# CSE 413
# Languages & Implementation

Hal Perkins

Winter 2019

Structs, Implementing Languages
(credits: Dan Grossman, CSE 341)

# Goals

- Representing programs as data
- Racket structs as a better way to represent abstract programs (& other data)
- Writing interpreters to execute programs represented as data
- Using closures to implement higher-order functions

- This stuff is crucial for the next assignment
  – Without it you will be totally lost
  – With it, it's challenging but straightforward

# Data structures in Racket

We've been using functions to abstract from lists

```
(make-expr left op right) =>
                (list left op right)
(operator expr) => (cadr expr)
```
etc.

We could also build "weakly typed" or self-describing data by tagging each list:

```
(define (Const i)   (list 'Const i))
(define (Add e1 e2) (list 'Add e1 e2))
(define (Negate e)  (list 'Negate e))
```

See sample code for more examples + evaluator

# Comments on what we did

Using lists where the car of the list encodes "what kind of expression"

Key points

- Defined our own constructor, test, extract-data functions
  - Much better style than car, cadr, etc.
- Elegant recursive evaluator with big cond
- With no type system, no notion of "what is an expression" other than documentation
  - But if we use the helper functions correctly, we're ok
  - Could add more explicit error checking if desired

# Racket structs

New Racket feature

```
(struct foo (bar baz bam)    #:transparent)
```

Defines a new kind of thing (type) and introduces several associate functions:

- Constructor: `(foo e1 e2 e3)`
  - Returns a new "`foo`" with bar, baz, and bam fields initialized to e1, e2, e3
- Predicate: `(foo? e)`
  - Evaluate e and return #t if it is something made with the `foo` (constructor) function
- Extractors: `(foo-bar e)`
  - Evaluate e.  If result was made with the `foo` constructor, return the `bar` field, else an error
  - Similar extractors for other fields (foo-baz, foo-bam)

# An idiom

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

For "types" like expressions, create one struct for each "kind of" expression

- Conveniently definies constructors, tester, and extractor functions, e.g., **add add? add-e1**
- There's nothing that ties **add**, **negate**, **const** together as the "expression type" other than the convention we have in our heads and in comments
- Also nothing that restricts "types" of fields – also just convention in code and in comments

# Representing program as trees

Can use either lists or structs (we'll use structs) to build trees to represent compound data & programs

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

See code for more extensive set of struct defintions and associated evaluator for a small integer language

# Attributes

- #:transparent
  - Optional; makes struct fields visible & for us prints them in the interactions window

Also available (and optional):

- #:mutable
  - Can decide if each struct type supports mutation
    - (we will avoid this; guarantees no mutation)
  - mcons is just a predefined mutable struct

# Contrasting approaches

```
(struct add (e1 e2)    #:transparent)
```

vs

```
(define (add e1 e2) (list 'add e1 e2)
(define (add? e) (eq? (car e) 'add))
(define (add-e1) (car (cdr e)))
(define (add-e2) (car (cdr (cdr e))))
```

- This is *not* just "syntactic sugar"
  - i.e., not just convenient syntax for writing something already in the language

# The key difference

```
(struct add (e1 e2) #:transparent)
```

- The result of evaluating **(add x y)** is *not* a list
  - And there is no list for which **add?** returns #t

- **struct** makes a new kind of thing – a new type

- So using an "add" as an argument to car, cdr, etc. is a runtime error – not true for the version with lists

# Now

Step back to look at approaches to implementing programming languages…

# Implementing languages

Much of the course so far has been about fundamental concepts for *using* PLs

Syntax, semantics, idioms

Important concepts like closures, delayed evaluation, …
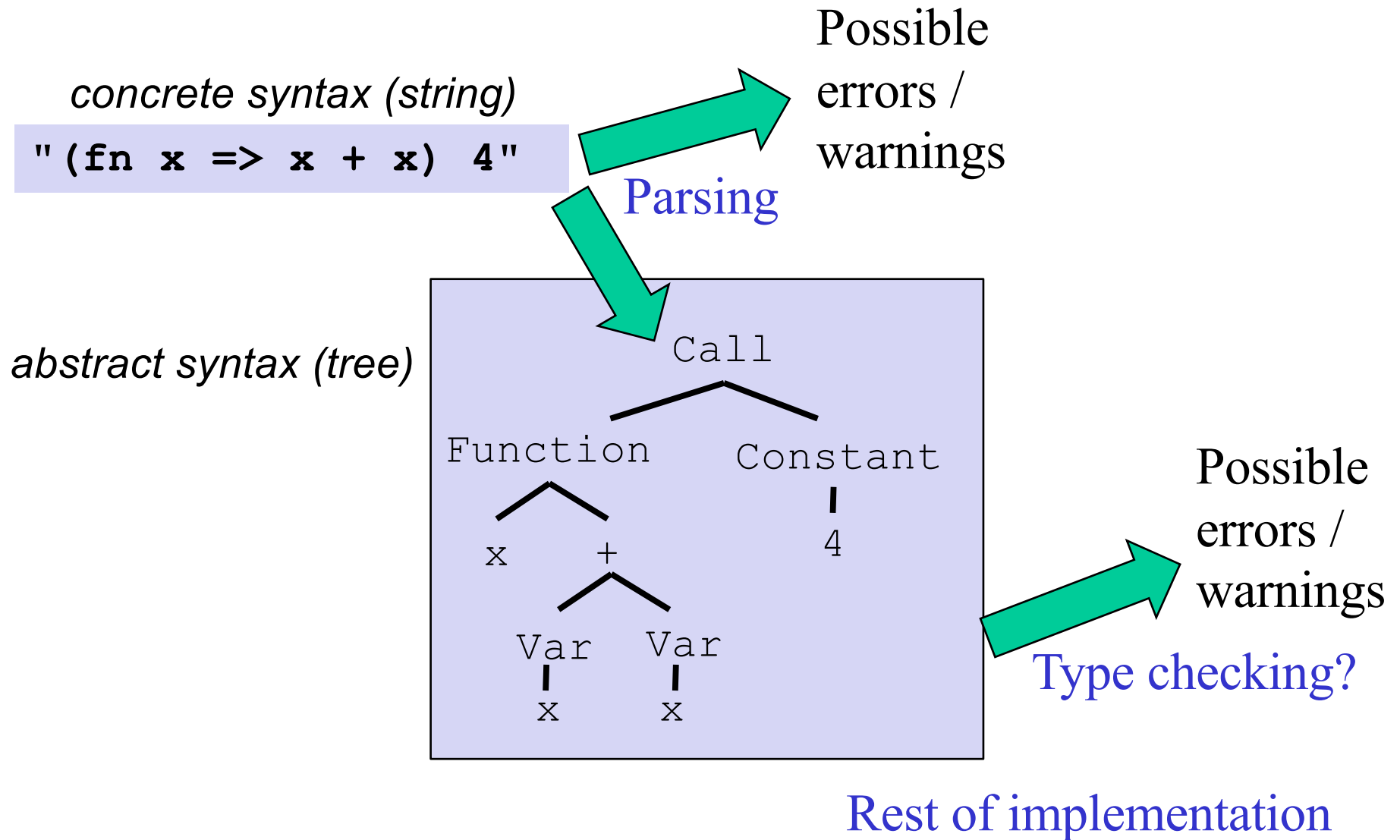

But we also want to learn basics of *implementing* PLs

Requires fully understanding semantics

Things like closures and objects are not "magic"

Many programming techniques are related/similar

Ex: rendering a document ("program" is the structured document, "pixels" is the output)

# Typical workflow

*concrete syntax (string)*

`"(fn x => x + x) 4"`

Parsing

Possible errors / warnings

*abstract syntax (tree)*

```
              Call
           /        \
    Function       Constant
      /    \           |
     x      +          4
           / \
        Var   Var
         |     |
         x     x
```

Possible errors / warnings

Type checking?

Rest of implementation

# Interpreters or Compilers

The "rest of implementation" takes the abstract syntax tree (AST) as data and "runs the program" to produce a result

Two fundamental ways to implement a prog. lang. A
- Write an *interpreter* in another language B
  - (Better names: evaluator, executor)
  - Read program in A and produce an answer (in A)
- Write a *compiler* in another language B
  - (Better name: translator)
  - Read program in A, produce an equivalent program in another language C
  - Translation must *preserve meaning* (equivalence)
- We call B the "metalanguage"
  - Crucial to keep A and B straight

# It's really more complicated

Evaluation (interpreter) and translation (compiler) are the options, but many languages are implemented with both and have multiple layers

Example: Java

- Compiler to bytecode intermediate language (.class)
- Can interpret the bytecode directly, but also…
- Can compile frequently executed code to binary
- The processor chip is an interpreter for binary
    - Except these days the chip translates x86 binary to a more primitive code that it executes

Racket uses a similar mix

# Sermon (er, rant)

Interpreter vs compiler vs combinations is about language implementation, not language definition

There is no such thing as a "compiled language" or "interpreted language"

Program cannot see how the implementation works

Unfortunately you hear things like this all the time:

"C is faster because it's compiled and LISP is interpreted"

Nonsense: You can write a C interpreter or a LISP compiler

Please politely correct your managers, friends, and other professors. ☺

# OK, they do have a point

A traditional compiler does not need the language implementation to run the program

> Can "ship the binary" without the compiler

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

> At runtime can create data and treat it as a program
>
> Then run that program
>
> So we need an implementation (compiler, interpreter, combination) at runtime

It is also true that some languages are designed with a particular implementation strategy in mind, but it doesn't mean they couldn't be implemented differently.

# Embedding one language in another

How is `(negate (add (const 2) (const 2)))` a "program" compared to "-(2+2)" ?

A traditional implementation includes a *parser* to read the string "-(2+2)" and turn it into a tree-like data structure called an *abstract syntax tree (AST)*.

    Ideal representation for either interpreting or as an intermediate stage in compiling

    Our `(negate ...)` data structure is an AST

    For now we'll create trees directly and interpret them

        Will cover parsing later in the course

    We'll also assume perfect programmers and not worry about syntax errors – parser would normally guarantee legal AST

        Interpreter still needs to worry about semantic (type) errors

# The arith-exp example

This embedding approach is exactly what we did to represent the language of arithmetic expressions using Racket structs

```
(struct const (i)    #:transparent)
(struct add (e1 e2) #:transparent)
(struct negate (e)  #:transparent)
(add (const 4)
      (negate (add (const 1)
                    (negate (const 7)))))
```

The missing piece is to define the interpreter

```
(define (eval-exp e) … )
```

# 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 helpers 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"))))
```

See example code – two versions, one with just ints, one with a second boolean type

# "Macros"

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

> They generate the (abstract) syntax

> Result can *then* be put in a larger program or evaluated

Example – Racket functions that produce abstract code:

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

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

> (all this does in create a program with 4 constant expressions)

# What's missing

- Two major things missing from this language
  - Variables: let bindings, function arguments
  - Higher-order functions with lexical scope (closures)

- To support local variables:
  - Interpreter function(s) need an *environment* as an additional argument
    - Environment maps names to values
    - A Racket association list works fine for us
  - Evaluate a variable expression by looking up the name
  - A let-body is evaluated in an augmented environment with the local binding(s)

# Higher-order functions

The "magic": How is the "right environment" found 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 when the closure function is executed

To evaluate a function expression:

>    A function is not a value, a closure is a value

>       Evaluation of a function returns a closure

>    Create a closure out of (i) the function and (ii) the current environment

To evaluate a function call…

# Function calls

To evaluate (`call` `exp1` `exp2`)

Evaluate `exp1` in the current environment to get a closure

Evaluate `exp2` in the current environment to get a value

Evaluate the closure's function's body *in the closure's environment* extended to map the function's argument name to the argument value

We only will implement single-argument functions

For recursion, a function name will evaluate to its entire closure

This is the same semantics we've been learning

Given a closure, the code part is always evaluated using the closure's environment part (extended with the argument binding), *not* the current environment at the call site.

# Sounds expensive!

It isn't!!

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

*Space* to store closures *might* be large if the environment is large

But environments are immutable, so lots of sharing is natural and correct

Possible HW challenge problem (extra credit): when creating a closure store a possibly smaller environment holding only function *free variables*, i.e., "global" variables used in a function but not bound in it

Function body would never need anything else from the environment

# What's next?

- Specific details of MUPL (interpreter assignment)
  - Demo/questions now

- Then we're mostly done with functional programming…

  …but need to take out the garbage later

  (and fit in a midterm exam when MUPL is done)

- After that: Ruby and object-oriented programming, grammars, scanners, parsers, more implementation