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

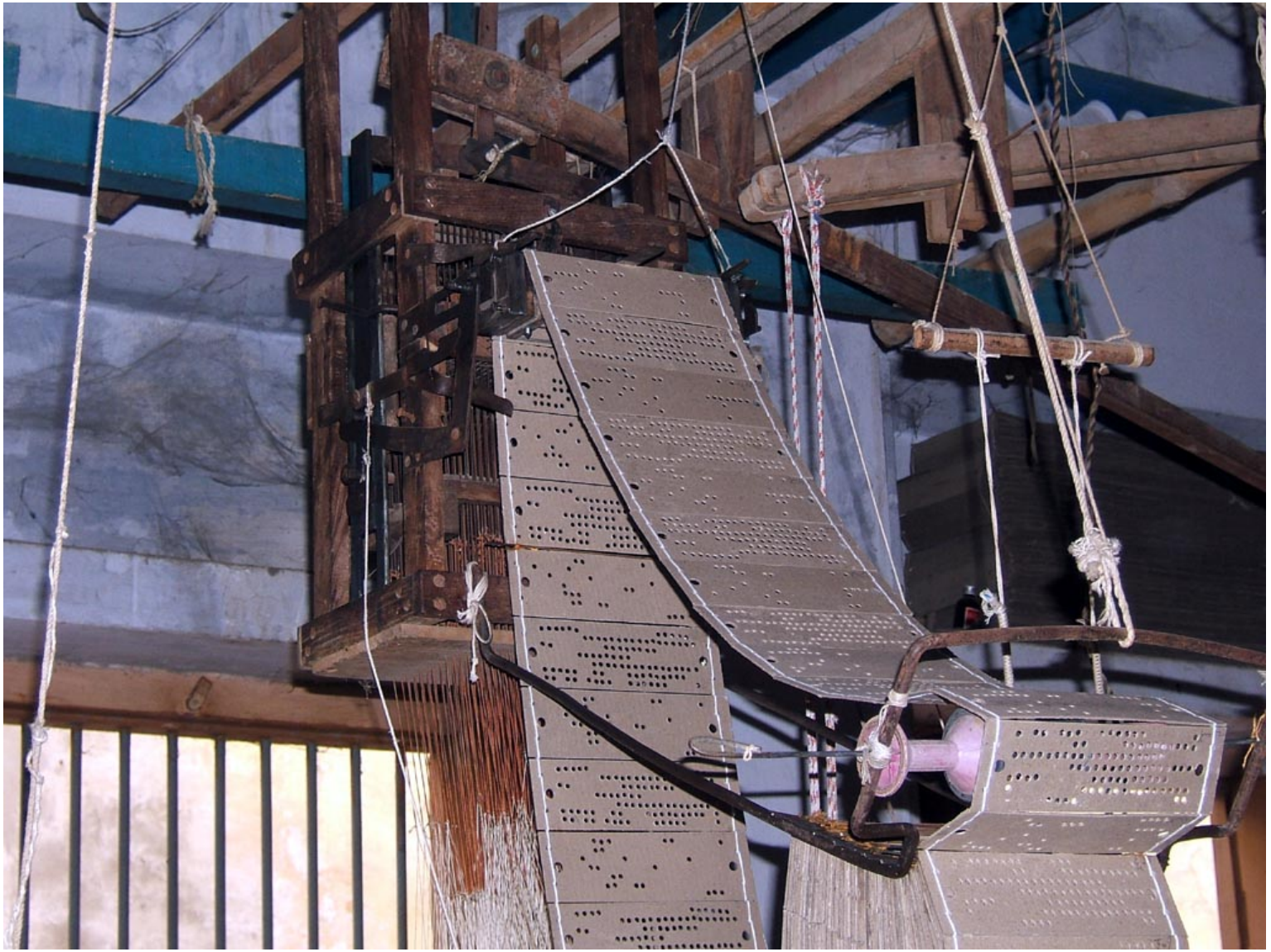
- Implementing something requires fully understanding its semantics
- Things like closures and objects are not “magic”
- Many programming tasks are like implementing PLs
  - Example: "connect-the-dots programming language" from 141

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: '

Further digression: quoting

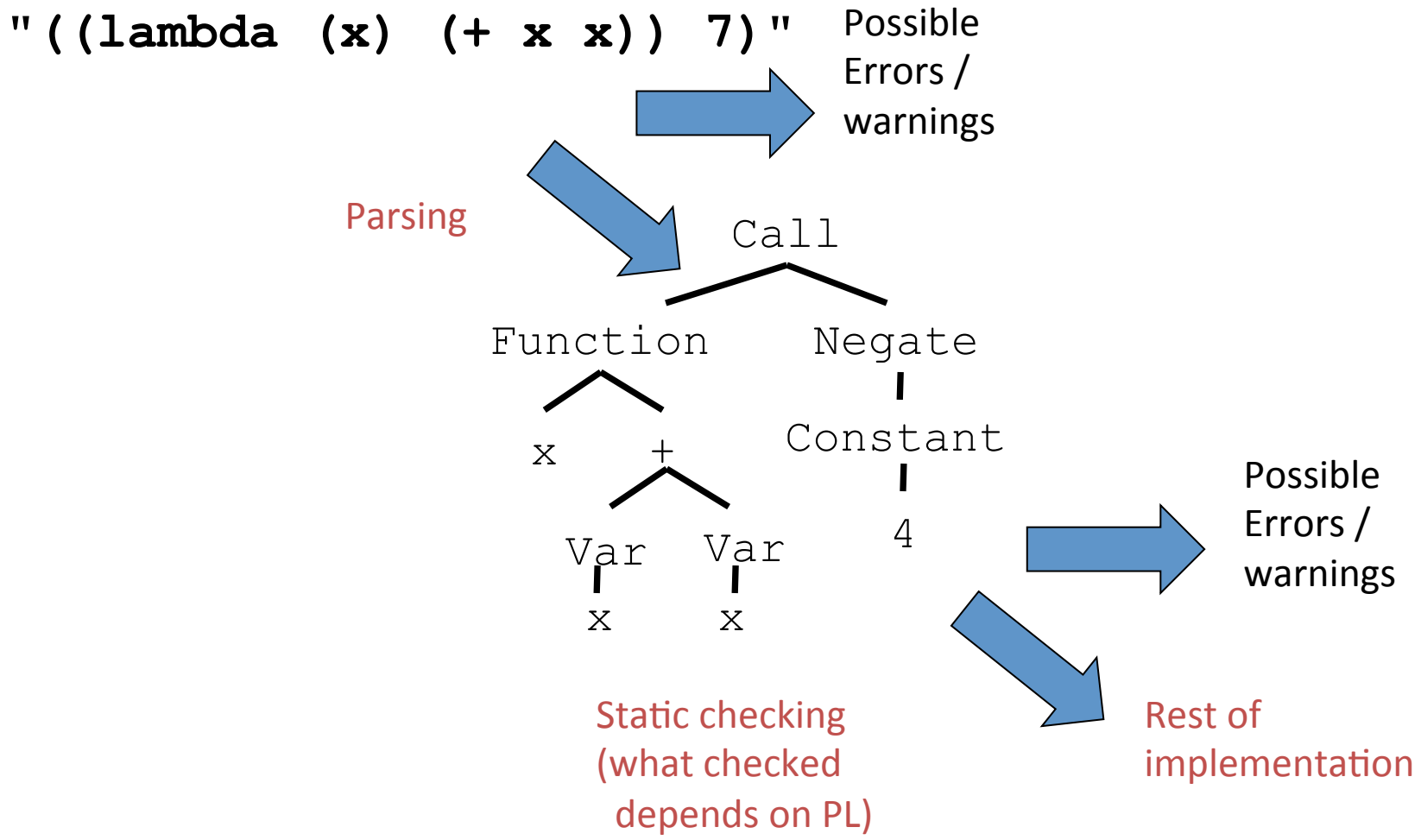
- Quoting **`(quote …)`** or **`'(…)`** is a special form that makes "everything underneath" symbols and lists, not variables and calls
  - But then calling **`eval`** on it looks up symbols as code
  - So **`quote`** and **`eval`** are *inverses*

```
(list 'begin
      (list 'print "hi")
      (list '+ 4 2))
```

=

```
(quote (begin
          (print "hi")
          (+ 4 2)))
```

Back to implementing a language



Skipping those steps

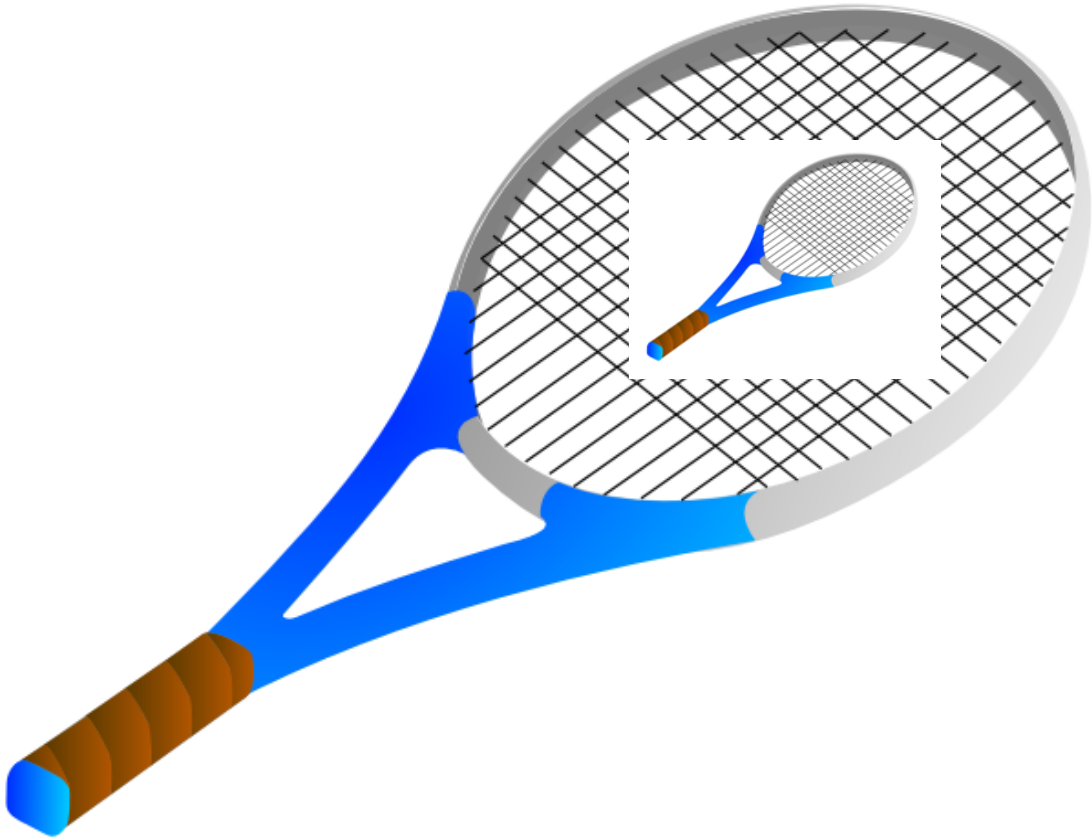
If language to be interpreted (X) is very close to the interpreter language (Y), then take advantage of this!

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

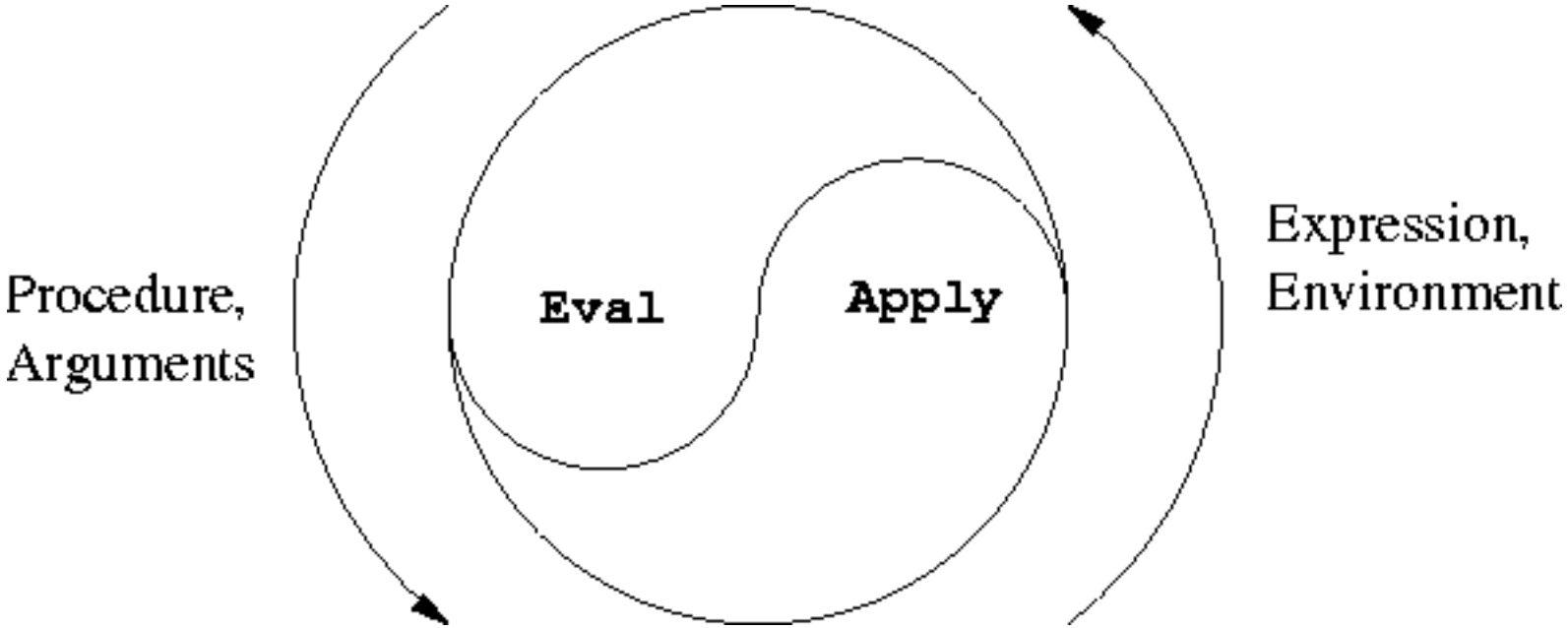
We can also, for simplicity, skip static checking

- Assume subexpressions have correct types
  - *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 mini-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 mini-eval inside eval-add.

You try

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

Adding Variables

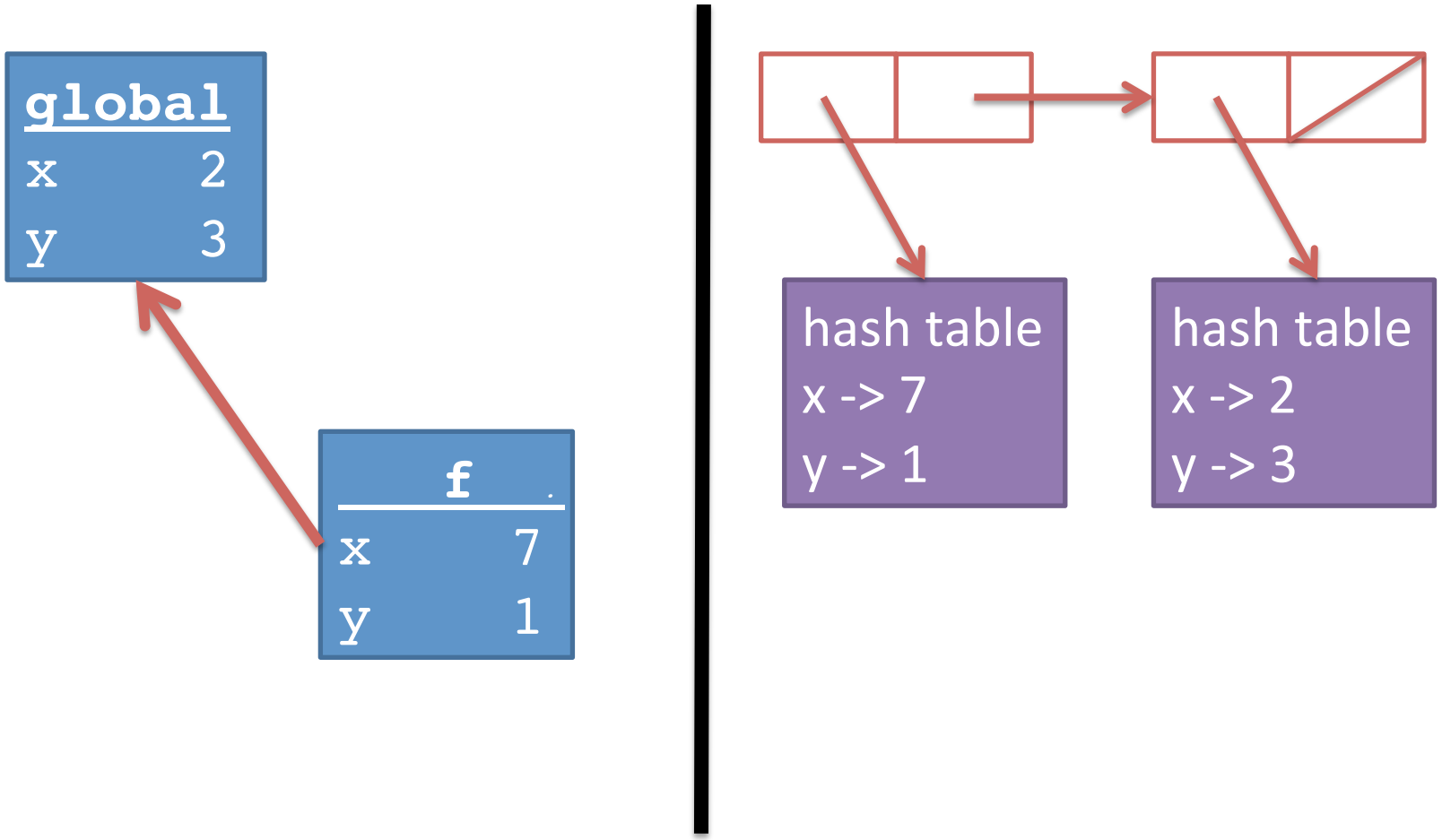
Implementing variables

- 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 mini-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

- Mini-eval needs to handle expressions that look like (define variable expr1)
  - expr1 can contain sub-expressions
- 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.

- Designing our interpreter around **`mini-eval`**.
- **`(define (mini-eval expr env) …`**
- Determines what type of expression **`expr`** is
- Dispatch the evaluation of the expression to the appropriate function
  - **`number?`** -> evaluate in place
  - **`symbol?`** -> **`lookup-variable-value`**
  - **`add?/subtract?/multiply?`** -> appropriate math func
  - **`definition?`** -> **`eval-define`**
  - **`ifzero?`** -> **`eval-ifzero`**

Today

- Two more pieces to add:
  - Closures (lambda? / eval-lambda)
  - Function calls (call? / eval-call)

Implementing closures

- In Mini-Racket, all (user-defined) functions and closures will have a single argument.
- Syntax: `(lambda var expr)`
- Semantics: Creates a new closure (anonymous function) of the single argument `var`, whose body is `expr`.

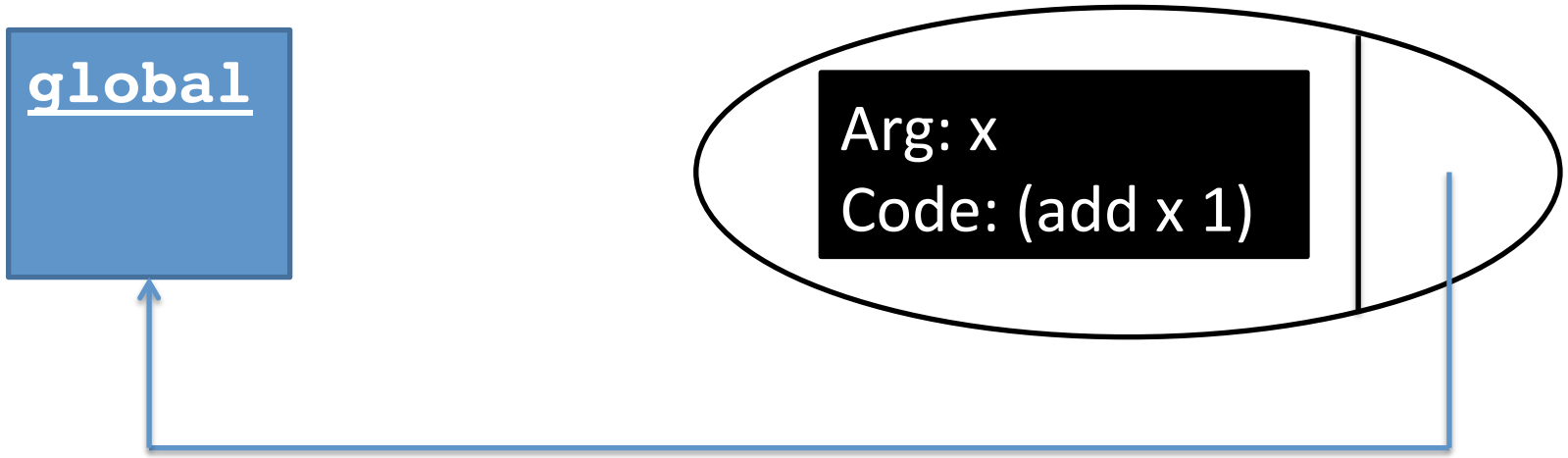
(lambda var expr)

- Need a new data structure to represent a closure.
- Why can't we just represent them as the list (lambda var closure) above?
  - Hint: What is missing? Think of environment diagrams.

(lambda var expr)

- We choose to represent closures using a list of four components:
  - The symbol `'closure`
  - The argument variable (`var`)
  - The body (`expr`)
  - The environment in which this closure was defined.

Evaluate at top level: `(lambda x (add x 1))`



Our evaluator should return

```
'(closure x (add x 1) (#hash(…)))
```

Write lambda? and eval-lambda

- lambda? is easy.
- eval-lambda should:
  - Extract the variable name and the body, but don't evaluate the body (not until we call the function)
  - Return a list of the symbol 'closure, the variable, the body, and the current environment.

```
(define (eval-lambda expr env)
  (list 'closure
        (cadr expr)        ; the variable
        (caddr expr)       ; the body
        env))
```

Function calls

- First we need the other half of the eval/apply paradigm.

- Remember from environment diagrams:

- To evaluate a function call, make a new frame with the function's arguments bound to their values, then run the body of the function using the new environment for variable lookups.

Apply

```
(define (mini-apply closure argval)
```

Pseudocode:

- Make a new frame mapping the closure's argument (i.e., the variable name) to argval.
- Make a new environment consisting of the new frame pointing to the closure's environment.
- Evaluate the closure's body in the new environment.

Apply

```
(define (mini-apply closure argval)
  (let ((new-frame (make-hash)))
    (hash-set! new-frame <arg name> argval)
    (let ((new-env
            <construct new environment>))
      <eval body of closure in new-env>
    )))
```

Apply

```
(define (mini-apply closure argval)
  (let ((new-frame (make-hash)))
    (hash-set! new-frame (cadr closure) argval)
    (let ((new-env
           (cons new-frame (cadddr closure))))
      (mini-eval (caddr closure) new-env))))
```

Function calls

- Syntax: (call expr1 expr2)
- Semantics:
  - Evaluate expr1 (must evaluate to a closure)
  - Evaluate expr2 to a value (the argument value)
  - Apply closure to value (and return result)

You try it

- Write call? (easy)
- Write eval-call (a little harder)
  - Evaluate expr1 (must evaluate to a closure)
  - Evaluate expr2 to a value (the argument value)
  - Apply closure to value (and return result)
- When done, you now have a Turing-complete language!

```
; expr looks like
; (call expr1 expr2)
(define (eval-call expr env)
  (mini-apply
    <eval the function>
    <eval the argument>)
```

```
(define (eval-call expr env)
  (mini-apply
    (mini-eval (cadr expr) env)
    (mini-eval (caddr expr) env)))
```

Magic in higher-order functions

The “magic”: How is the “right environment” around for lexical scope when functions may return other functions, store them in data structures, etc.?

Lack of magic: The interpreter uses a closure data structure to keep the environment it will need to use later

Is this expensive?

- *Time* to build a closure is tiny: make a list with four items.

- *Space* to store closures *might* be large if environment is large.

Interpreter steps

- Parser
  - Takes code and produces an intermediate representation (IR), e.g., abstract syntax tree.
- Static checking
  - Typically includes syntactical analysis and type checking.
- Interpreter directly runs code in the IR.

Compiler steps

- Parser
- Static checking
- Code optimizer
  - Take AST and alter it to make the code execute faster.
- Code generator
  - Produce code in output language (and save it, as opposed to running it).

Code optimization

```
// Test if n is prime
boolean isPrime(int n) {
 for (int x = 2; x < sqrt(n); x++) {
   if (n % x == 0) return false;
 }
 return true;
}
```

Code optimization

```
// Test if n is prime
boolean isPrime(int n) {
 double temp = sqrt(n);
 for (int x = 2; x < temp; x++) {
   if (n % x == 0) return false;
 }
 return true;
}
```

Common code optimizations

- Replacing constant expressions with their evaluations.
- Ex: Game that displays an 8 by 8 grid. Each cell will be 50 pixels by 50 pixels on the screen.
  - int CELL_WIDTH = 50;
  - int BOARD_WIDTH = 8 * CELL_WIDTH;

Common code optimizations

- Replacing constant expressions with their evaluations.
- Ex: Game that displays an 8 by 8 grid. Each cell will be 50 pixels by 50 pixels on the screen.
  - int CELL_WIDTH = 50;
  - int BOARD_WIDTH = 400;
- References to these variables would probably replaced with constants as well.

Common code optimizations

- Reordering code to improve cache performance.

```
for (int x = 0; x < HUGE_NUMBER; x++) {
  huge_array[x] = f(x)
  another_huge_array[x] = g(x)
}
```

Common code optimizations

- Reordering code to improve cache performance.

```
for (int x = 0; x < HUGE_NUMBER; x++) {
  huge_array[x] = f(x)
}
for (int x = 0; x < HUGE_NUMBER; x++) {
  another_huge_array[x] = g(x)
}
```

Common code optimizations

- Loops: unrolling, combining/distribution, change nesting
- Finding common subexpressions and replacing with a reference to a temporary variable.
  - (a + b)/4 + (a + b)/3
- Recursion: replace with iteration if possible.
  - That's what tail-recursion optimization does!

- Why don't interpreters do these optimizations?

- Usually, there's not enough time.
  - We need the code to run ***NOW***!
  - Sometimes, can optimize a little (e.g., tail-recursion).

Code generation

- Last phase of compilation.
- Choose what operations to use in the output language and what order to put them in *(instruction selection, instruction scheduling).*
- If output in a low-level language:
  - Pick what variables are stored in which registers (register allocation).
  - Include debugging code? (store "true" function/ variable names and line numbers?)

Java

- Uses both interpretation and compilation!
- Step 1: Compile Java source to bytecode.
  - Bytecode is "machine code" for a made-up computer, the Java Virtual Machine (JVM).
- Step 2: An interpreter interprets the bytecode.

- Historically, the bytecode interpreter made Java code execute very slowly (1990s).

Just-in-time compilation

- Bytecode interpreters historically would translate each bytecode command into machine code and immediately execute it.
- A just-in-time compiler has two optimizations:
  - Caches bytecode -> machine code translations so it can re-use them later.
  - Dynamically compiles sections of bytecode into machine code "when it thinks it should."

JIT: a classic trade-off

- Startup is slightly slower
  - Need time to do some initial dynamic compilation.
- Once the program starts, it runs faster than a regular interpreter.
  - Because some sections are now compiled.