Interpreters

Implementing PLs

Most of the course is learning fundamental concepts for *using* PLs

- Syntax vs. semantics vs. idioms
- Powerful constructs like closures, first-class objects, iterators (streams), multithreading, …

An educated computer scientist should also know some things about *implementing* PLs

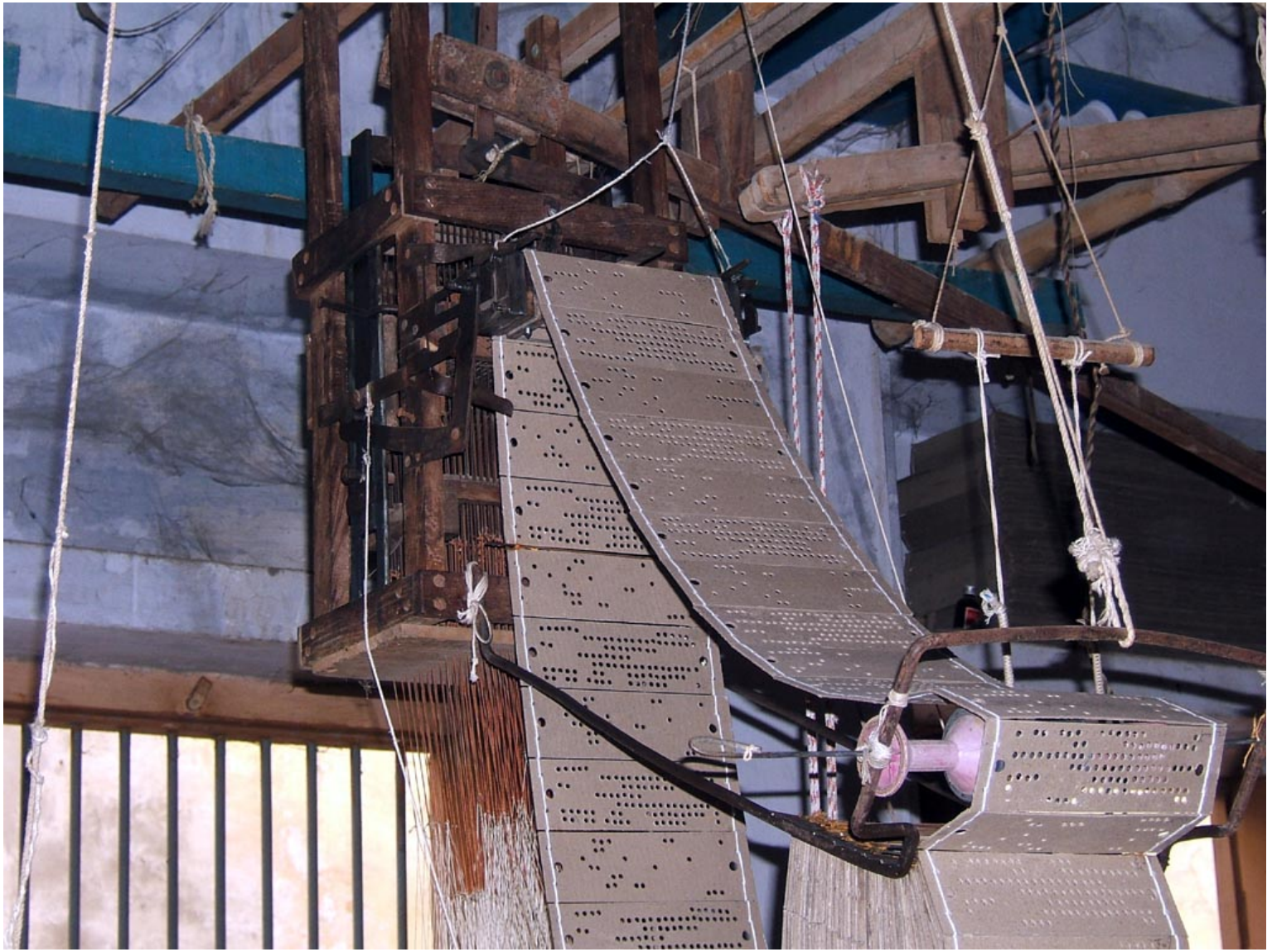
- Implementing something requires fully understanding its semantics
- Things like closures and objects are not "magic"
- Many programming tasks are like implementing PLs
  - Example: rendering a document ("program" is the [structured] document and "pixels" is the output)

Ways to implement a language

Two fundamental ways to implement a programming language *X*

- Write an interpreter in another language *Y*
  - Better names: evaluator, executor
  - Immediately executes the input program as it's read
- Write a compiler in another language *Y* to a third language *Z*
  - Better name: translator
  - Take a program in *X* and produce an equivalent program in *Z*.

First programming language?



First programming language?

Interpreters vs compilers

- Interpreters
  - Takes one "statement" of code at a time and executes it in the language of the interpreter.
  - Like having a human interpreter with you in a foreign country.
- Compilers
  - Translate code in language X into code in language Z and save it for later. (Typically to a file on disk.)
  - Like having a person translate a document into a foreign language for you.

Reality is more complicated

Evaluation (interpreter) and translation (compiler) are your options

- But in modern practice we can have multiple layers of both

A example with Java:

- Java was designed to be platform independent.
  - Any program written in Java should be able to run on any computer.
- Achieved with the "Java Virtual Machine"
  - An idealized computer for which people have written interpreters that run on "real" computers.

Example: Java

- Java programs are compiled to an "intermediate representation" called bytecode.
  - Think of bytecode as an instruction set for the JVM.
- Bytecode is then interpreted by a (software) interpreter in machine-code.
- Complication: Bytecode interpreter can compile frequently-used functions to machine code if it desires.
- CPU itself is an interpreter for machine code.

Sermon

Interpreter versus compiler versus combinations is about a particular language **implementation**, not the language **definition**

So clearly there is no such thing as a "compiled language" or an "interpreted language"

- Programs cannot "see" how the implementation works

Unfortunately, you hear these phrases all the time

- "C is faster because it's compiled and LISP is interpreted"
- Nonsense: I can write a C interpreter or a LISP compiler, regardless of what most implementations happen to do
- Please politely correct your bosses, friends, and other professors

Okay, they do have one point

In a traditional implementation via compiler, you do not need the language implementation (the compiler) to run the program

- Only to compile it
- So you can just "ship the binary"

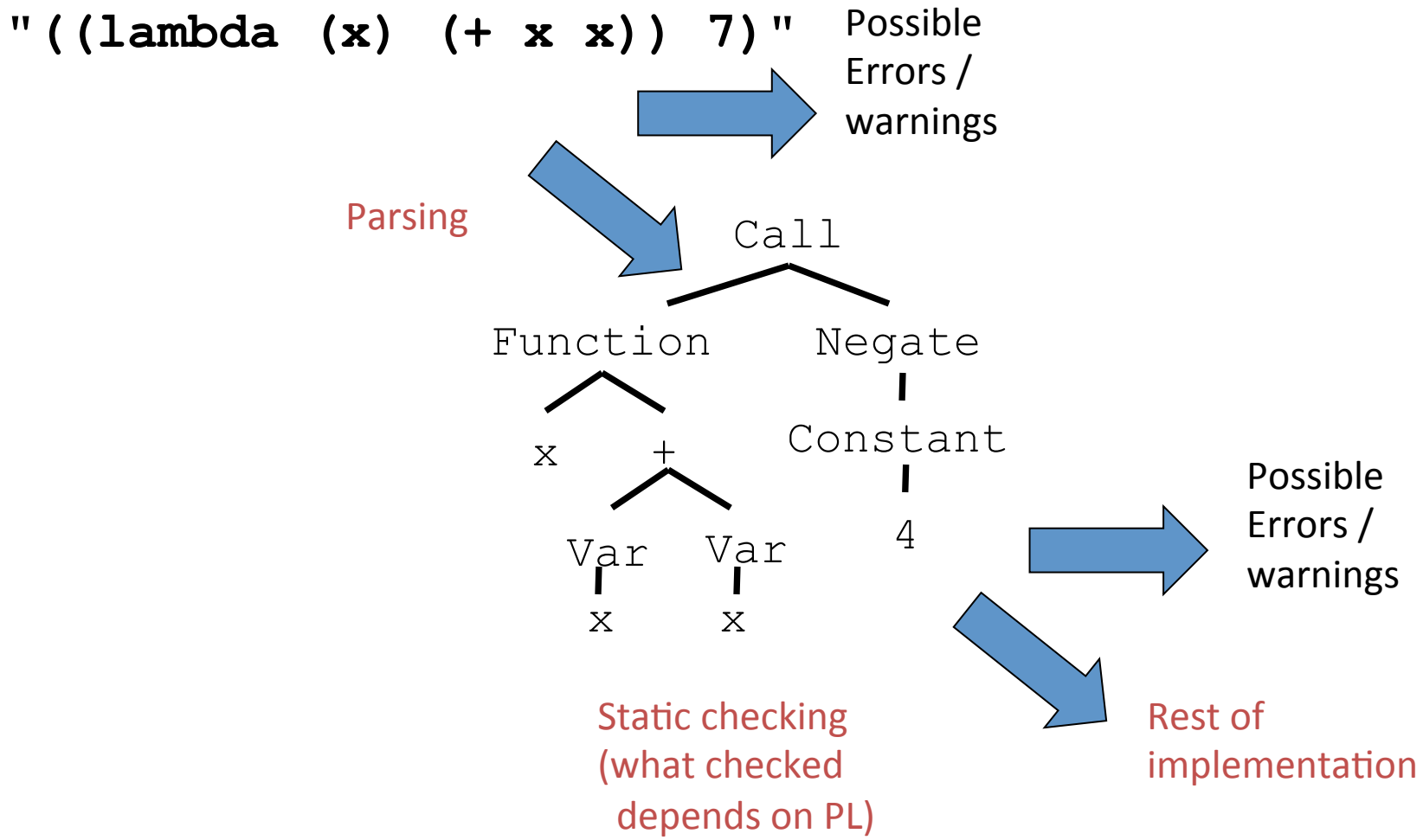
But Racket, Scheme, LISP, Javascript, Ruby, … have **`eval`**

- At run-time create some data (in Racket a list, in Javascript a string) and treat it as a program
- Then run that program
- Since we don't know ahead of time what data will be created and therefore what program it will represent, we need a language implementation at run-time to support **`eval`**
  - Could be interpreter, compiler, combination

Digression

- Eval/Apply
  - Built into Racket, traditionally part of all LISP-ish interpreters
- Quote
  - Also built-in
  - Happens behind the scenes when you use the single quote operator: '

Back to implementing a language



Skipping those steps

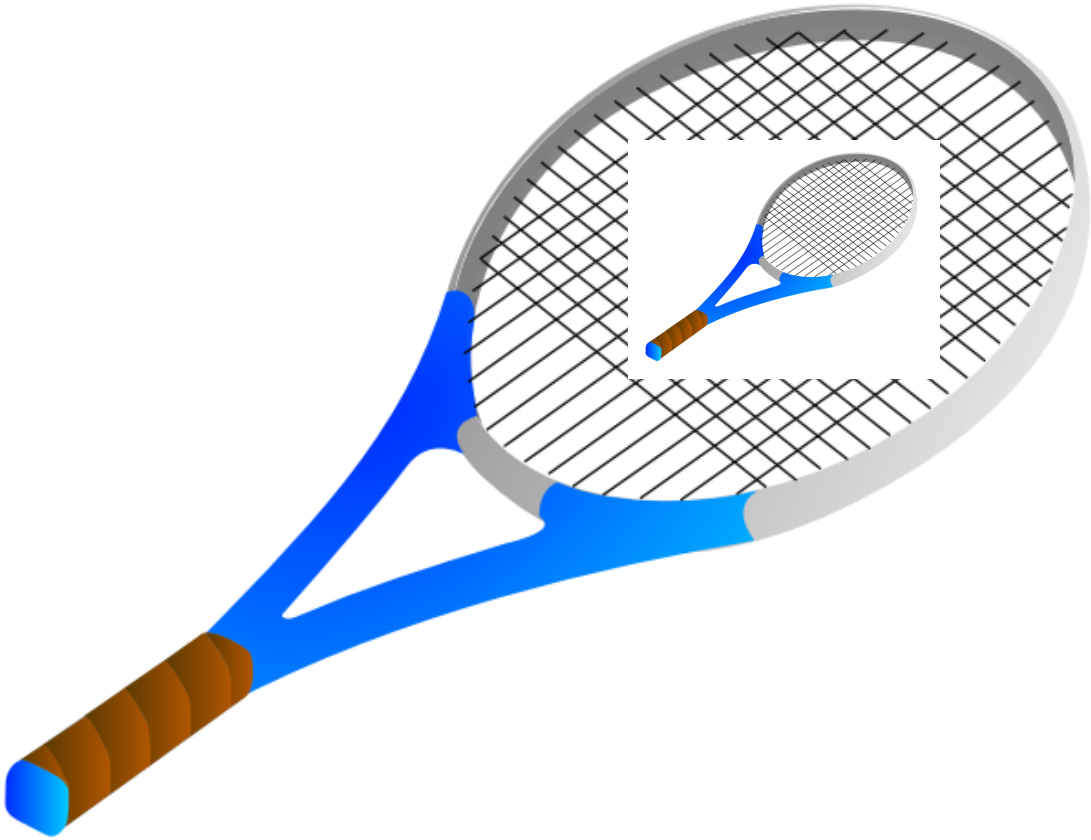
If language to be interpreted (X) is very close to the interpreter language (Y), then take advantage of this!

- Skip parsing? Maybe Y already has this.
- These abstract syntax trees (ASTs) are already ideal structures for passing to an interpreter

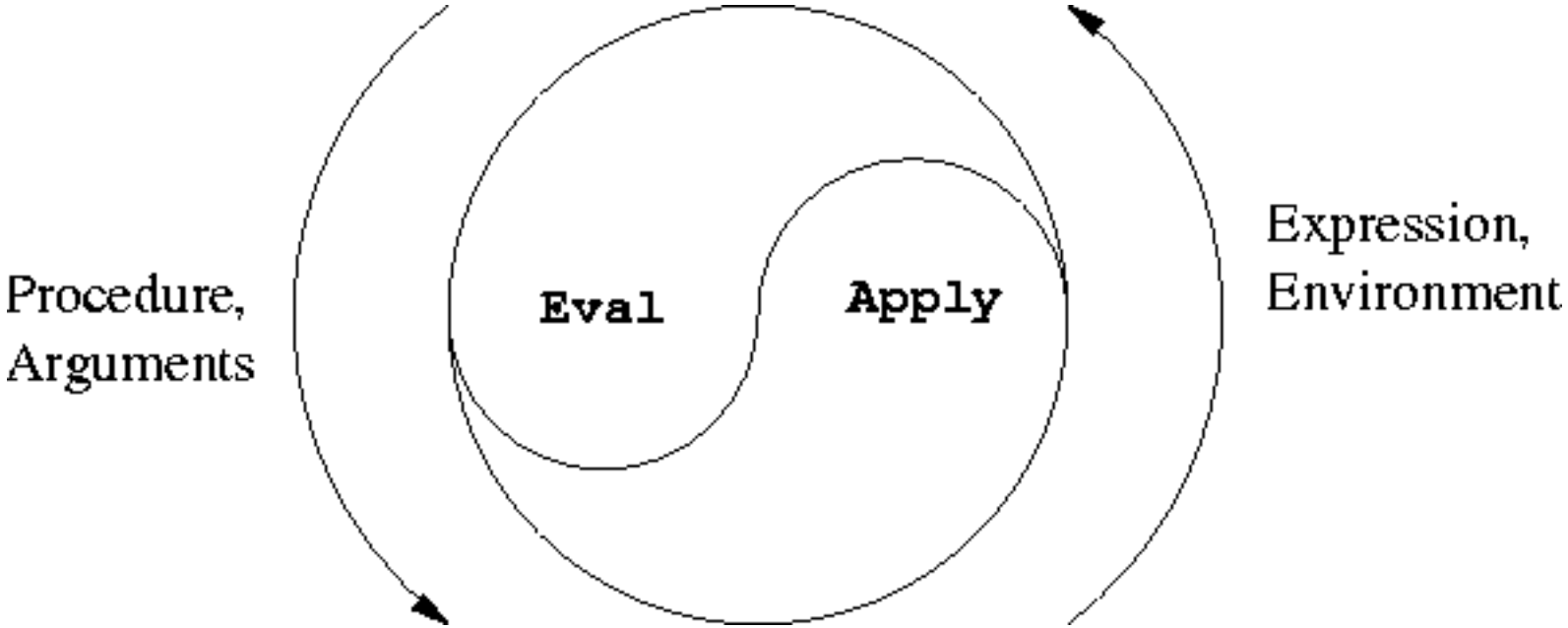
We can also, for simplicity, skip static checking

- Assume subexpressions are actually subexpressions
  - *Do not* worry about **`(add #f "hi")`**
- For dynamic errors in the embedded language, interpreter can give an error message (e.g., divide by zero)

Write Racket in Racket



Heart of the interpreter



- Mini-Eval: Evaluates an expression to a value (will call apply to handle functions)
- Mini-Apply: Takes a function and argument values and evaluate its body (calls eval)

```
(define (mini-eval expr env)
```
    is this a ___ expression?
    if so, then call our special handler
        for that type of expression.
```
)
```
What kind of expressions will we have?

- numbers
- variables (symbols)
- math functions +, -, *, etc
- others as we need them

- How do we evaluate a (literal) number?

- Just return it!

- Psuedocode for first line of math-eval:
  - If this expression is a number, then return it.

- How do we handle `(add 3 4)`?

- Need two functions:
  - One to detect that an expression is an addition.
  - One to evaluate the expression.

`(add 3 4)`

- Is this an expression an addition expression?

```
(equal? 'add (car expr))
```

- Evaluate an addition expression:

```
(+ (cadr expr) (caddr expr))
```

You try

- Add subtraction (e.g., sub)
- Add multiplication (mul)
- Add division (div)
- Add exponentiation (exp)
- It's *your* programming language, so you may name these commands whatever you want.

(add 3 (add 4 5))

- Why doesn't this work?

(add 3 (add 4 5))

- How *should* our language evaluate this sort of expression?
- We could forbid this kind of expression.
  - Insist things to be added always be numbers.
- Or, we could allow the things to be added to be expressions themselves.
  - Need a recursive call to math-eval inside eval-add.

You try

- Fix your math commands so that they will recursively evaluate their arguments.

Adding Variables

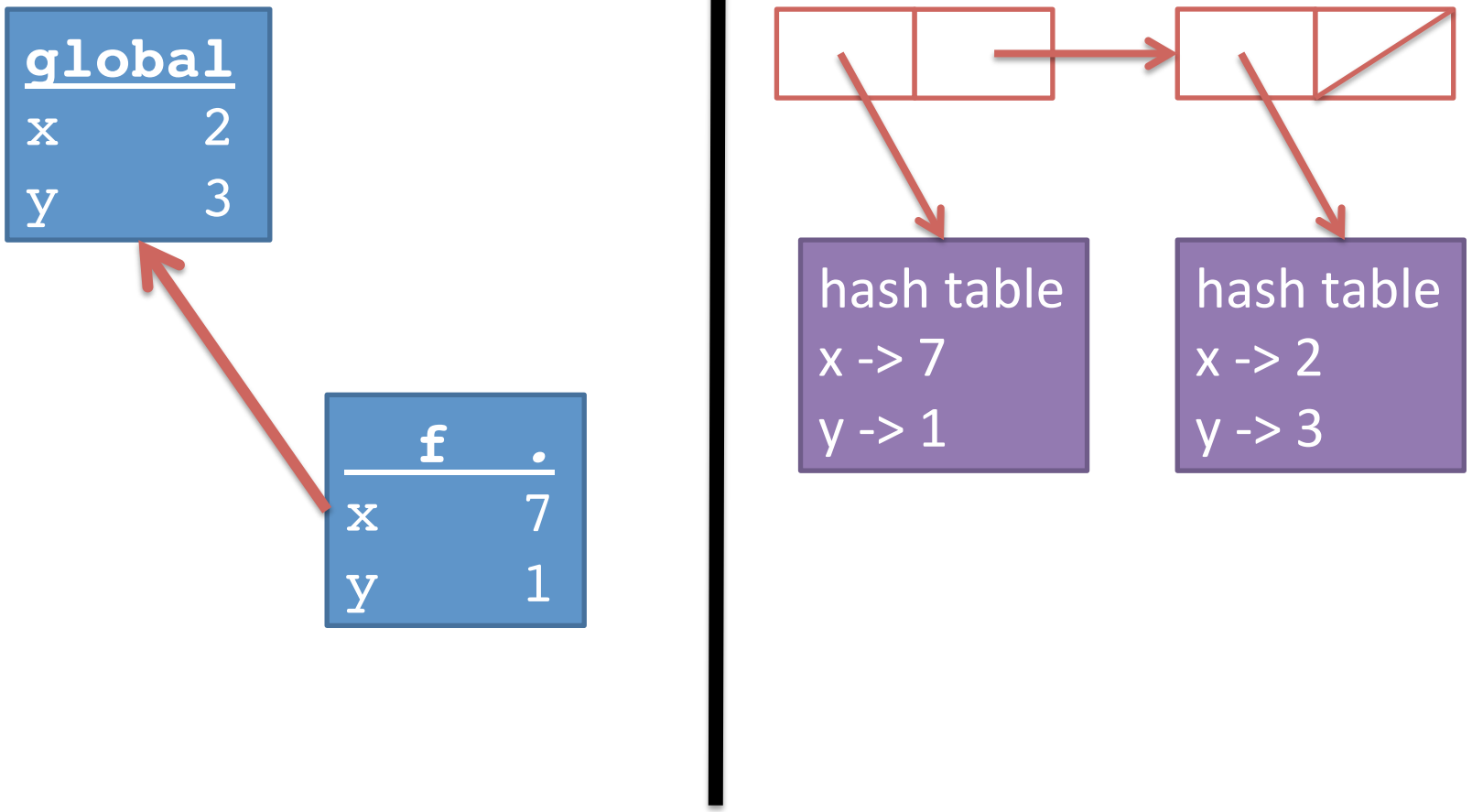
Implementing variables

- Represent a *frame* as a hashtable.
- Racket's hashtables:

```
(define ht (make-hash))
(hash-set! ht key value)
(hash-has-key? ht key)
(hash-ref ht key)
```

Implementing variables

- Represent an environment as a list of frames.



Implementing variables

- Two things we can do with a variable in our programming language:
  - Define a variable
  - Get the value of a variable

Getting the value of a variable

- New type of expression: a symbol.
- Whenever math-eval sees a symbol, it should look up the value of the variable corresponding to that symbol.

Getting the value of a variable

```
(define (lookup-variable-value var env)
  ; Pseudocode:
  ; If our current frame has the variable bound,
  ;   then get its value and return it.
  ; Otherwise, if our current frame has a frame
  ;   pointer, then follow it and try the lookup
  ;   there.
  ; Otherwise, throw an error.
```

Getting the value of a variable

```
(define (lookup-variable-value var env)
  (cond ((hash-has-key? (car env) var)
         (hash-ref (car env) var))
        ((not (null? env))
         (lookup-variable-value var (cdr env)))
        ((null? env)
         (error "unbound variable" var))))
```

Defining a variable

- Math-eval needs to handle expressions that look like (define variable expr1)
  - expr1 can be a sub-expression
- Add two functions to the evaluator:
  - definition?: tests if an expr fits the form of a definition.
  - eval-definition: extract the variable, recursively evaluate expr1, and add a binding to the current frame.

Implementing conditionals

- We will have one conditional in our mini-language: ifzero
- Syntax: (ifzero expr1 expr2 expr3)
- Semantics:
  - Evaluate expr1, test if it's equal to zero.
  - If yes, evaluate and return expr2.
  - If no, evaluate and return expr3.

Implementing conditionals

- Add functions ifzero? and eval-ifzero.