

# DELIMITED CONTINUATIONS DEMYSTIFIED

---

Alexis King, Tweag

ZuriHac 2023

# HISTORY

- Delimited continuations introduced by Matthias Felleisen 35 years ago.
- Flurry of initial publications, mostly in Scheme.
- Not much mainstream adoption.
- Recently: some renewed interest.

# Haskell

- Initial proposal in early 2020; revised version accepted in late 2020.
- Implementation in limbo for several years.
- Started at Tweag last year; patch landed last fall.
- Finally released this past March in GHC 9.6!

Problem: nobody knows what they are.

# DEMYSTIFICATION

# TERMINOLOGY

~~// continuations //~~

// native, first-class,  
delimited continuations //

① continuations

② delimited

③ first-class

④ native

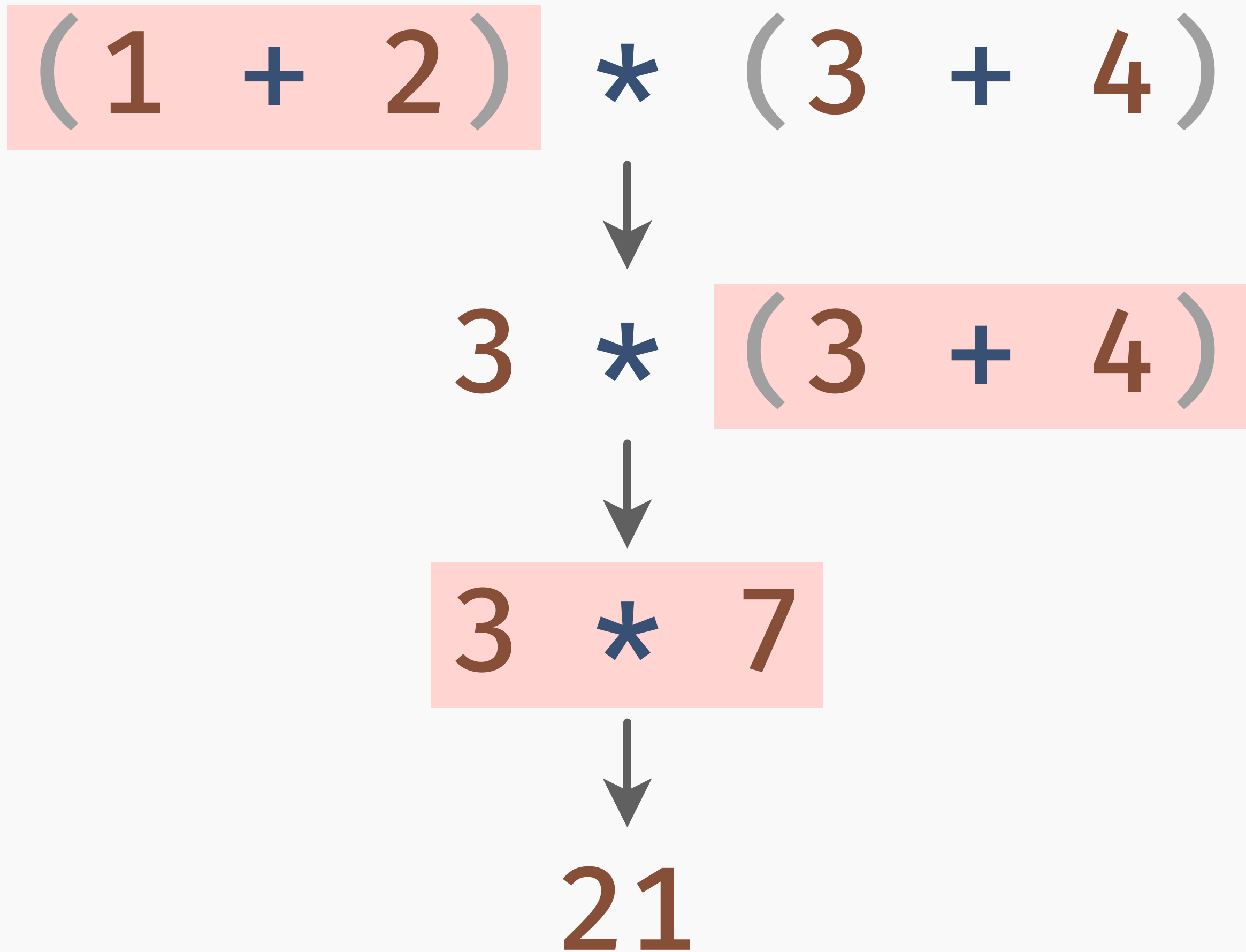
A “continuation” is a *concept*,  
not a language feature.

(Like “scope” or “value”.)

Applies to most programming languages!

Useful for talking about *evaluation*.





$(1 + 2)$

$* (3 + 4)$

$1 + 2$

$\bullet * (3 + 4)$

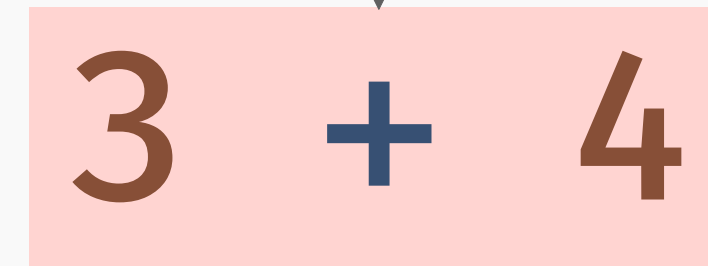
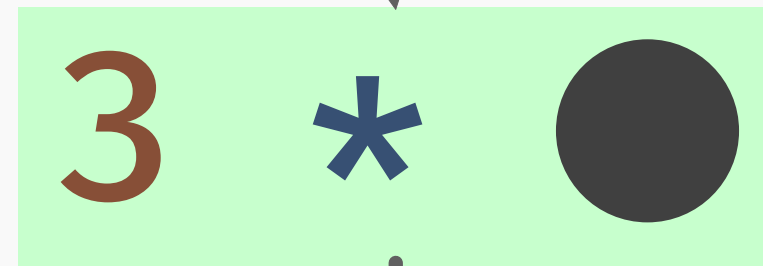
“redex”

“continuation”

$3$

$3$

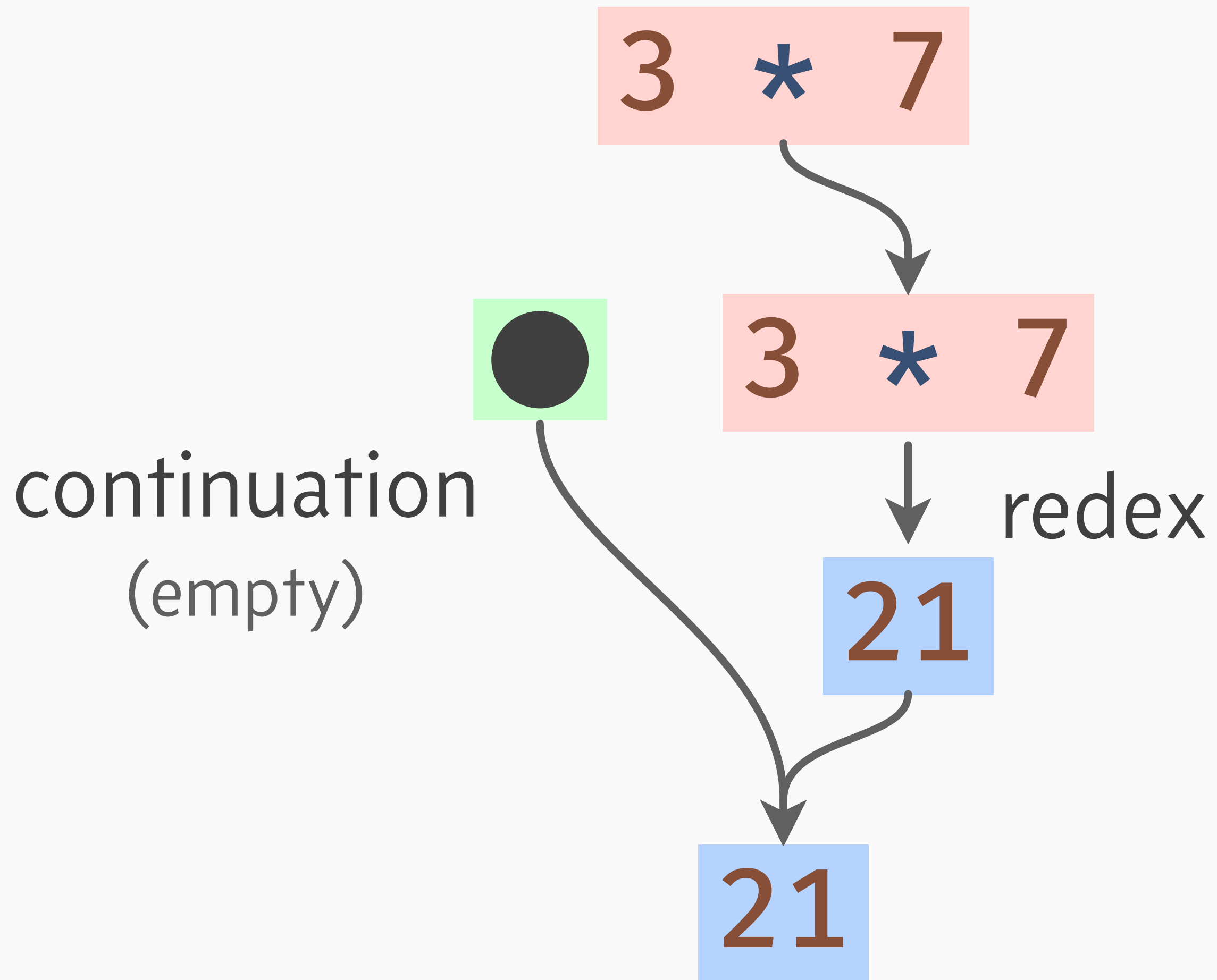
$* (3 + 4)$



continuation

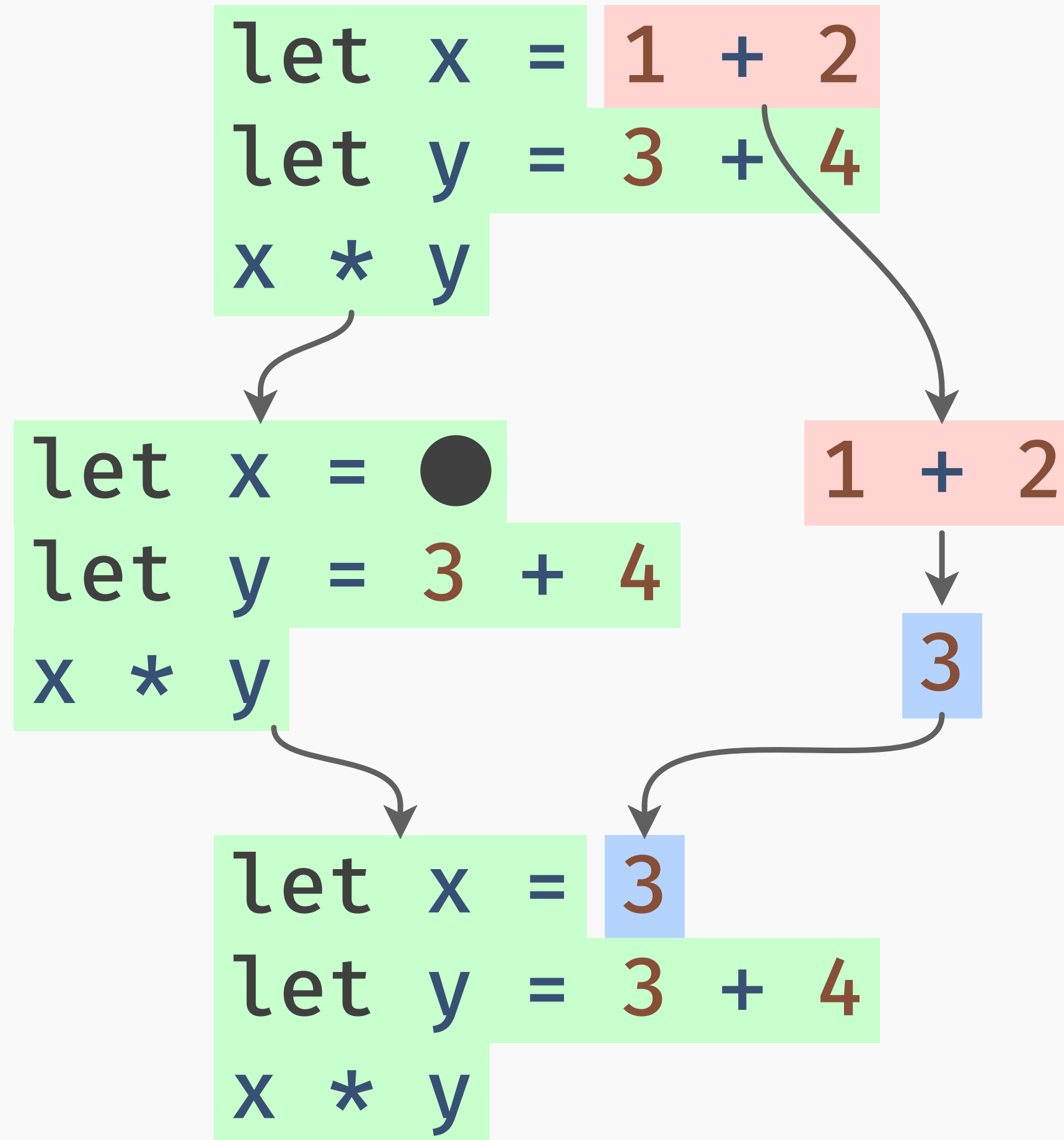
redex





# What is the continuation?

- The “context” in which the redex is evaluated.
- An expression with a hole.
- The place the redex’s value is “returned to”.
- “The rest of the program.”



```
let x = 3
```

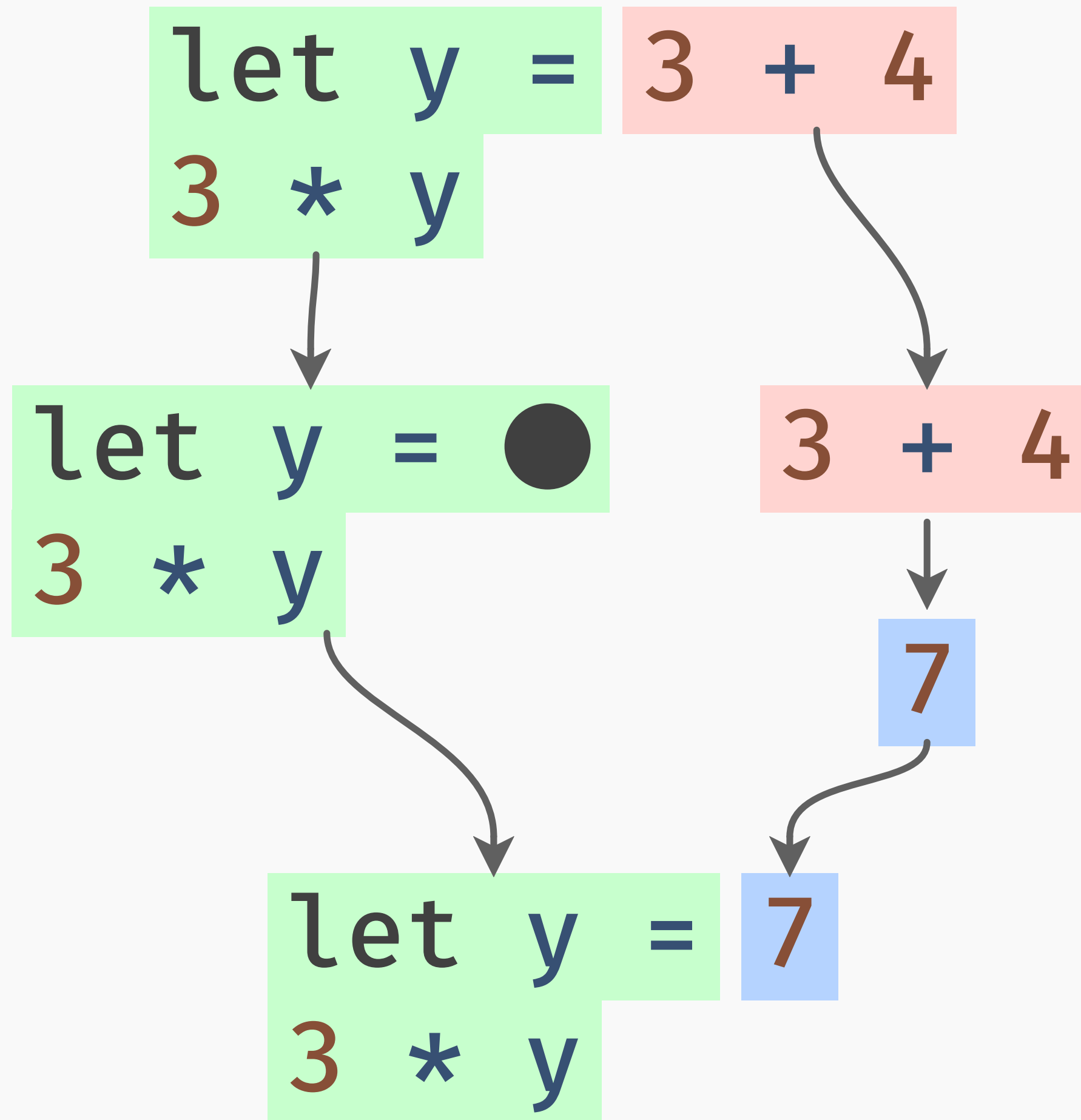
```
let y = 3 + 4
```

```
x * y
```



```
let y = 3 + 4
```

```
3 * y
```





# Why care about continuations?

Evaluation is *extremely* regular:

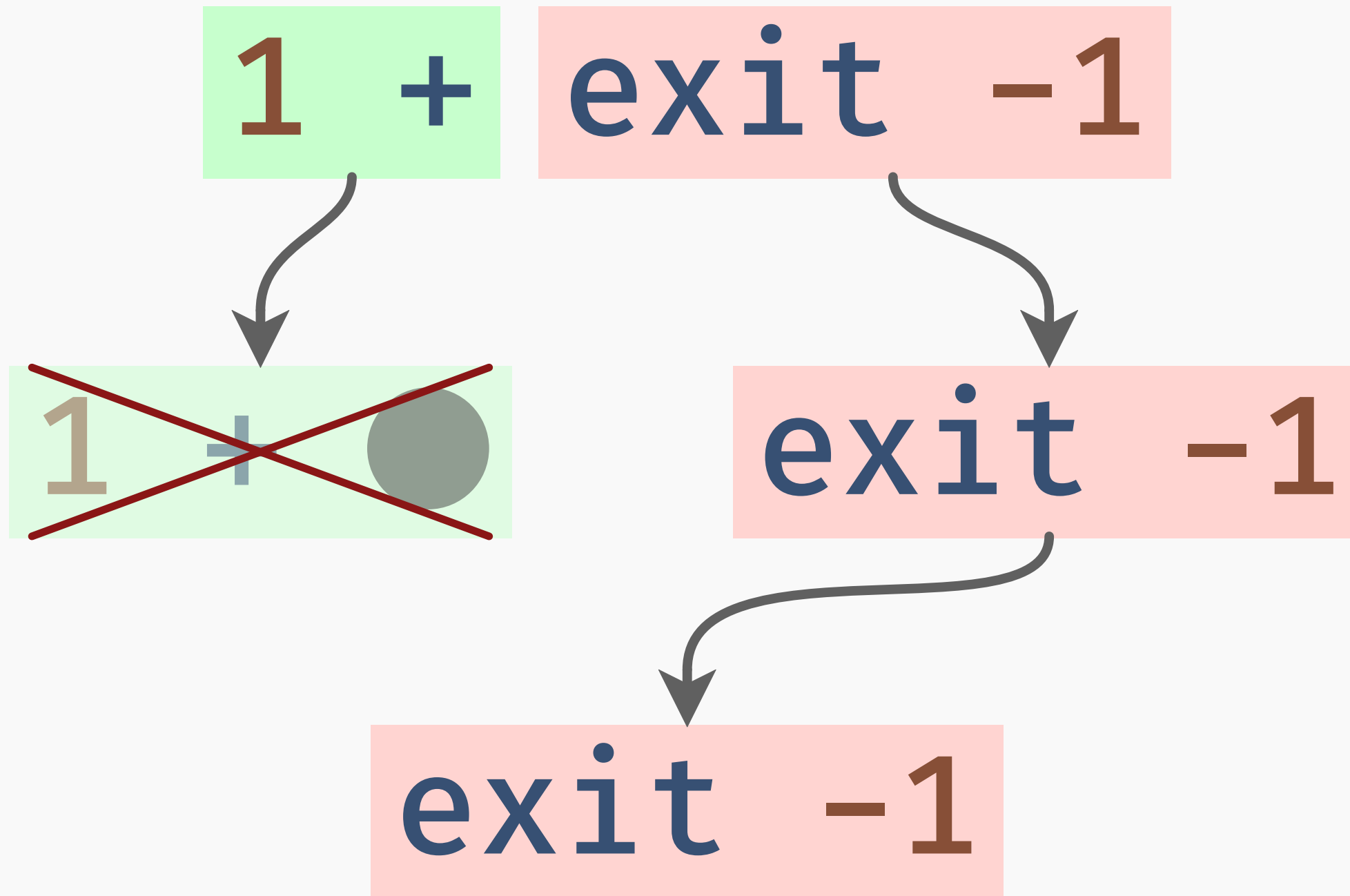
- ① Split the redex and continuation.
- ② Reduce the redex.
- ③ Substitute the result into the continuation.
- ④ Repeat.

Why is the continuation itself interesting?

Compiler writers care about the continuation!

Most programmers don't have much  
reason to, most of the time.

...but what about operators that use different rules?



Continuation is thrown away!

`exit` is still not terribly interesting.

# What about **throw** / **catch**?

## **throw** **exn**

Raises **exn** as an exception.

## **catch** **{body}** **handler**

Evaluates **body**, and if an exception is raised, evaluates **handler exn**.

1 + catch {2 \* throw 5}  
(\n → 3 \* n)

1 + (3 \* 5)

1 + 15

16

1 + catch {2 \* throw 5}  
(\n → 3 \* n)

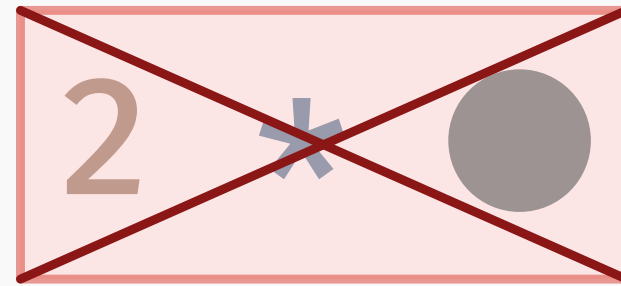
1 + catch {2 \* ●}  
(\n → 3 \* n)

throw 5

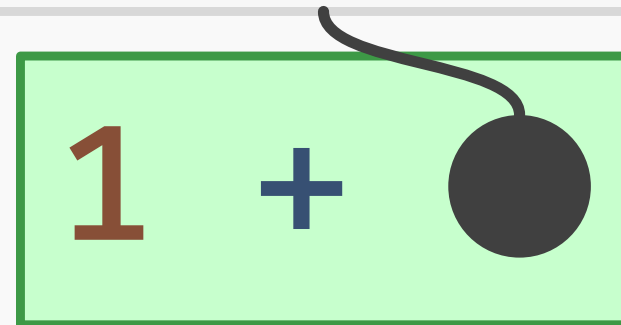
???

1 + (3 \* 5)

$1 + \text{catch } \{2 * \bullet\} (\backslash n \rightarrow 3 * n)$



$\text{catch } \{\bullet\} (\backslash n \rightarrow 3 * n)$



*catch delimits* the discarded continuation.



# INTERLUDE: NOTATION

$$A \longrightarrow B$$

" $A$  reduces to  $B$ ."

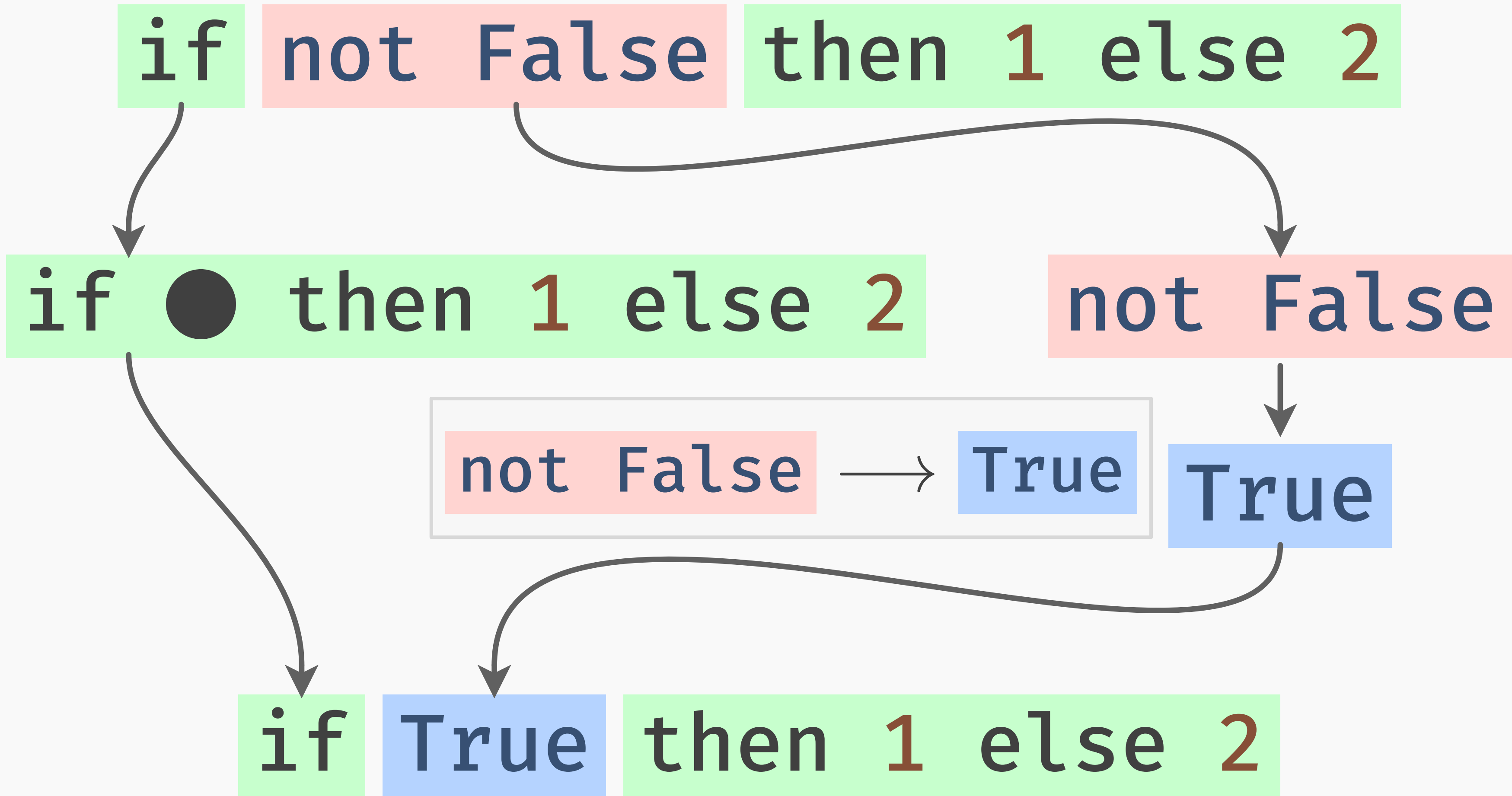
not False  $\longrightarrow$  True

not True  $\longrightarrow$  False

if True then  $e_1$  else  $e_2 \longrightarrow e_1$

if False then  $e_1$  else  $e_2 \longrightarrow e_2$

if not False then 1 else 2?



$$\cancel{\text{not False}} \longrightarrow \text{True}$$

$$E[\text{not False}] \longrightarrow E[\text{True}]$$

- $E$  stands for “some arbitrary continuation”.
- $E[x]$  denotes “plugging the hole” in  $E$  with  $x$ .

$E = \text{if } \bullet \text{ then } 1 \text{ else } 2$

$x = \text{not False}$

$E[x] = \text{if not False then } 1 \text{ else } 2$

Why bother with all of this?

$$E[\mathbf{exit} \ v] \longrightarrow \mathbf{exit} \ v$$

$$E_1[\mathbf{catch} \ \{E_2[\mathbf{throw} \ v]\} \ f] \longrightarrow E_1[f \ v]$$

Lots of operations can be described this way!

① continuations ✓

② delimited ✓

③ first-class

④ native

What makes something “first class”?

How could a *continuation* be a *value*?



$\backslash x \rightarrow 1 + (x * 2)$

$\backslash x \rightarrow \text{if } x > 0 \text{ then } 1 \text{ else } -1$

$\backslash x \rightarrow f \text{ (catch \{throw } x \} \text{ handle)}$

What is a “first-class continuation”?

Answer: a continuation reified as a function.

# call\_cc

“call with current continuation”

$$E[\text{call\_cc } f] \longrightarrow E[f] (\backslash x \rightarrow E[x])$$

This has some problems!

~~1 + (● \* 2)~~

```
print (1 + (● * 2))  
shutdown_runtime  
run_libc_atexit  
exit_process
```

We need more control!

prompt / control

$E_1[\text{prompt } \{E_2[\text{control } f]\}]$

$\longrightarrow E_1[f] (\backslash x \rightarrow E_2[x])$

**1** + prompt { **2** \* control ( \k → k **3** + k **5** ) }

**1** + ●

**2** \* ●

\k → k **3** + k **5**

let k = \x → **2** \* x  
k **3** + k **5**

**1** +

(let k = \x → **2** \* x  
k **3** + k **5**)

1 + prompt {2 \* control (\k → k 3 + k 5)}

↓

1 + (let k = \x → 2 \* x  
k 3 + k 5)

↓

1 + (6 + 10)

↓

1 + 16

↓

17

# Why is this so confusing?

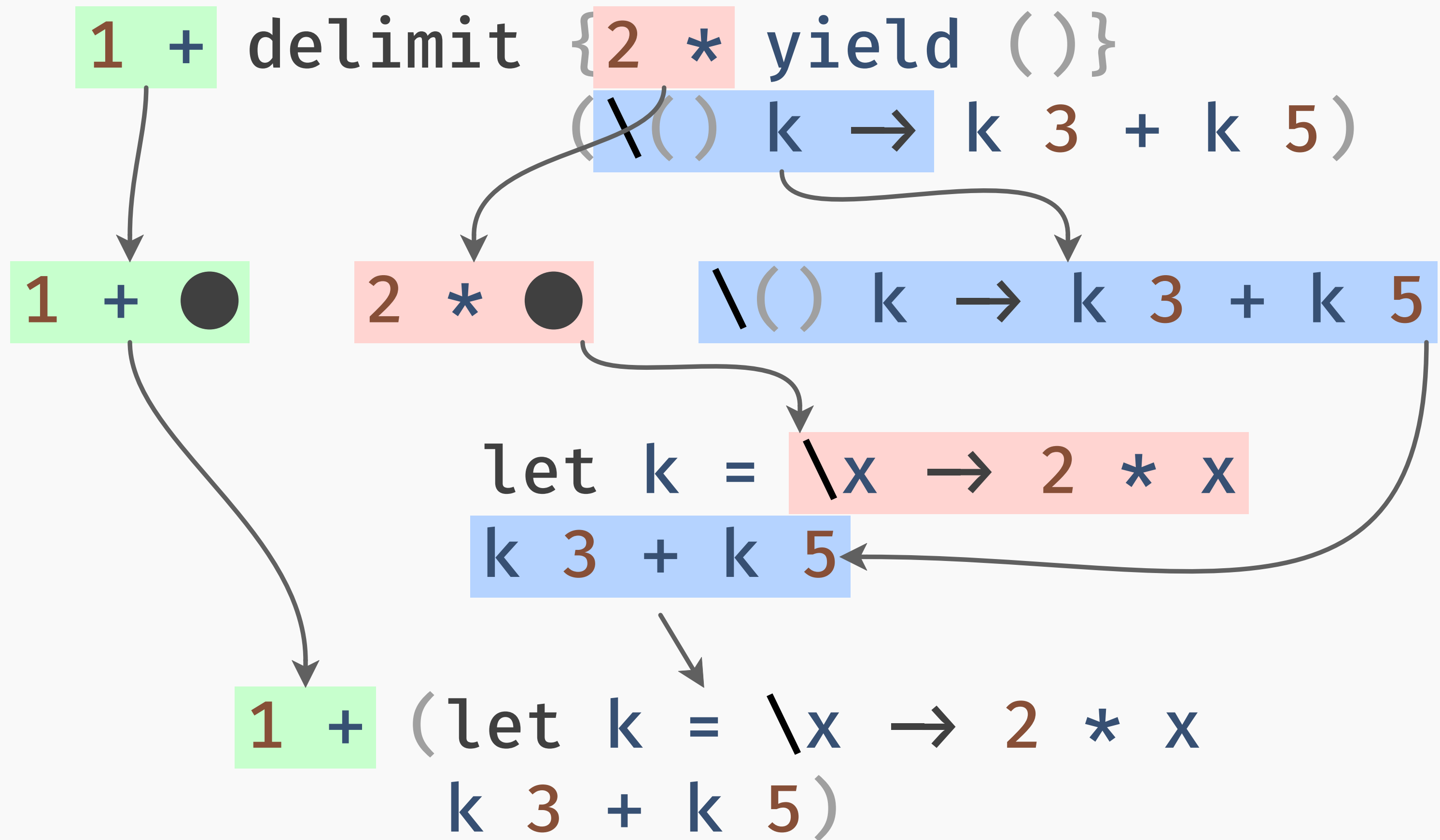
$$E_1[\text{catch } \{E_2[\text{throw } v]\} f] \longrightarrow E_1[f v]$$

$$E_1[\text{prompt } \{E_2[\text{control } f]\}] \longrightarrow E_1[f (\backslash x \rightarrow E_2[x])]$$

$$E_1[\text{delimit } \{E_2[\text{yield } v]\} f] \longrightarrow E_1[f v \backslash x \rightarrow E_2[x]]$$

`delimit` / `yield` provide *resumable exceptions*.





# Why prompt / control?

- In some sense “simpler”.
- Historical relationship to `call_cc`.
- Easier to statically type.

# TYPES

# Even typing exceptions is hard!

`throw :: Exception → a`

`catch {body} handler :: b`

`body :: b`

`handler :: Exception → b`

`yield :: DelimiterTag → a`

`delimit {body} handler :: b`

`body :: b`

`handler :: DelimiterTag → (a → b) → b`

$$E_1[\text{delimit } \{E_2[\text{yield } v]\} f] \\ \longrightarrow E_1[f \ v \ (\backslash x \rightarrow E_2[x])]$$

**prompt {body} :: b**

**body :: b**

**control :: ((a → b) → b) → a**

$E_1[\text{prompt } \{E_2[\text{control } f]\}]$

$\longrightarrow E_1[f \ (\backslash x \rightarrow E_2[x])]$

Solution: tagged prompts.

$\text{new\_prompt\_tag} :: () \rightarrow \text{PromptTag } b$

$\text{prompt tag \{body\}} :: b$

$\text{tag} :: \text{PromptTag } b$

$\text{body} :: b$

$\text{control} :: \text{PromptTag } b \rightarrow ((a \rightarrow b) \rightarrow b) \rightarrow a$

$E_1[\text{prompt tag } \{E_2[\text{control tag } f]\}]$

$\longrightarrow E_1[f (\backslash x \rightarrow E_2[x])]$



① continuations ✓

② delimited ✓

③ first-class ✓

④ native

# How do we implement this?

Option one: continuation-passing style.

Problem: slow! (See my talk from ZuriHac 2020.)

Option two: bake them into the runtime.

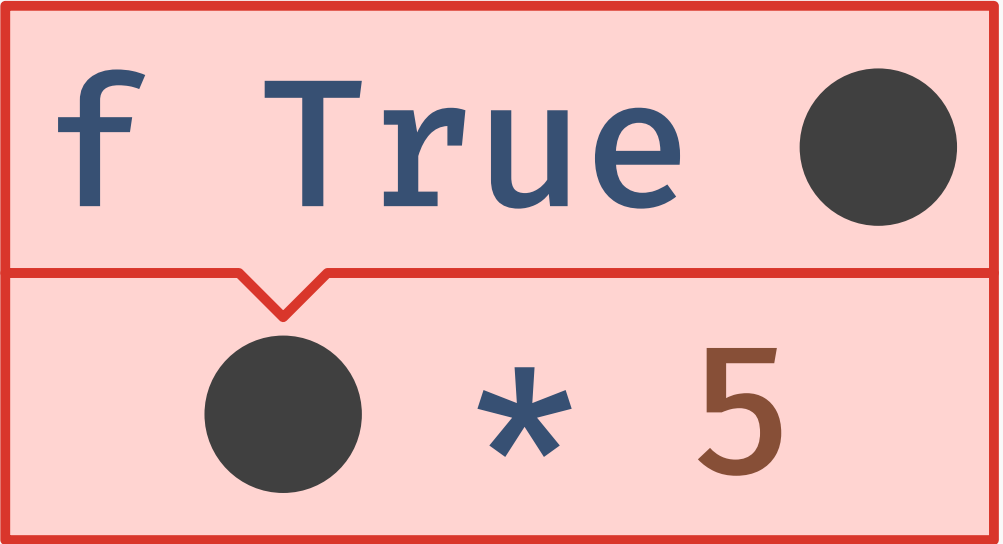
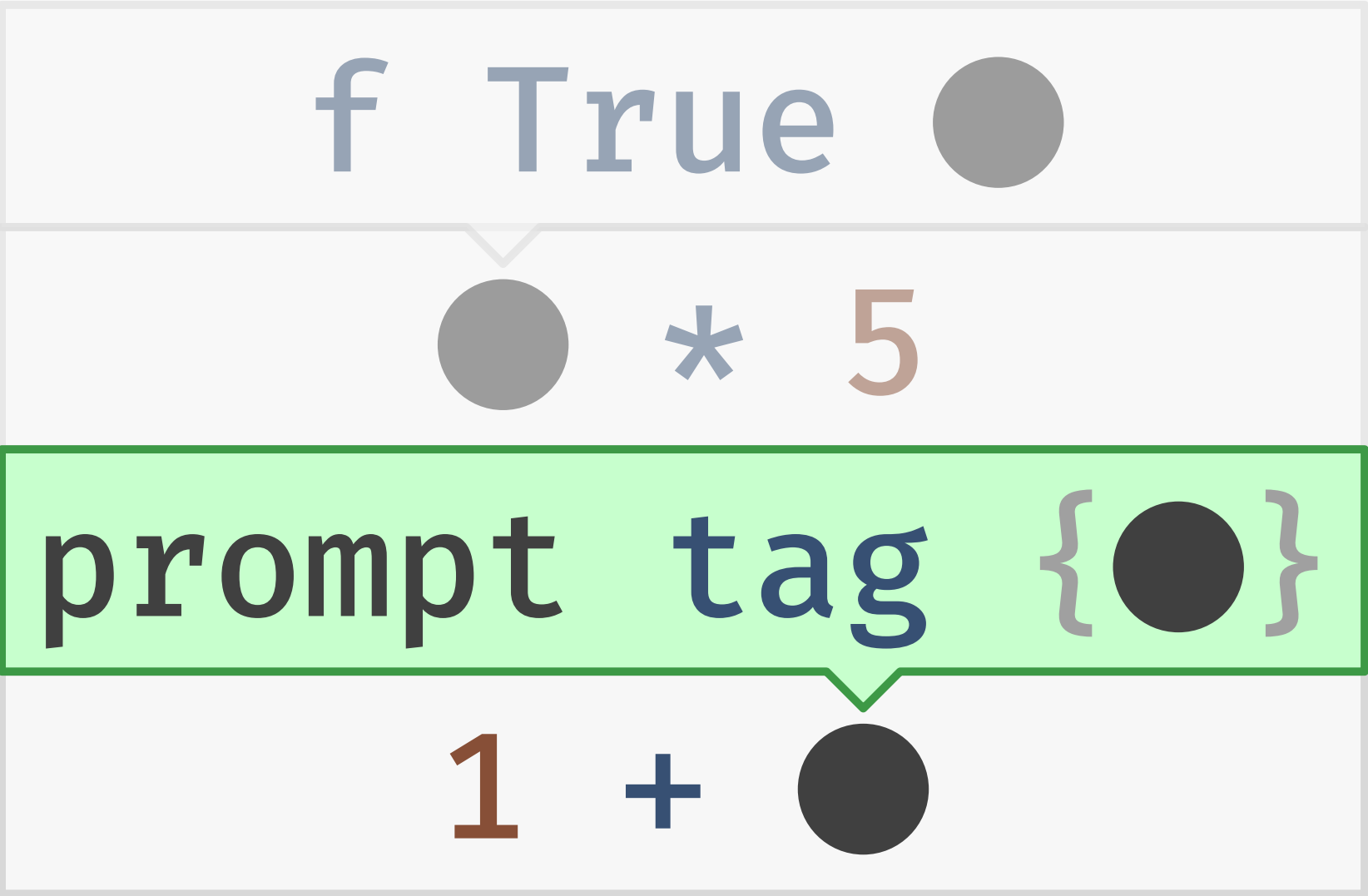
1 + prompt tag {f True ● \* 5}



This is a call stack!

redex: **control** tag g

stack:



redex: **g**

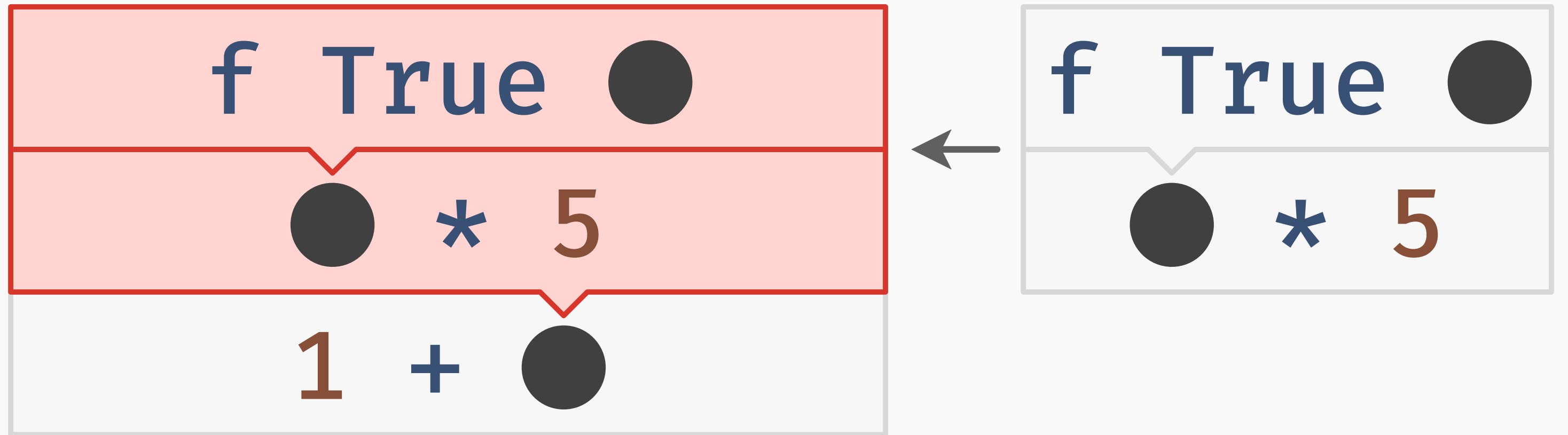


stack:



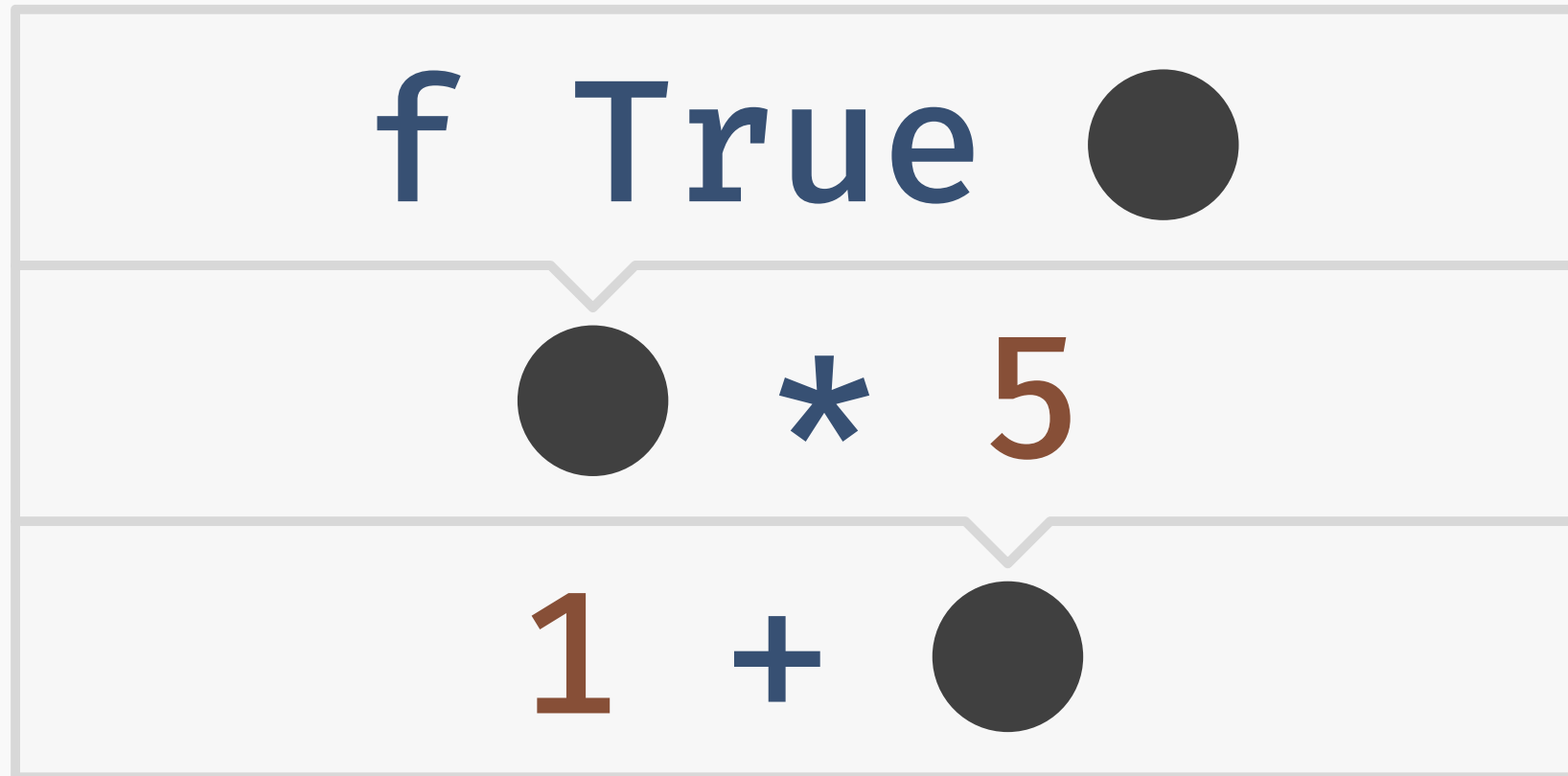
redex: "hello"

stack:



redex: "hello"

stack:



Capture/restore are just **memcpy**!

① continuations ✓

② delimited ✓

③ first-class ✓

④ native ✓



# MISCELLANY

- Can further optimize implementation for specific use cases.
- Strict monads permit embedding into a lazy language.
- Reality is always at least a little more complicated (e.g. stack overflow, async exceptions).
- We sorely lack non-synthetic continuation benchmarks!

The unsung hero of this talk:  
reduction semantics.

- ① continuations ✓
- ② delimited ✓
- ③ first-class
- ④ native

Still extremely useful!

- Continuations are a concept that arises naturally in evaluation.
- Special operators like **catch** delimit portions of the continuation.
- First-class continuations allow reifying the continuation as a function.
- Remarkably, this corresponds to manipulation of the call stack.

Thanks!

me: <https://lexi-lambda.github.io/>

[https://twitter.com/lexi\\_lambda](https://twitter.com/lexi_lambda)

Tweag: <https://www.tweag.io/>