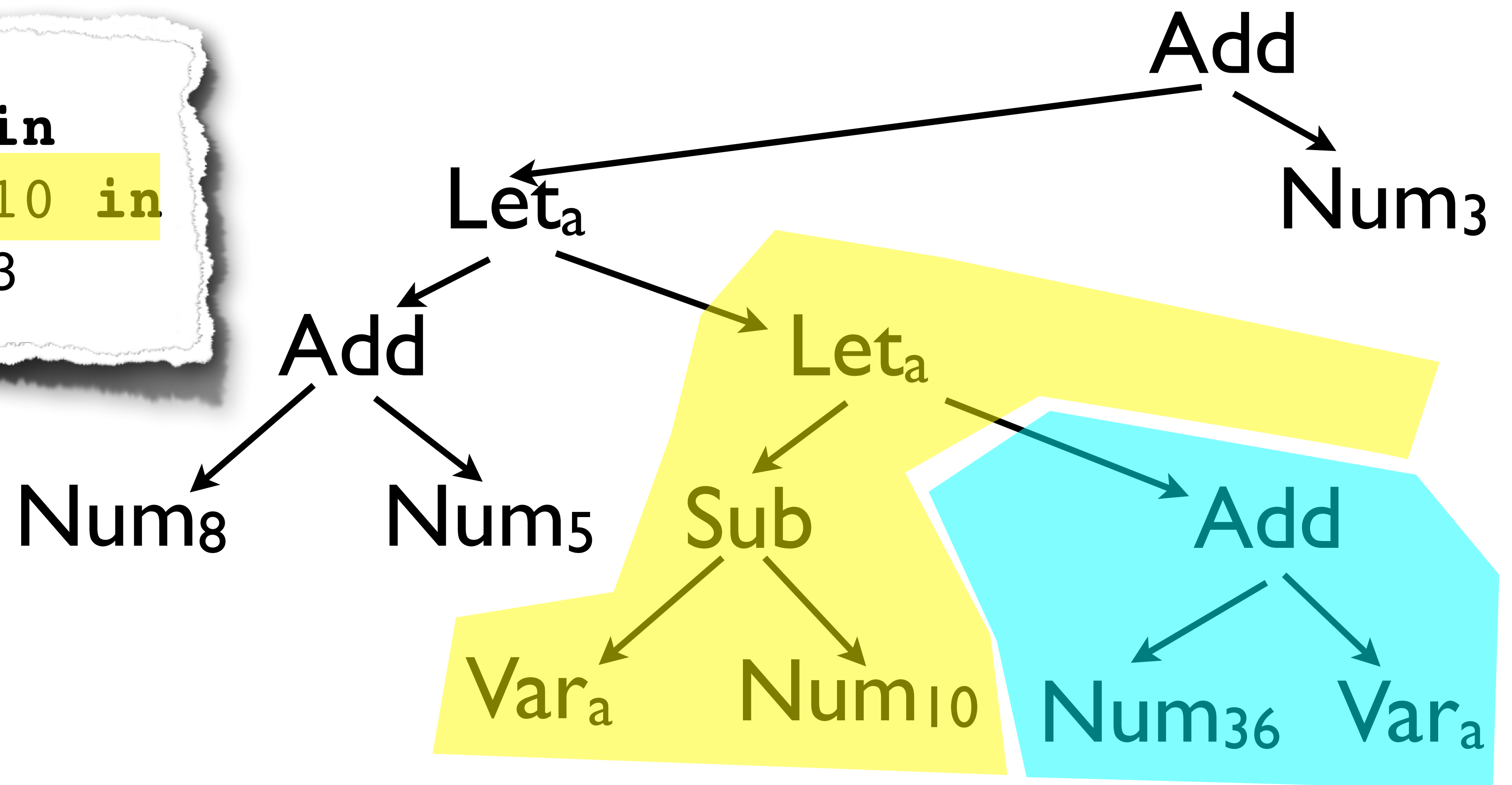


Shadowing

$scope_a$

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

$scope_a$

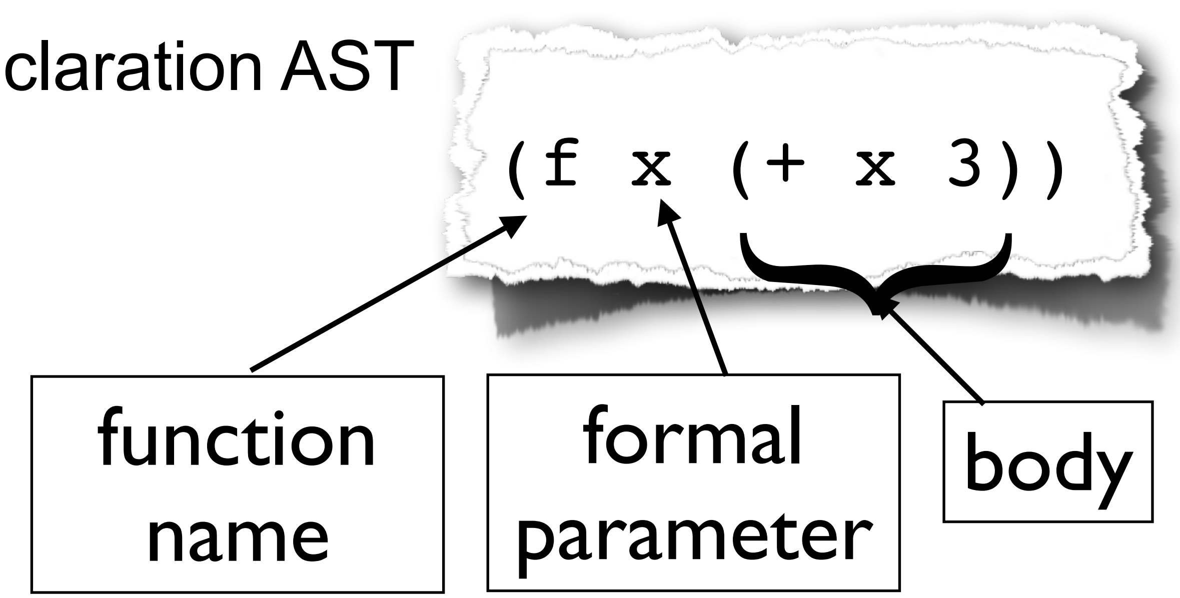


Common but not universal solution:
"Nearest enclosing binding" wins

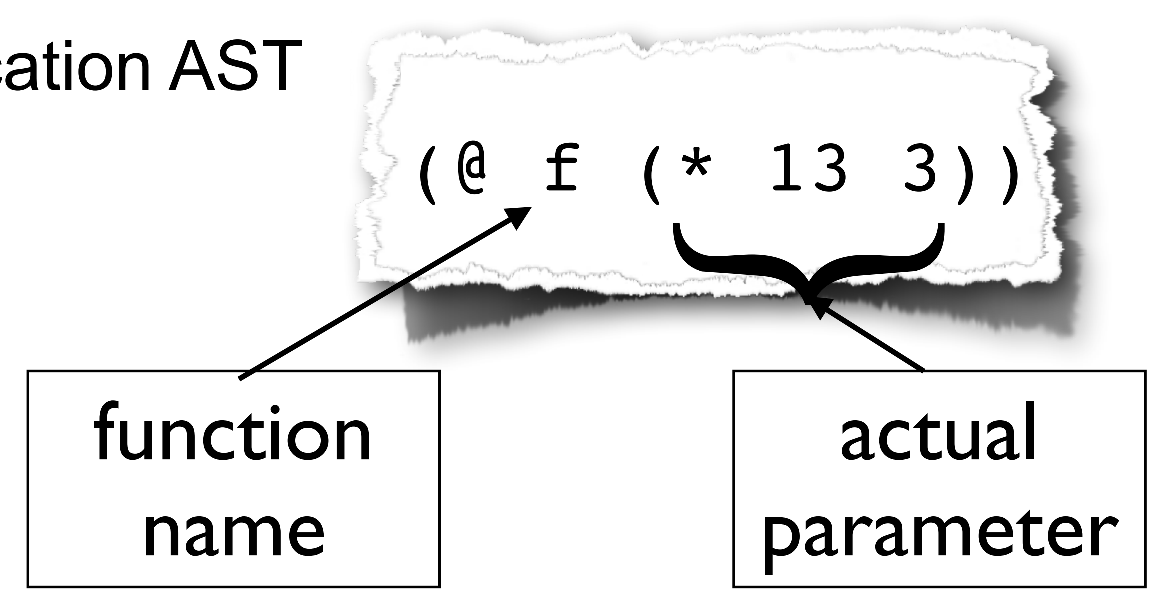
Functions and parameters

- Consider adding functions with parameters to our expression language
- We give names to 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

declaration AST



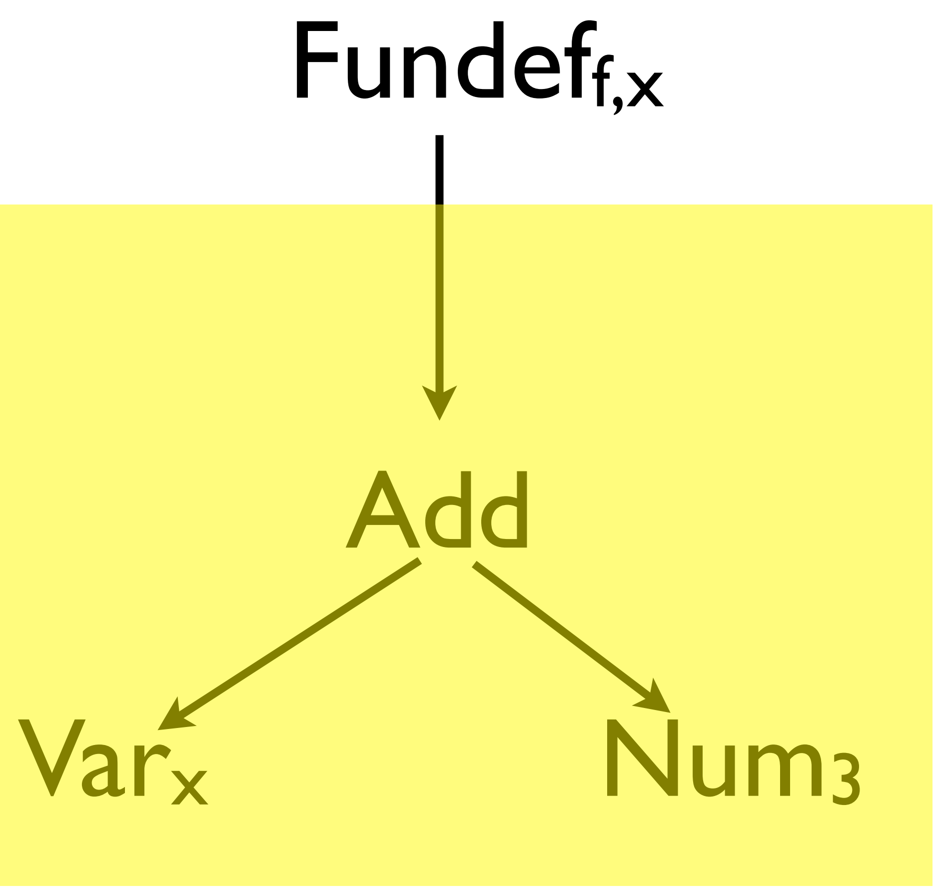
application AST



Function parameter scoping

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

$scope_x$



Function Name Scoping

- Typically, we want to allow functions to be recursive
- Scope of function's name includes its own body

scope_f

```
letfun f x = if x = 0 then 1 else x*f(x-1)
in f(42)
```

Mutually Recursive Definitions

```
letfun f(x) = g(x + 1)
and    g(y) = f(y - 1)
in
f(2) + g(4)
```

$scope_{f,g}$

Many earlier languages were designed to be compiled by a single pass through the source code and therefore use forward declarations

```
void g (double y);  /* declares g but doesn't define it */
void f(double x) { g(x+1.0); }
void g(double y) { f(y-1.0); }   /* definition is here */
```

C

In some languages, all top-level definitions are (implicitly) treated as mutually recursive.

"Dynamic Scope"

What should happen in the following program?

```
letfun f(x) = x + y
in f(42)
```

How about this one?

```
letfun f(x) = x + y
in let y = 2
   in f(42)
```

One possible answer: let the value of `y` "leak" into `f`

This is an example of "dynamic scope"   Bad idea!

Why?

“Static scope”/“Lexical scope”

Better if this code is considered to have an error

```
letfun f(x) = x + y
in let y = 2
   in 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

Aside: Erroneous Programs

- Important part of language specification is distinguishing valid from invalid programs
- Useful to define three classes of errors that make programs invalid:
  - Static errors
  - Checked run-time errors
  - Unchecked run-time errors
- Of course, even valid programs may not act as the programmer intended!

Static Errors

- Static errors can be detected before the program is run (at compile or pre-interpretation time)
  - Includes lexical errors, syntactic errors, type errors, etc.
  - Error checker can give precise feedback about erroneous location in source code
  - Language semantics are usually defined only for programs that have no static errors

Checked Run-time Errors

- Checked run-time errors are violations that the language implementation is required to detect and report at run time, in a clean way.
  - E.g. in Scala or Java: division by 0, array bounds violations, dereferencing a null pointer
  - Depending on language, might:
    - cause an error message and abort
    - raise an exception (which in principle can be caught by program)
  - Language semantics must specify what run-time errors are checked and how

Unchecked Run-time Errors

- Unchecked run-time errors are violations that the implementation does not have to detect.
  - Subsequent behavior of the computation is arbitrary (language semantics typically silent about this)
  - No “fail-stop” behavior: error might not be manifested until long after it occurs
  - E.g. in C: division by 0, array bounds violations, dereferencing a null pointer, signed integer overflow, unsequenced assignments, etc.
  - Safe languages like Scala, Java, Python have no such errors!

Re-using names

- What happens when the same name is bound twice in the same scope?

- If the bindings are to different kinds of things (e.g. types vs. variables), can often disambiguate based on syntax, so no problem arises (except maybe readability):

```
type Foo = Int
val Foo : Foo = 10
val Bar : Foo = Foo + 1
```

Scala

- Here we say that types and variables live in different name spaces

- If the bindings are in the same namespace, typically an error. But sometimes additional info (such as types) can be used to pick the right binding; this is called overloading

Named scopes: modules, classes

- Often, the construct that delimits a scope can itself has a name, allowing the programer to manage explicitly the visibility of the names inside it

OCaml modules

```
module Env = struct
  type env = (string * int) list
  let empty : env = []
  let rec lookup (e:env) (k:string) : int = ...
end
let e0 : Env.env = Env.empty in Env.lookup e0 "abc"
```

Java classes

```
class Foo {
  static int x;
  static void f(int x);
}
int z = Foo.f(Foo.x)
```

Semantics via Environments

- An environment is a mapping from names to their bindings
  - The environment at a program point describes all the bindings in scope at that point
  - Environment is extended when binding constructs are evaluated
  - Environment is consulted to determine the meaning of names during evaluation

Environments for everything

- Environments can hold binding information for all kinds of names
  - a variable name is (typically) bound to location [in the store] containing the variable
  - a value (constant) name may be bound directly bound to the value [environment = store]
  - a function name is bound to description of the function’s parameters and body
  - a type name is bound to a type description, including the layout of its values
  - a class name is bound to a list of the class’s content
  - etc.

Variables, Environment, Store

- In most imperative languages, variable names are bound to locations, which in turn contain values.
  - So creating a variable involves two things:
    1. allocating a new store location (and possibly initializing its contents)
    2. updating the environment to create a new binding from the variable name to that location
  - For simplicity, we sometimes elide the difference between the environment and the store, and think of names as being bound directly to values (i.e. names **are** locations)
    - This works unless multiple names are aliased to a single location; more about this later

Initialization Values

- Many languages require variables to be declared before they are used: this gives them a scope, perhaps a type, and (maybe) an initial value given by an expression

- Whether or not declarations are required, it is surely an error to use any variable as an r-value unless it has been previously assigned a value.

  - But many languages let us write such code, resulting in runtime errors—either checked (e.g. as in Python) or unchecked (e.g. as in C)

- Simplest fix is to require an initial value to be given for every declared variable (e.g. as in Scala)

Checking Initialization

- Java takes a more sophisticated approach
  - variables do not need to be initialized at the point of declaration, but
  - they must be initialized before they are used; otherwise a static error occurs

```
int a;
if (b)  /* b is boolean */
  a = 3;
else
  a = 4;
a = a + 1;
```

a legal Java program

But checking initialization before use is uncomputable in general! (Why?)

Definite Assignment

- So the Java definition carefully details a conservative, computable, set of conditions, which every program must meet, that guarantee the absence of uses before definition.

- This is called the definite assignment property; just defining it takes 16 pages of the reference manual.

Having these rules in the Java definition ensures portability

Being conservative means that some programs that actually do initialize before use will be rejected

```
int a;
if (b)
  a = 3;
if (!b)
  a = 4;
a = a + 1;
```

an illegal Java program