Parsing: LL vs LR
Parsing assignment
Scoping

Parsing?
Generally, two classes of parsing algorithms

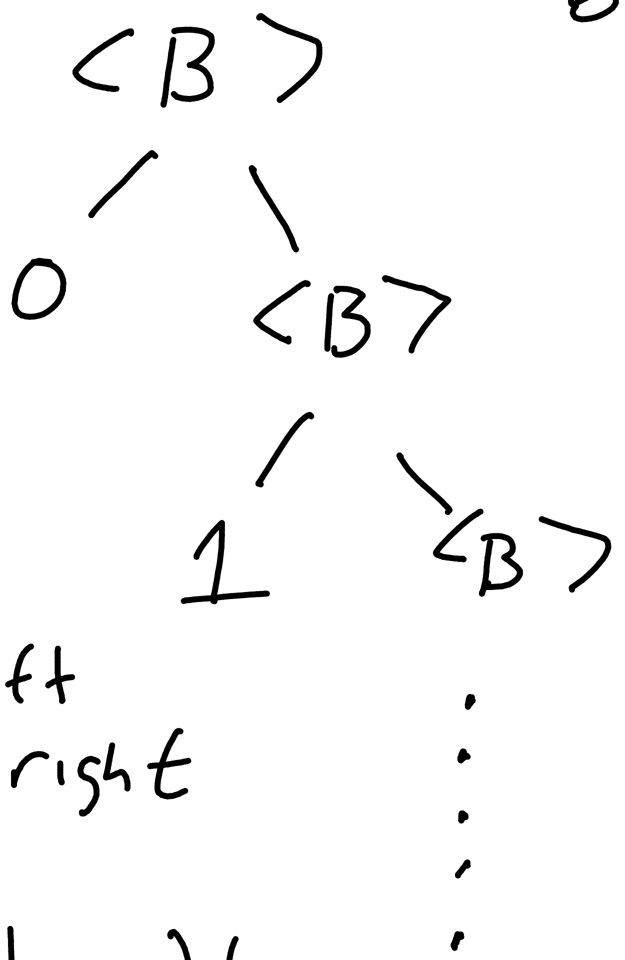
① (we're not doing for assignment)

$$\begin{tikzcd}\langle B\rangle \arrow[rr, bend left] \arrow[rrrr, bend left] & ::= & 0\langle B\rangle & \mid & 1\langle B\rangle & \mid & \varepsilon\end{tikzcd}$$

↖ start

nothing

Parse $\underline{0}\underline{1}011$

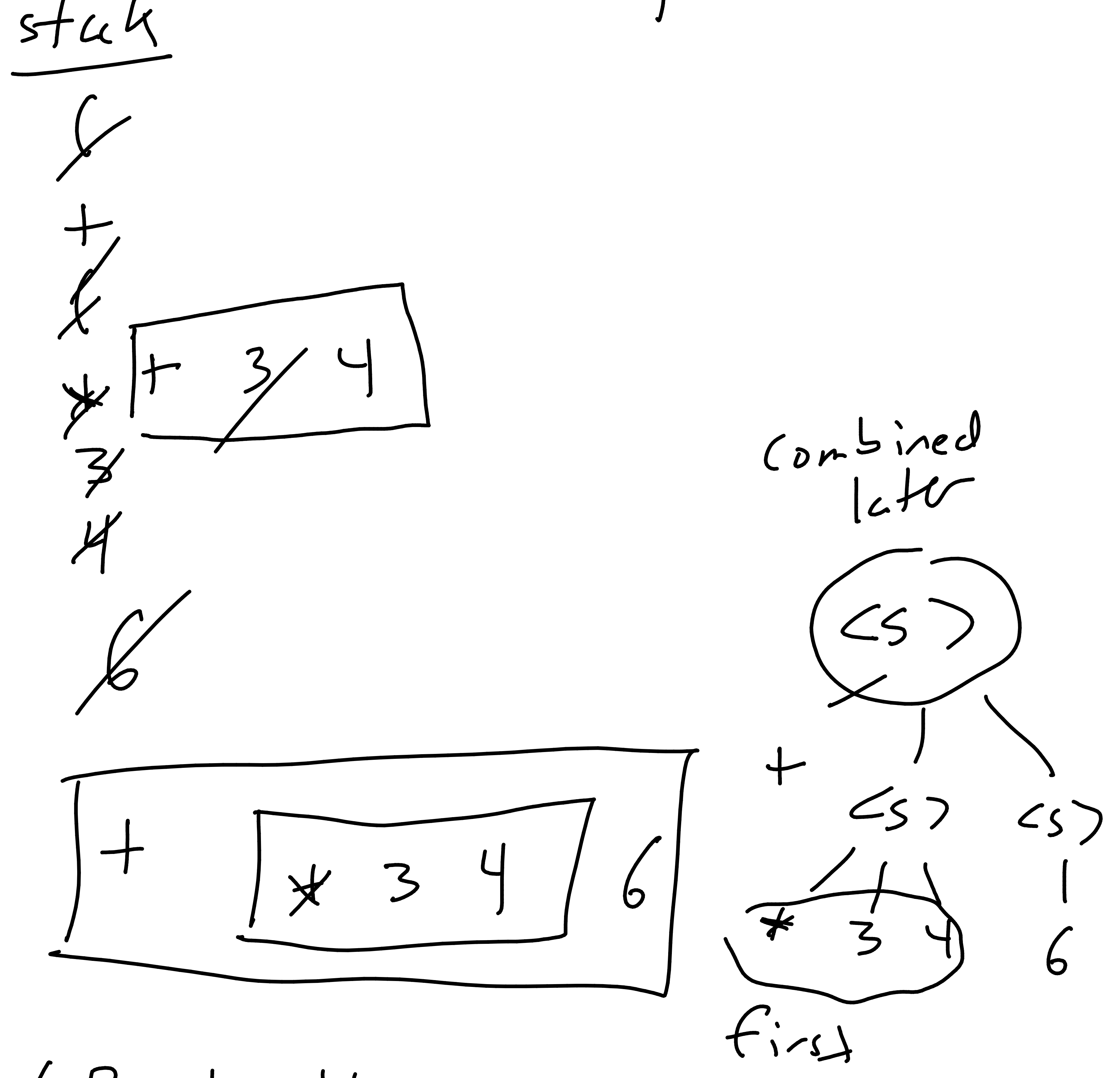


building this top-down, choose which rule based on which token is up next

traverse grammar left to right
↑
LL algorithm
↓
left to right (read program)

② Approach we're using on our assignment

(+ (* 3 4) 6)



LR algorithm
↳ traversing grammar from right to left

$$\langle s \rangle ::= \text{atom} \mid \underbrace{(\langle s \rangle^{*})}$$

0 or more of these

$$\begin{tikzcd}\langle s \rangle \\ (+\ \ 3\ \ 4) \arrow[u]\end{tikzcd}$$

Pros and cons of each

LR is generally applicable for more languages than LL, but not universally

---

LL algorithms can (not always) be faster

Parsing assignment: parse your tokens!

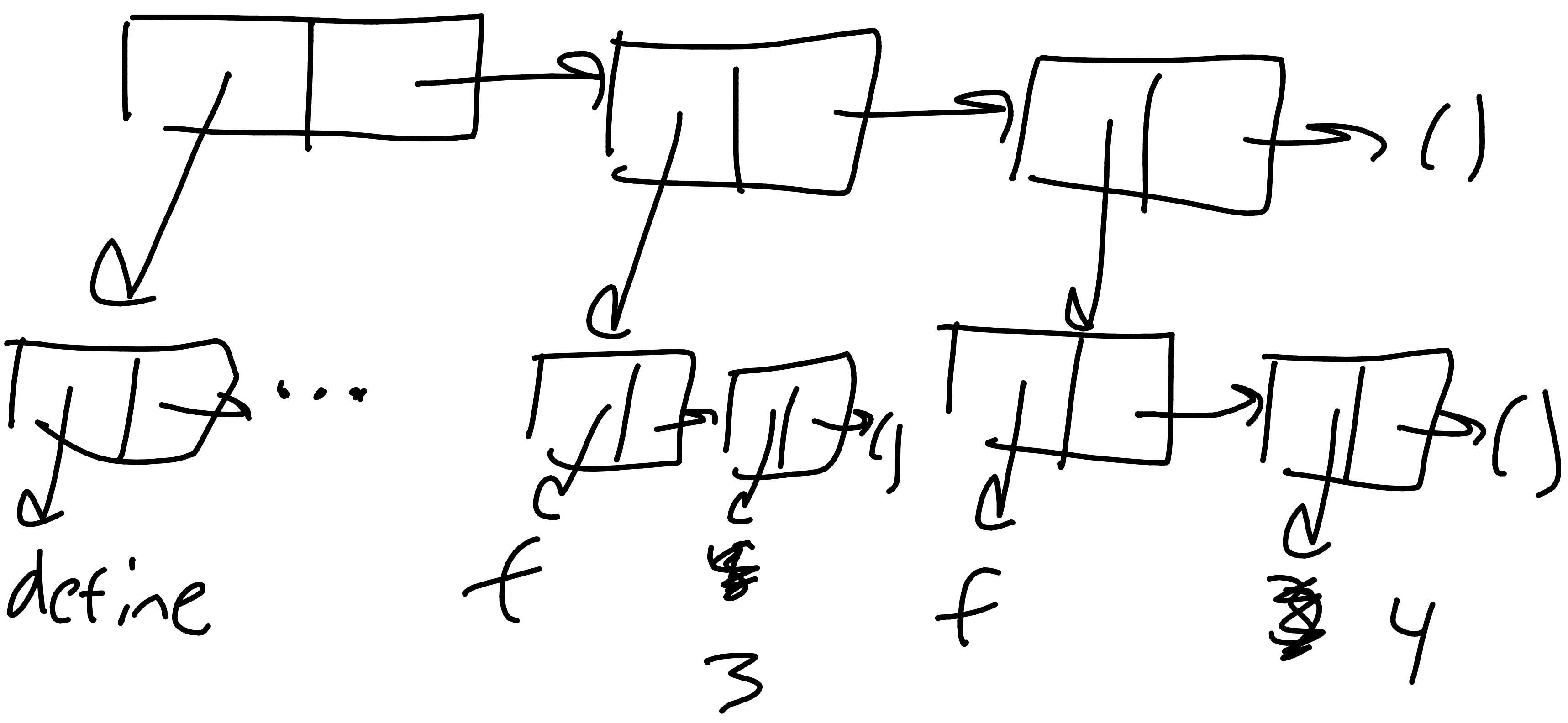
Uses the Scheme parsing algorithm we've been looking it

Scheme is inconsistent at the very top level of your program

```
(
 (define f (lambda (x) (+ x 1)))
 (f 3)
 (f 4)
) ← these don't exist
```

A Scheme program is a sequence of parsable Scheme expessions

- you need to have a top level list of these



Single quote → '(a b)

Also breaks the rules.

Really just an abbreviation for

(quote (a b))

transform

"syntactic sugar"

Scoping

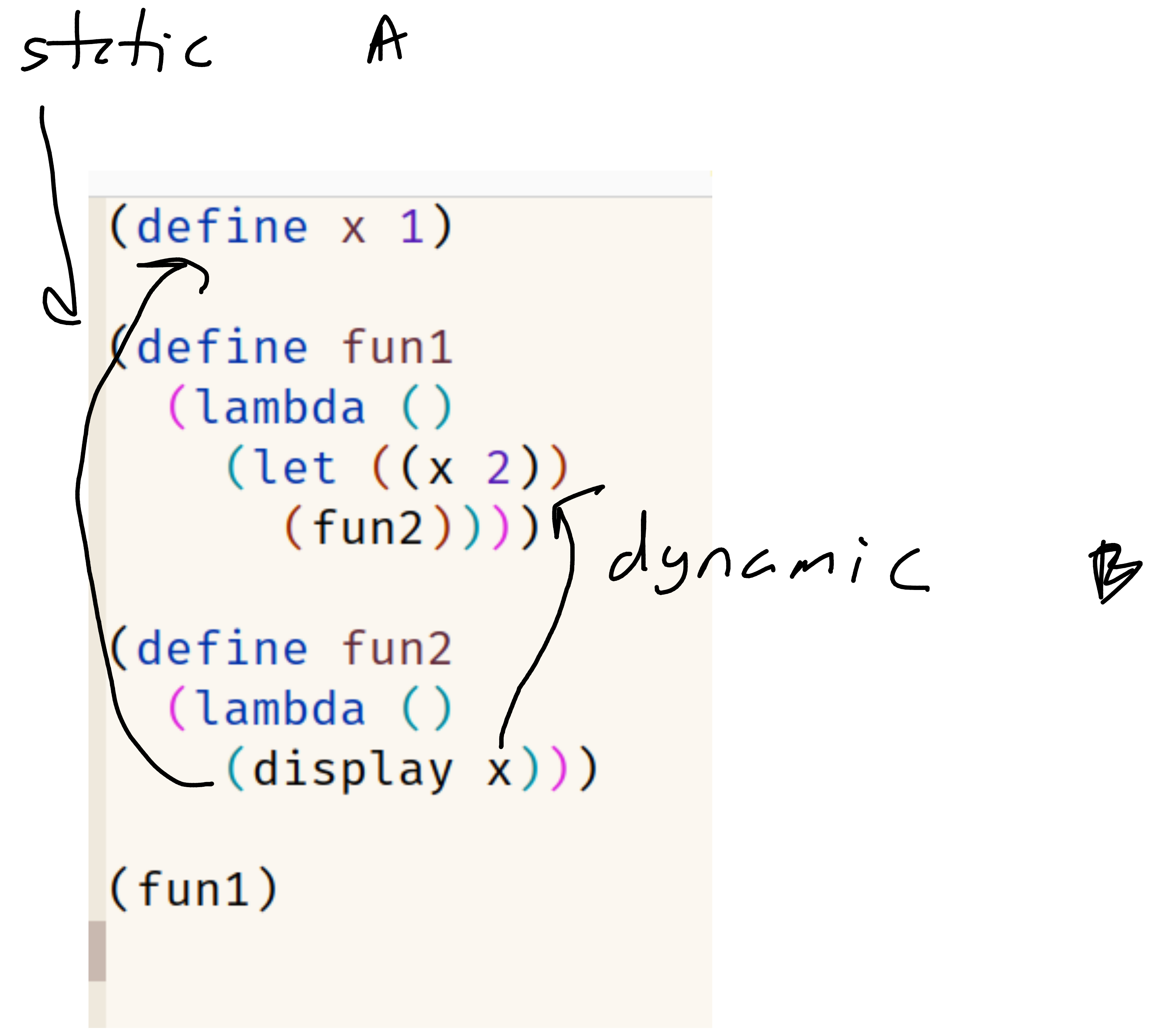
When resolving a variable, how do you find it?

1

static scoping = lexical scoping
- find a variable based on structure of code
  - typically, by looking outward in surrounding blocks

dynamic scoping -
- find a variable based on what was last seen based on program execution

2



```scheme
(define x 1)

(define fun1
  (lambda ()
    (let ((x 2))
      (fun2))))

(define fun2
  (lambda ()
    (display x)))

(fun1)
```

Python, Java, Kotlin, Scheme all use static scoping

Bash uses dynamic scoping

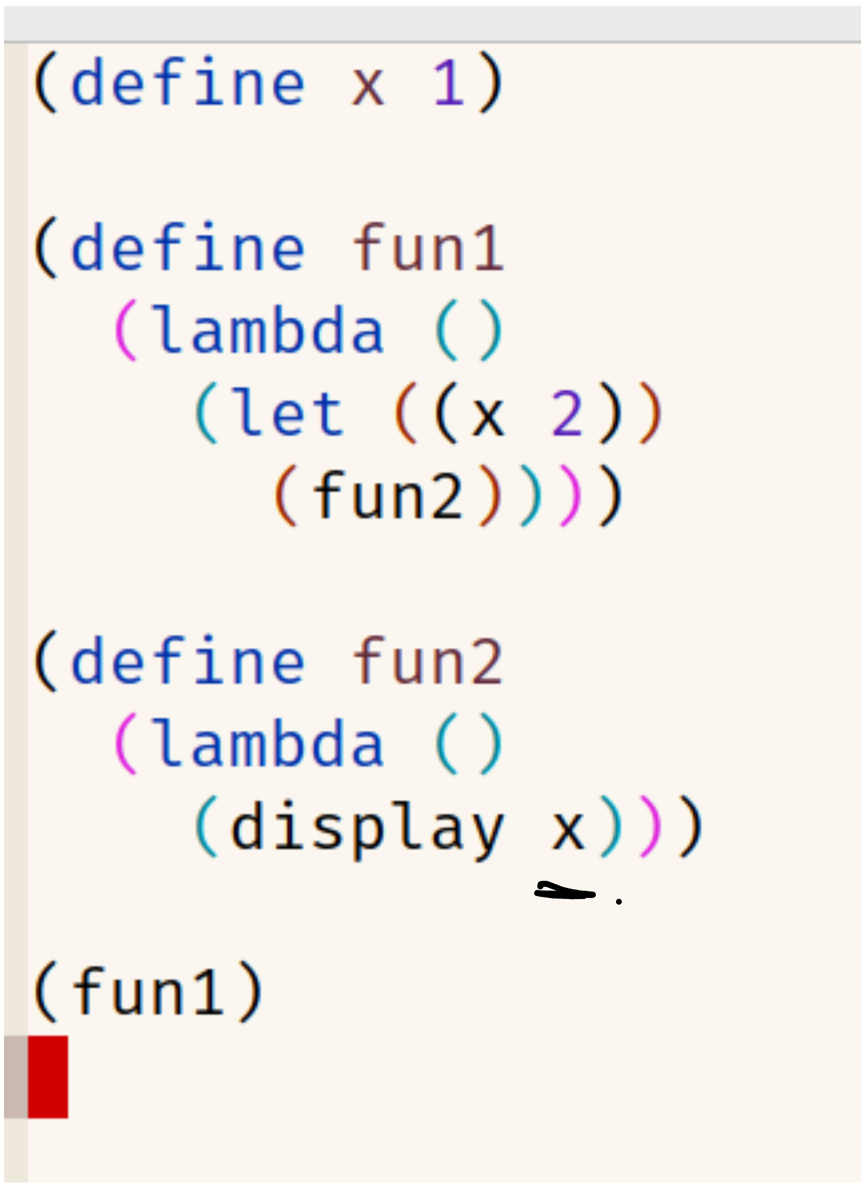
Why the difference, and what are pros/cons?

Many people find dynamic scoping to be more intuitive.

Most early languages did dynamic scoping. Early developers thought it made sense.

History begged to differ.

Why?

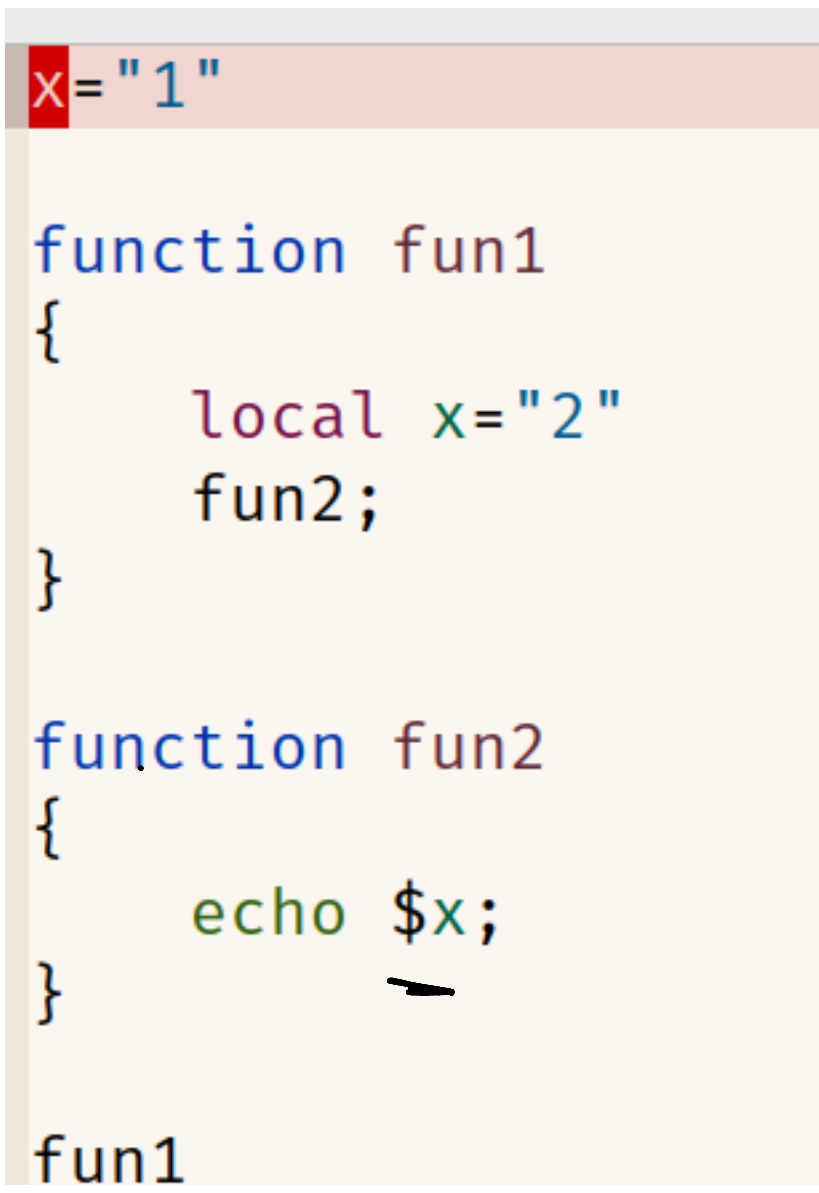


```
(define x 1)

(define fun1
  (lambda ()
    (let ((x 2))
      (fun2))))

(define fun2
  (lambda ()
    (display x)))

(fun1)
```



```
x="1"

function fun1
{
    local x="2"
    fun2;
}

function fun2
{
    echo $x;
}

fun1
```

Task: debug fun2

So... which x?

Static scoping: look at code (or use editor tools)

Dynamic:
I have to conside all
1000 places in code that
call fun2.

E.g. we call talloc from
everywhere.

```
void *talloc( ____ ) {
    ____
    ____
    ____
    list = ....
    ↑        ----_-
    global      malloc
                ---\
}
```

---

but buried, deep in code

```
void evilmaker() {
  Node *list = bad thing
  talloc()
} // if dynamically scoped,
          talloc uses wrong list
```

In hindsight, dynamic scoping, though seemingly intuitive was a bad idea, mostly.

- still here in langs w/ historical baggage
- arguments made that it still makes sense in highly interactive langs, designed to be used one command at a time, like shell scripting

---

Going forward, we need to implement static scoping in Scheme.