DELIMITED CONTINUATIONS DEMYSTIFIED

Alexis King, Tweag

Lambda Days 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

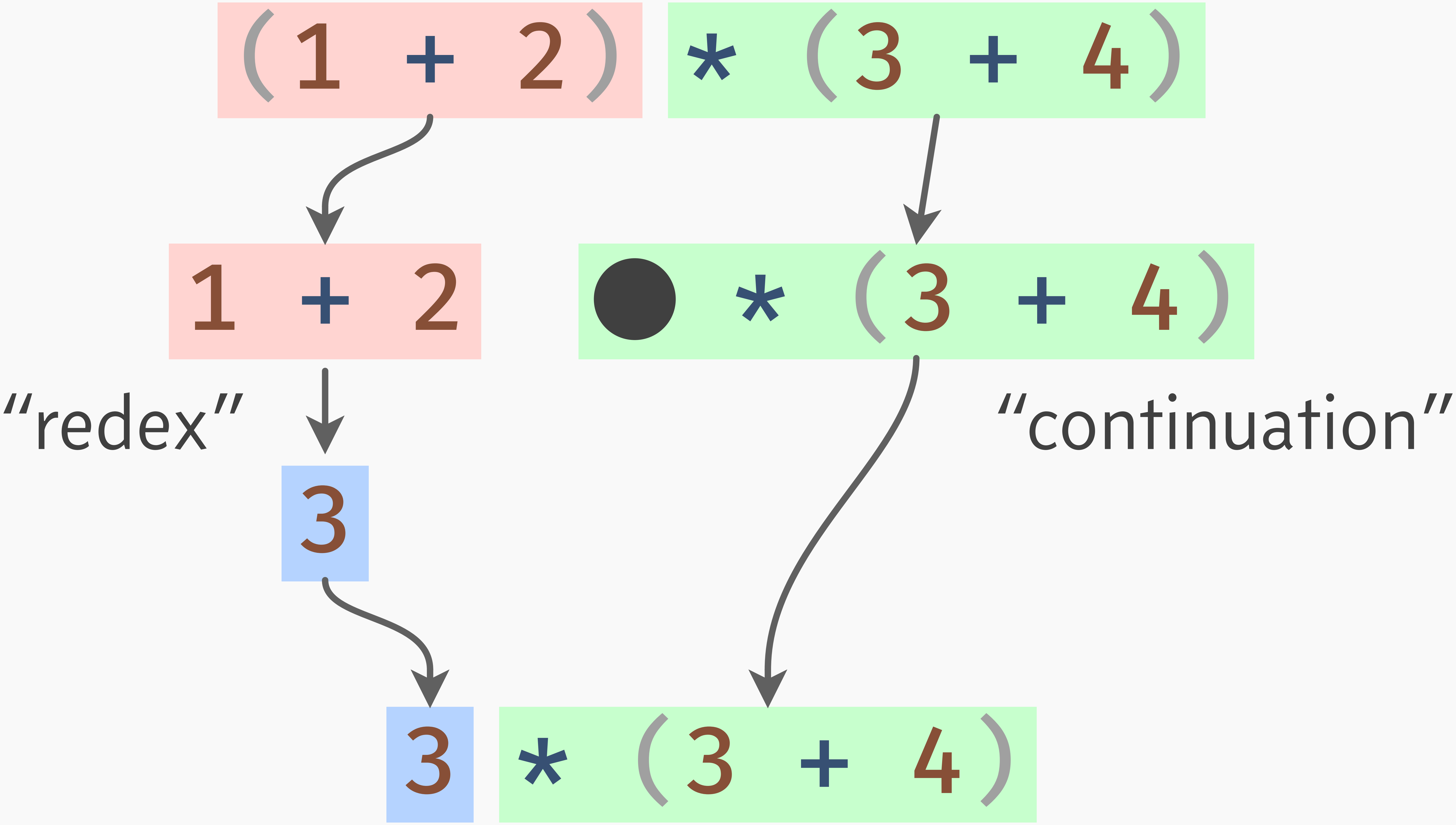
A “continuation” is a *concept*,
not a language feature.

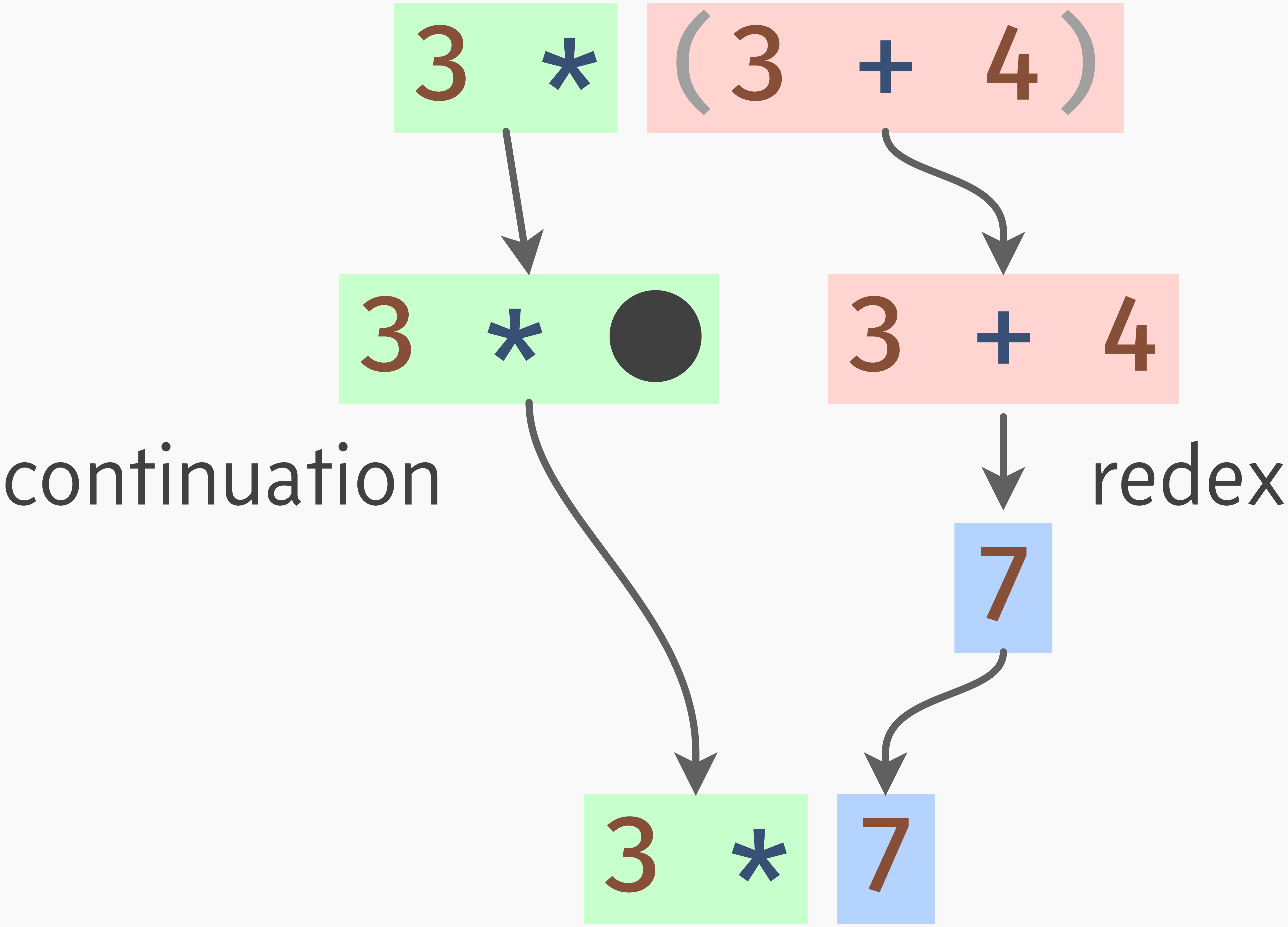
(Like “scope” or “value”.)

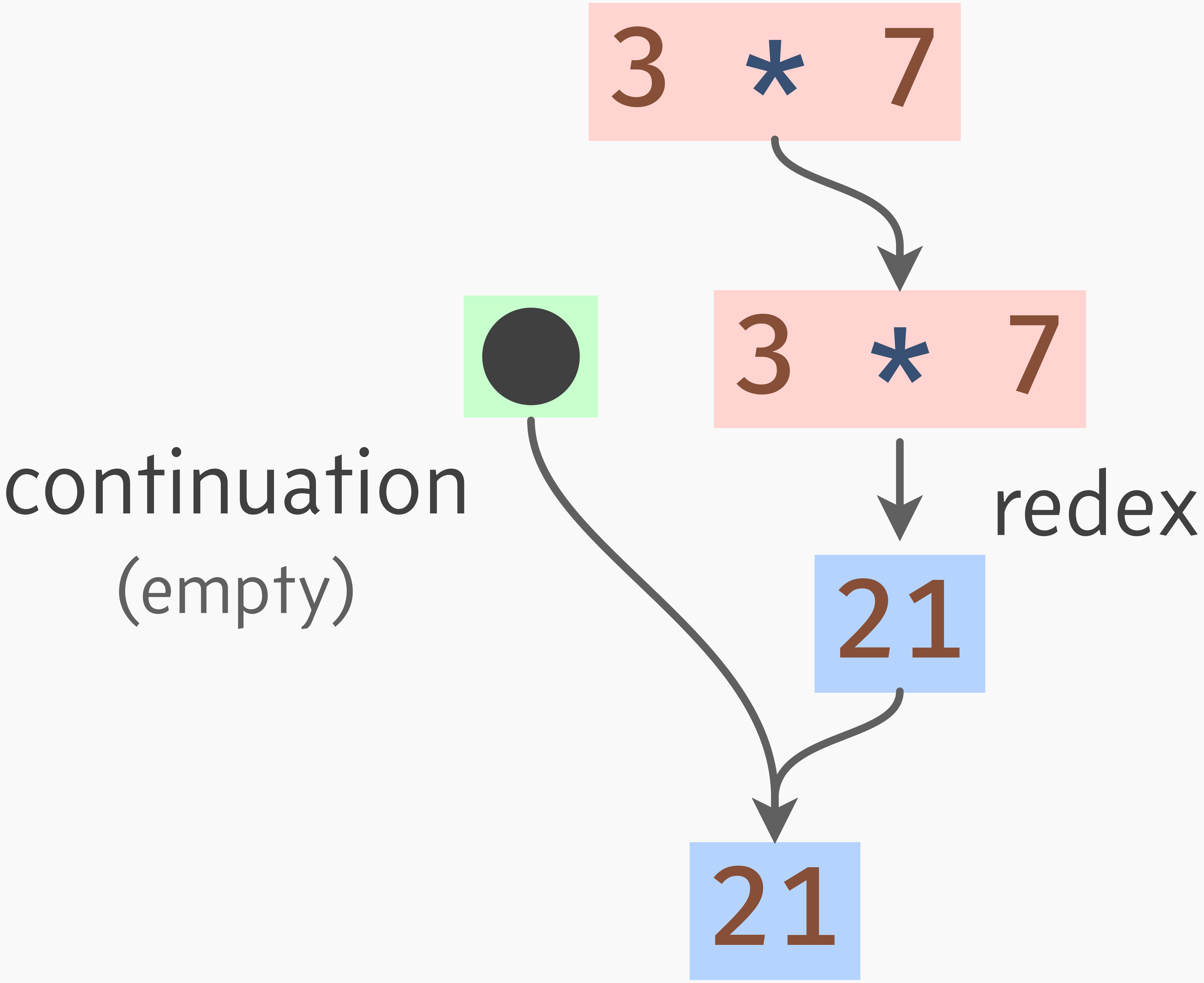
Applies to most programming languages!

Useful for talking about *evaluation*.

```
(1 + 2) * (3 + 4)
3 * (3 + 4)
3 * 7
21
```


(1 + 2) * (3 + 4)
1 + 2
● * (3 + 4)
"redex"
"continuation"
3
3 * (3 + 4)


3 * (3 + 4)
3 * ●
3 + 4
continuation
redex
7
3 * 7



3 * 7
3 * 7
continuation
(empty)
redex
21
21

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 = 1 + 2
let y = 3 + 4
x * y
```

```
let x = ●
let y = 3 + 4
x * y
```

```
1 + 2
```

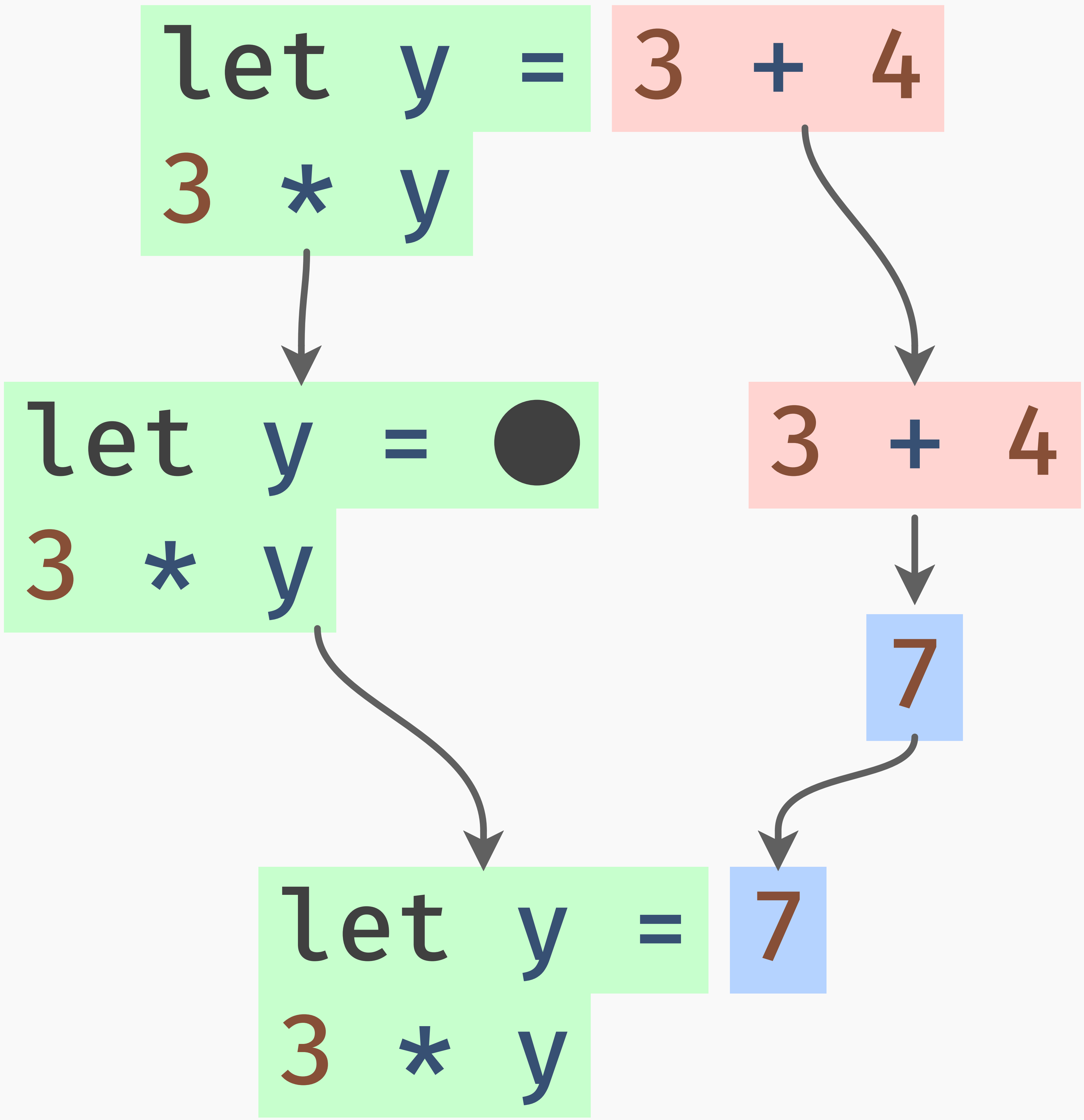
```
3
```

```
let x = 3
let y = 3 + 4
x * y
```

```
let x = 3
let y = 3 + 4
x * y
```

↓

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


let y = 3 + 4
3 * y
let y = ●
3 * y
3 + 4
7
let y = 7
3 * y

Why care about continuations?

Evaluation is *extremely* regular:

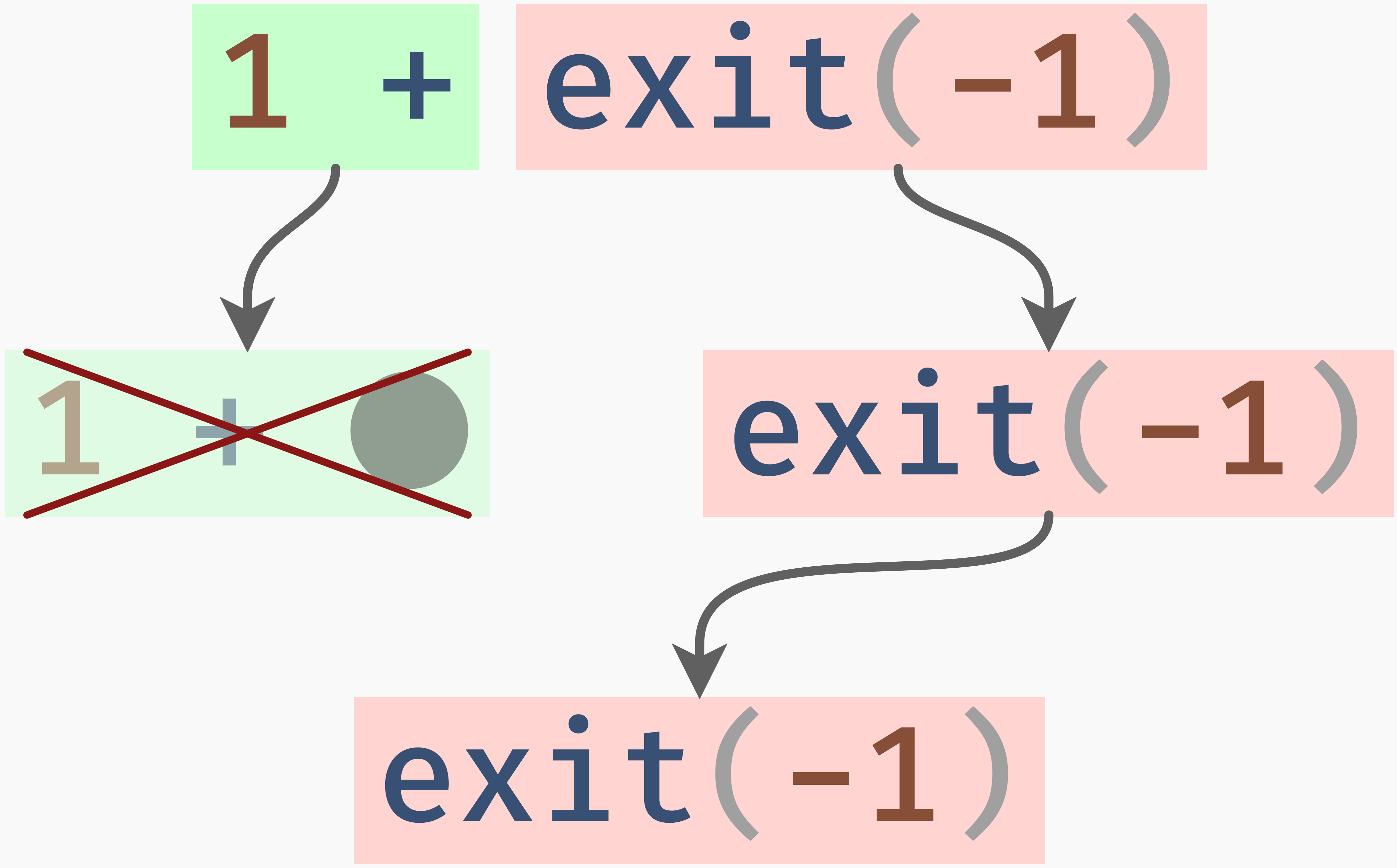
1. Split the redex and continuation.
2. Reduce the redex.
3. Substitute the result into the continuation.
4. Repeat.

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)
```

↓

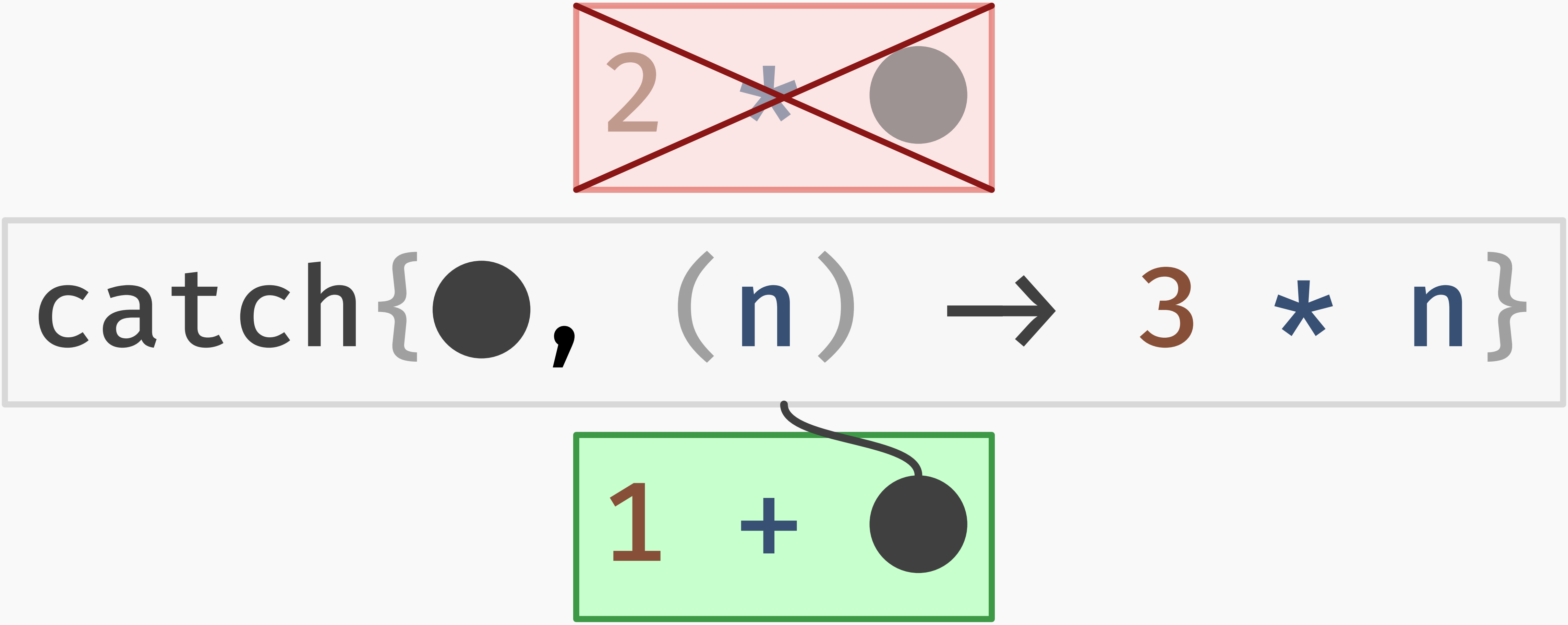
```
1 + 15
```

↓

```
16
```

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

1 + catch{2 * ●,
          (n) → 3 * n}

throw(5)

???

1 + (3 * 5)
```

```
1 + catch{2 * ●, (n) → 3 * n}
```



`catch` *delimits* the discarded continuation.

INTERLUDE: NOTATION

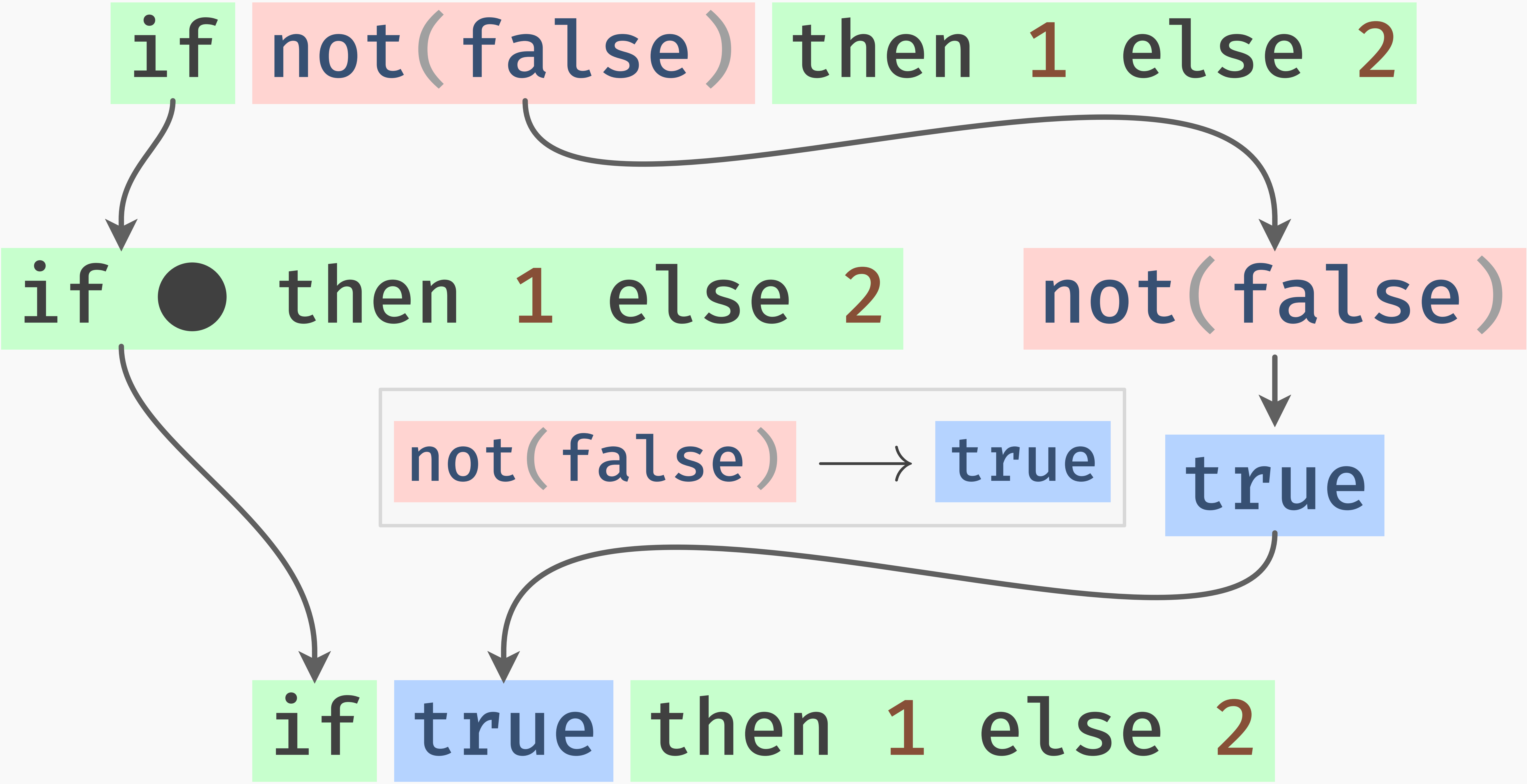
$$A \longrightarrow B$$

"A reduces to B."

$$\texttt{not(false)} \longrightarrow \texttt{true}$$
$$\texttt{not(true)} \longrightarrow \texttt{false}$$

$$\texttt{if true then } e_1 \texttt{ else } e_2 \longrightarrow e_1$$
$$\texttt{if false then } e_1 \texttt{ else } e_2 \longrightarrow e_2$$

$$\texttt{if not(false) then 1 else 2}?$$


if not(false) then 1 else 2
if ● then 1 else 2
not(false)
not(false) ⟶ true
true
if true then 1 else 2

$$\cancel{\texttt{not(false)} \longrightarrow \texttt{true}}$$

$$E[\texttt{not(false)}] \longrightarrow E[\texttt{true}]$$

→ E stands for “some arbitrary continuation”.

→ $E[x]$ denotes “plugging the hole” in E with x.

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

$$x = \texttt{not(false)}$$

$$E[x] = \texttt{if not(false) then 1 else 2}$$

Why bother with all of this?

$$E[\mathtt{exit}(v)] \longrightarrow \mathtt{exit}(v)$$

$$E_1[\mathtt{catch}\{E_2[\mathtt{throw}(v)],\ f\}] \longrightarrow E_1[f(v)]$$

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*?

```
(x) → 1 + (x * 2)

(x) → if x > 0 then 1 else -1

(x) → f(catch{throw(x), handle})
```

What is a “first-class continuation”?

Answer: a continuation reified as a function.

`call_cc`

"call with current continuation"

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

This has some problems!

~~1 + (● * 2)~~

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

We need more control!

`prompt` / `control`

$$E_1[\texttt{prompt}\{E_2[\texttt{control}(f)]\}]$$

$$\longrightarrow E_1[f((\texttt{x}) \to E_2[\texttt{x}])]$$

```
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
```

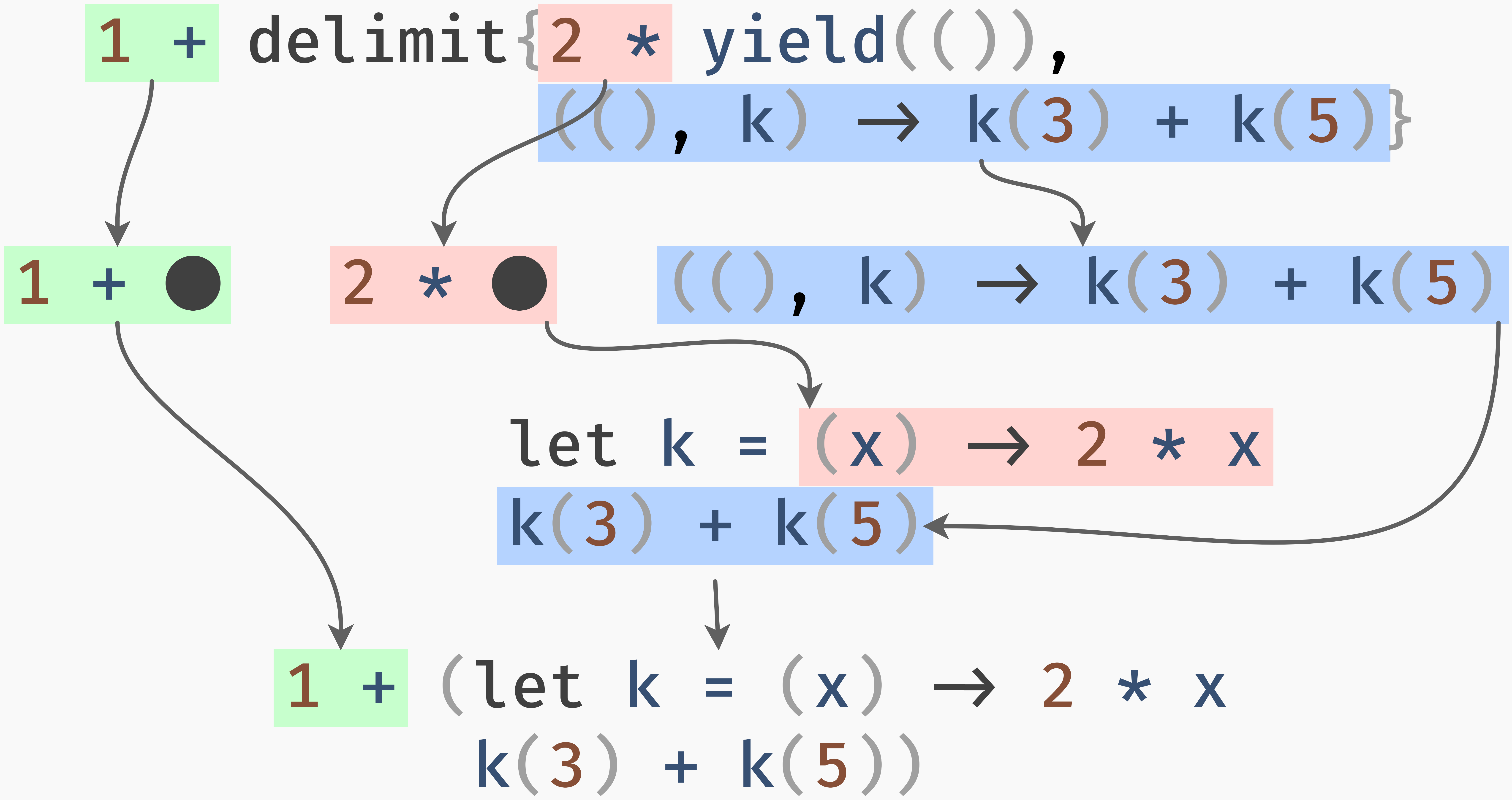
Why is this so confusing?

$$E_1[\texttt{catch}\{E_2[\texttt{throw}(v)], f\}] \longrightarrow E_1[f(v)]$$

$$E_1[\texttt{prompt}\{E_2[\texttt{control}(f)]\}] \longrightarrow E_1[f((\texttt{x}) \to E_2[\texttt{x}])]$$

$$E_1[\texttt{delimit}\{E_2[\texttt{yield}(v)], f\}] \longrightarrow E_1[f(v, (\texttt{x}) \to E_2[\texttt{x}])]$$

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


1 + delimit{2 * yield(()),
((), 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))

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
```

$$E_1[\texttt{delimit}\{E_2[\texttt{yield}(v)],\ f\}]$$
$$\longrightarrow E_1[f(v,\ (\texttt{x}) \to E_2[\texttt{x}])]$$

$$\texttt{prompt}\{\texttt{body}\} : \boxed{\texttt{b}}$$

$$\texttt{body} : \boxed{\texttt{b}}$$

$$\texttt{control} : ((\texttt{a} \to \boxed{\texttt{b}}) \to \boxed{\texttt{b}}) \to \texttt{a}$$

$$E_1[\boxed{\texttt{prompt}\{E_2[\texttt{control}(f)]\}}]$$

$$\longrightarrow E_1[f((\texttt{x}) \to E_2[\texttt{x}])]$$

Solution: tagged prompts.

```
new_prompt_tag : () → PromptTag<b>

prompt{tag, body} : b
tag : PromptTag<b>
body : b

control : (PromptTag<b>, ((a → b) → b)) → a
```

$$E_1[\texttt{prompt}\{tag,\ E_2[\texttt{control}(tag,\ f)]\}]$$
$$\longrightarrow E_1[f((\texttt{x}) \to E_2[\texttt{x}])]$$

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}
```

```
f(true, ●)
● * 5
prompt{tag, ●}
1 + ●
```

This is a call stack!

redex: `control(tag, g)`

stack:

```
f(true, ●)
● * 5
prompt{tag, ●}
1 + ●
```

→

```
f(true, ●)
● * 5
```

redex: g(CONT ○)

stack: 1 + ●

f(true, ●)

● * 5

redex: "hello"

stack:

```
f(true, ●)
● * 5
1 + ●
```

```
f(true, ●)
● * 5
```

redex: `"hello"`

stack:

```
f(true, ●)
● * 5
1 + ●
```

```
f(true, ●)
● * 5
```

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