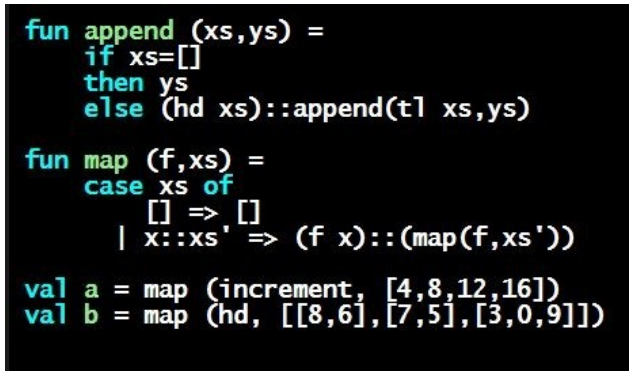


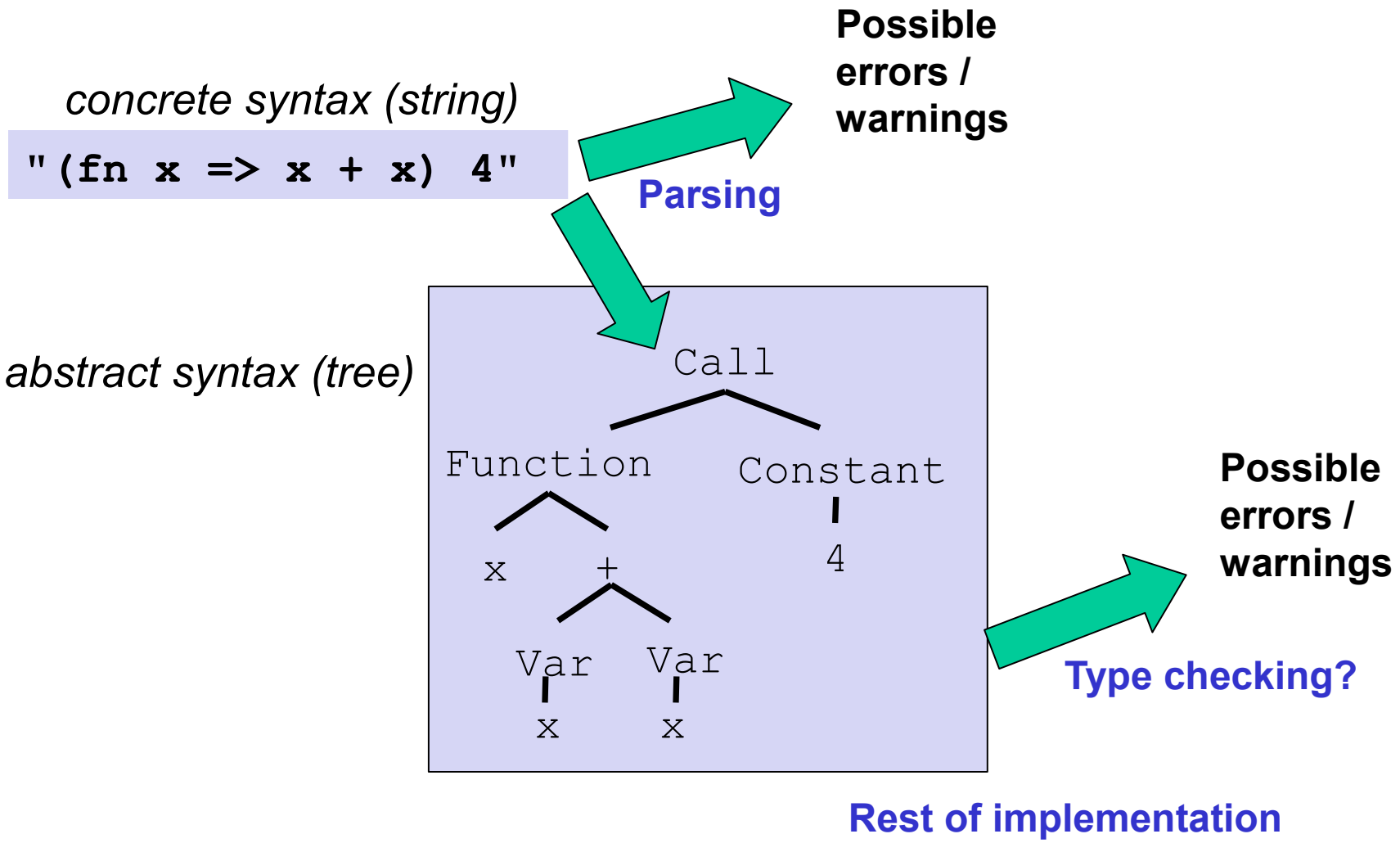
Programming Languages

Dan Grossman

2013

Implementing Programming Languages

Typical workflow



Interpreter or compiler

So “rest of implementation” takes the abstract syntax tree (AST) and “runs the program” to produce a result

Fundamentally, two approaches to implement a PL *B*:

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

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

We call *A* the metalanguage
  - Crucial to keep *A* and *B* straight

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

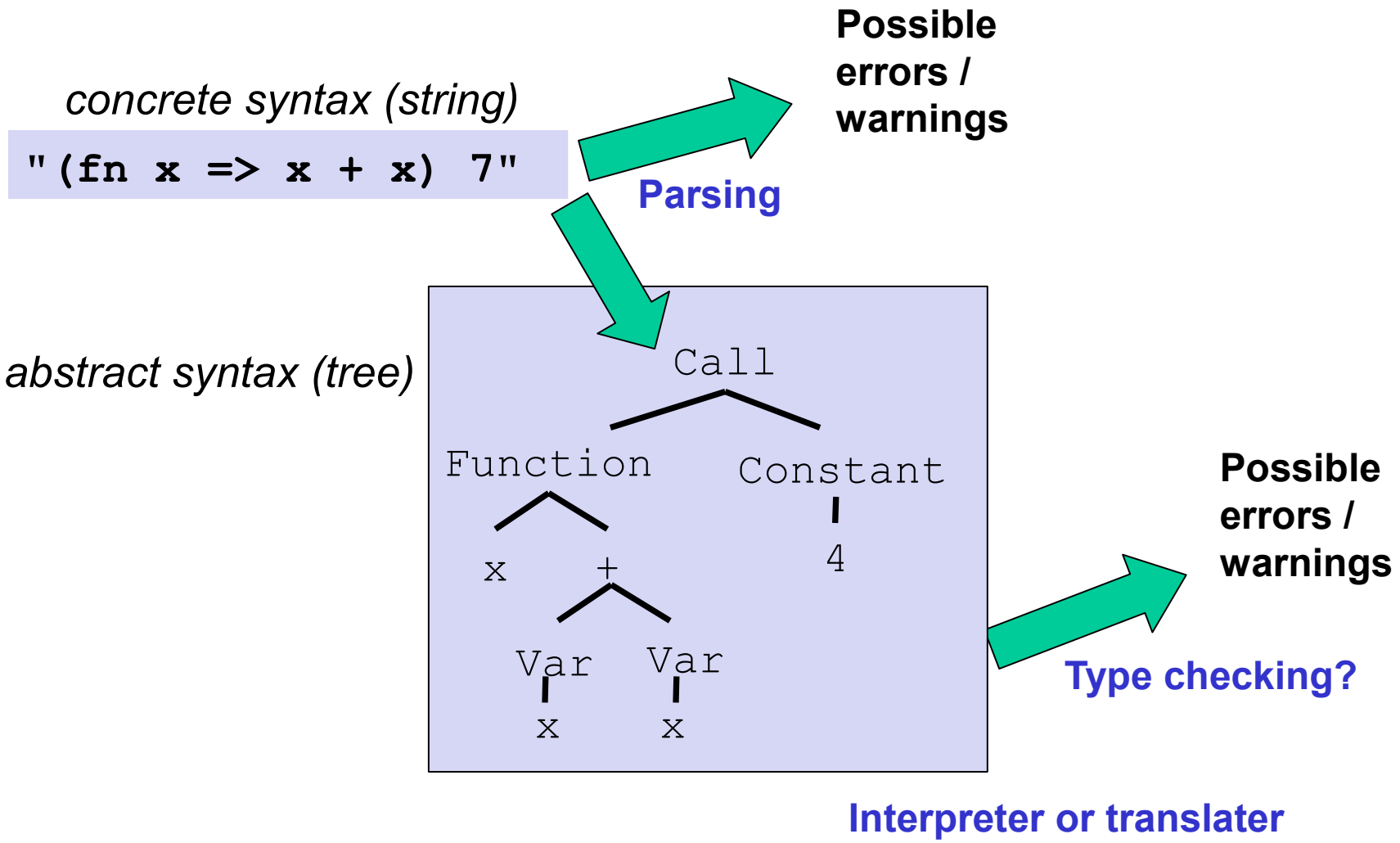
- Programs cannot "see" how the implementation works

Unfortunately, you often hear such phrases

- "C is faster because it's compiled and LISP is interpreted"
- This is nonsense; politely correct people

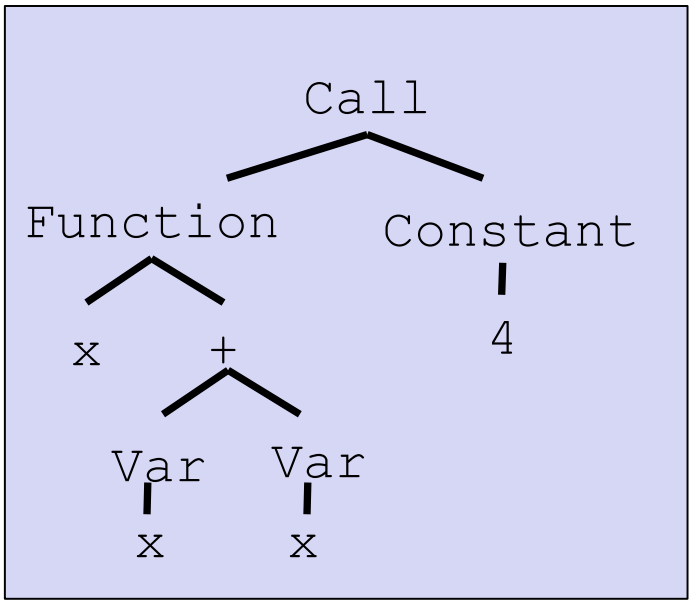
- (Admittedly, languages with "eval" must "ship with some implementation of the language" in each program)

Typical workflow



Skipping parsing

- If implementing PL *B* in PL *A*, we can skip parsing
  - Have *B* programmers write ASTs directly in PL *A*
  - Not so bad with ML constructors or Racket structs
  - Embeds *B* programs as trees in *A*




```
; define B's abstract syntax
(struct call …))
(struct function …)
(struct var …)
…
```

```
; example B program
(call (function (list "x")
                (add (var "x")
                     (var "x")))
      (const 4))
```

Already did an example!

- Let the metalanguage *A* = Racket
- Let the language-implemented *B* = "*Arithmetic Language*"
- Arithmetic programs written with calls to Racket constructors
- The interpreter is **`eval-exp`**

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

```
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e)
         (const (- (const-int
                    (eval-exp (negate-e e)))))]
        [(add? e) …]
        [(multiply? e) …]…
```

Racket data structure is *Arithmetic Language* program, which **`eval-exp`** runs