CSE341: Programming Languages

Lecture 17
Structs, Implementing Languages, Implementing Higher-Order Functions

Dan Grossman

Fall 2011

Review

- Given pairs and dynamic typing, you can code up "one-of types" by using first list-element like a constructor name:

```
(define (const i)    (list 'const i))
(define (add e1 e2) (list 'add e1 e2))
(define (negate e)  (list 'negate e))
```

- But much better and more convenient is Racket's structs
  - Makes a new dynamic type (**`pair?`** answers false)
  - Provides constructor, predicate, accessors

```
(struct const (i)    #:transparent)
(struct add (e1 e2) #:transparent)
(struct negate (e)  #:transparent)
```

Defines trees

- Either lists or structs (we'll use structs) can then let us build trees to represent compound data such as expressions

```
(add (const 4)
     (negate (add (const 1)
                  (negate (const 7)))))
```

- Since Racket is dynamically typed, the idea that a set of constructors are variants for "an expression datatype" is in our heads / comments
  - Skipping: Racket's *contracts* have such notions

ML's view of Racket's "type system"

One way to describe Racket is that it has "one big datatype"

– All values have this same one type

- Constructors are applied implicitly (values are *tagged*)
  - **`42`** is implicitly "int constructor with **`42`**"

- Primitives implicitly *check tags and extract data*, raising errors for wrong constructors
  - **`+`** is implicitly "check for int constructors and extract data"
  - [Actually Racket has a *numeric tower* that **`+`** works on]

- Built-in: numbers, strings, booleans, pairs, symbols, procedures, etc.
  - Each struct creates a *new constructor*, a feature many dynamic languages do not have
  - **`(struct …)`** can be neither a function nor a macro

Implementing PLs

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

- Syntax vs. semantics vs. idioms
- Powerful constructs like pattern-matching, closures, dynamically typed pairs, macros, …

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 PL *A*

- Write an interpreter in another language *B*
  - Better names: evaluator, executor
  - Take a program in *A* and produce an answer (in *A*)

- Write a compiler in another language *B* to a third language *C*
  - Better name: translator
  - Translation must *preserve meaning* (equivalence)

We call *B* the metalanguage; crucial to keep *A* and *B* straight

Very first language needed a hardware implementation

Reality more complicated

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

- But in modern practice have both and multiple layers

A plausible example:

- Java compiler to bytecode intermediate language
- Have an interpreter for bytecode (itself in binary), but compile frequent functions to binary at run-time
- The chip is itself an interpreter for binary
  - Well, except these days the x86 has a translator in hardware to more primitive micro-operations that it then executes

Racket uses a similar mix

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 managers, friends, and other professors ☺

Okay, they do have one point

In a traditional implementation via compiler, you do not need the language implementation 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`** *in Racket*

Appropriate idioms for **`eval`** are a matter of contention

- Often but not always there is a better way
- Programs with **`eval`** are harder to analyze

We won't use **`eval`**, but no point in leaving it mysterious

- It works on nested lists of symbols and other values

```
(define (make-some-code y) ; just returns a list
  (if y
     (list 'begin (list 'print "hi") (list '+ 4 2))
     (list '+ 5 3)))

(eval (make-some-code #t)) ; prints "hi", result 6
```

Further digression: quoting

- Quoting **`(quote …)`** or **`'(…)`** is a special form that makes "everything underneath" atoms 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))
```

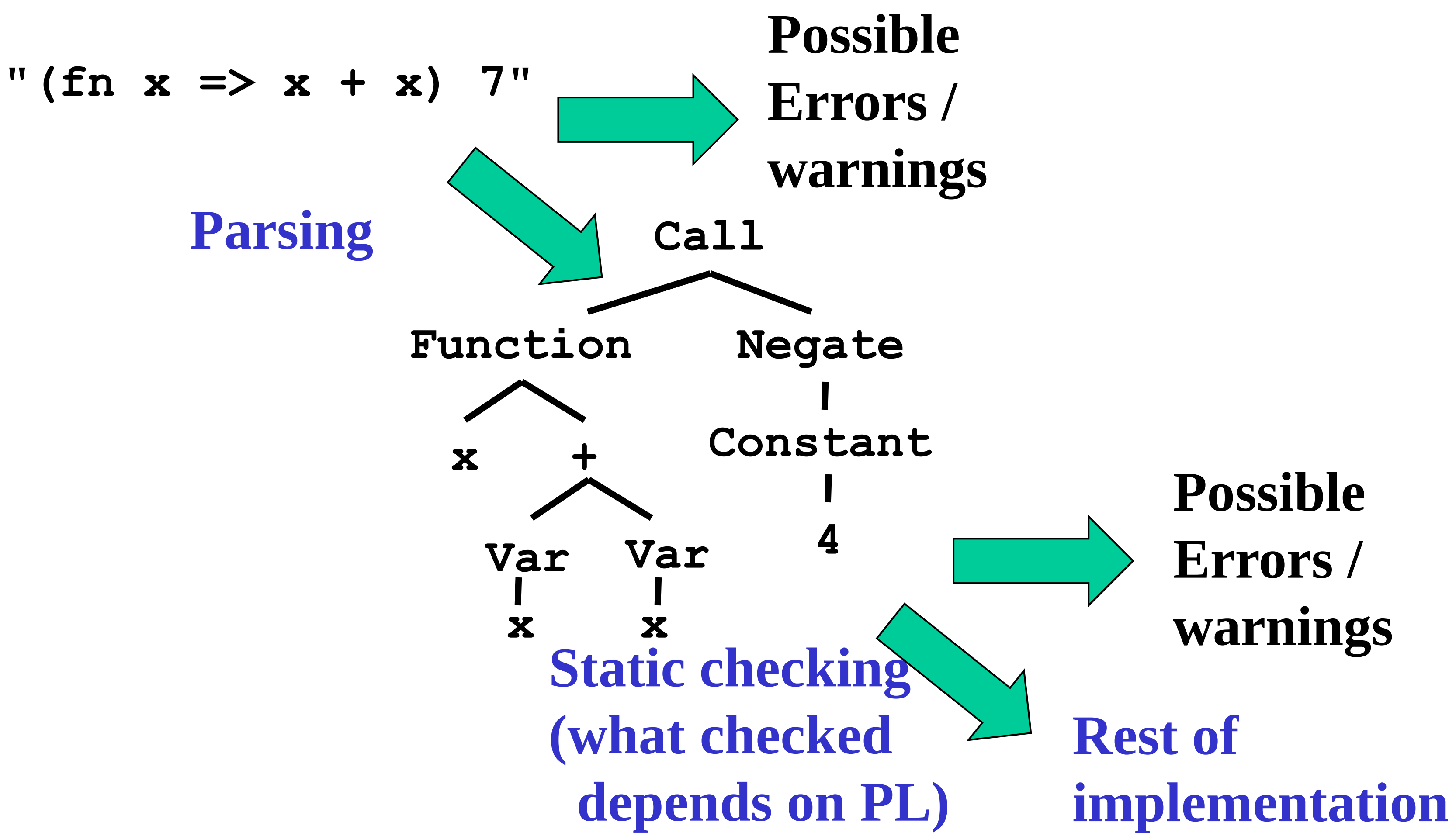
=

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

- There is also *quasiquoting*
  - Everything underneath is atoms and lists except if *unquoted*
  - Languages like Ruby, Python, Perl eval strings and support putting expressions inside strings, which is quasiquoting

- We won't use any of this: see The Racket Guide if curious

Back to implementing a language



Skipping those steps

Alternately, we can *embed* our language inside (data structures) in the metalanguage

- Skip parsing: Use constructors instead of just strings
- 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
  - *Do* worry about **`(add (fun …) (int 14))`**

The arith-exp example

This embedding approach is exactly what we did for the PL of arithmetic expressions:

```
(struct const (i)    #:transparent)
(struct add (e1 e2) #:transparent)
(struct negate (e)  #:transparent)
```

```
(add (const 4)
     (negate (add (const 1)
                  (negate (const 7)))))
```

```
(define (eval-exp e) … )
```

Note: So simple there are no dynamic type errors in the interpreter

The interpreter

An interpreter takes programs in the language and produces values (answers) in the language

- Typically via recursive helper functions with cases
- This example is so simple we don't need a helper and can assume all recursive results are constants

```
(define (eval-exp e)
   (cond
     [(const? e) e]
     [(add? e)
      (const (+ (const-i (eval-exp (add-e1 e)))
                (const-i (eval-exp (add-e2 e)))))]
     [(negate? e)
      (const (- (const-i (eval-exp (negate-e e)))))]
     [#t (error "eval-exp expected an expression")]))
```

"Macros"

Another advantage of the embedding approach is we can use the metalanguage to define helper functions that create programs in our language

- They generate the (abstract) syntax
- Result can *then* be put in a larger program or evaluated
- This is a lot like a macro, using the metalanguage as our macro system

Example:

All this does is create a program that has four constant expressions:

```
(define (triple x) (add x (add x x)))

(define p (add (const 1) (triple (const 2))))
```

What's missing

Two very interesting features missing from our arithmetic-expression language:

- Local variables
- Higher-order functions with lexical scope

How to support local variables:

- Interpreter helper function(s) need to take an *environment*
- As we have said since lecture 1, the environment maps variable names to values
  - A Racket association list works well enough
- Evaluate a variable expression by looking up the name
- A let-body is evaluated in a larger environment

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 (with two parts) to keep the environment it will need to use later

Evaluate a function expression:

- A function is not a value; a closure is a value
- Create a closure out of (a) the function and (b) the current environment

Evaluate a function call:

- …

Function calls

- Evaluate 1st subexpression to a closure with current environment
- Evaluate 2nd subexpression to a value with current environment
- Evaluate closure's function's body in the closure's environment, extended to map the function's argument-name to the argument-value
  - And for recursion, function's name to the whole closure

This is the same semantics we learned a few weeks ago "coded up"

Given a closure, the code part is only ever evaluated using the environment part (extended), not the environment at the call-site

Is that expensive?

- *Time* to build a closure is tiny: a struct with two fields

- *Space* to store closures *might* be large if environment is large
  - But environments are immutable, so natural and correct to have lots of sharing, e.g., of list tails (cf. lecture 3)

- Alternative: Homework 5 challenge problem is to, when creating a closure, store a possibly-smaller environment holding only the variables that are free variables in the function body
  - Free variables: Variables that occur, not counting shadowed uses of the same variable name
  - A function body would never need anything else from the environment

Free variables examples

```
(lambda () (+ x y z))

(lambda (x) (+ x y z))

(lambda (x) (if x y z))

(lambda (x) (let ([y 0]) (+ x y z)))

(lambda (x y z) (+ x y z))

(lambda (x) (+ y (let ([y z]) (+ y y))))
```

Free variables examples

```
(lambda () (+ x y z))    ; x y z

(lambda (x) (+ x y z))  ; y z

(lambda (x) (if x y z)) ; y z

(lambda (x) (let ([y 0]) (+ x y z))) ; z

(lambda (x y z) (+ x y z)) ; {}

(lambda (x) (+ y (let ([y z]) (+ y y)))) ; y z
```

Compiling higher-order functions

- Key to the interpreter approach: Interpreter helper function takes an environment argument
  - Recursive calls can use a different environment
- Can also compile higher-order functions by having the translation produce “regular” functions (like in C or assembly) that *all* take an extra *explicit* argument called “environment”
- And compiler replaces all uses of free variables with code that looks up the variable using the environment argument
  - Can make these fast operations with some tricks
- Running program still creates closures and every function call passes the closure’s environment to the closure’s code