Control Flow in Common Lisp

aka Why Lisp Doesn't Need To Throw Exceptions

Michał “phoe” Herda

phoe@disroot.org

https://phoe.github.io

If this presentation only had a single sentence

If this presentation only had a single sentence

- We need means of **unwinding** and **finally blocks**

If this presentation only had a single slide

- We need means of **unwinding** and **finally blocks**

If this presentation only had a single slide

- We need means of **unwinding** and **finally blocks**
  - We can implement all of CL-specific control flow with this!

If this presentation only had a single slide

- We need means of **unwinding** and **finally blocks**
  - We can implement all of CL-specific control flow with this!
    - `if`
    - `tagbody/go`
    - `block/return-from`
    - `catch/throw`
    - `unwind-protect`
    - `lambda/apply`

If this presentation only had a single slide

- We need means of **unwinding** and **finally blocks**
  - We can implement all of CL-specific control flow with this!
    - `if`
    - `tagbody/go`
    - `block/return-from`
    - `catch/throw`
    - `unwind-protect`
    - `lambda/apply`

- This includes non-local jumps (a secret Common Lisp feature)

If this presentation only had a single slide

- We need means of **unwinding** and **finally blocks**
  - We can implement all of CL-specific control flow with this!
    - `if`
    - `tagbody/go`
    - `block/return-from`
    - `catch/throw`
    - `unwind-protect`
    - `lambda/apply`

- This includes non-local jumps (a secret Common Lisp feature)

- This includes loops
- This includes switches
- This includes error handling
- This includes restart handling

If this presentation only had a single slide

- We need means of **unwinding** and **finally blocks**
  - We can implement all of CL-specific control flow with this!
    - `if`
    - `tagbody/go`
    - `block/return-from`
    - `catch/throw`
    - `unwind-protect`
    - `lambda/apply`

- This includes non-local jumps (a secret Common Lisp feature)

- This includes loops
- This includes switches
- This includes error handling
- This includes restart handling
- **The above four groups are implemented in Common Lisp itself**

Let’s talk about Common Lisp

Let’s talk about Common Lisp

- Compiled multiparadigm interactive programming language

Let’s talk about Common Lisp

- Compiled multiparadigm interactive programming language
- ANSI-standardized in 1994

Let’s talk about Common Lisp

- Compiled multiparadigm interactive programming language
- ANSI-standardized in 1994
- Multiple conforming implementations with different qualities

Let’s talk about Common Lisp

- Compiled multiparadigm interactive programming language
- ANSI-standardized in 1994
- Multiple conforming implementations with different qualities
- Relatively small but active community

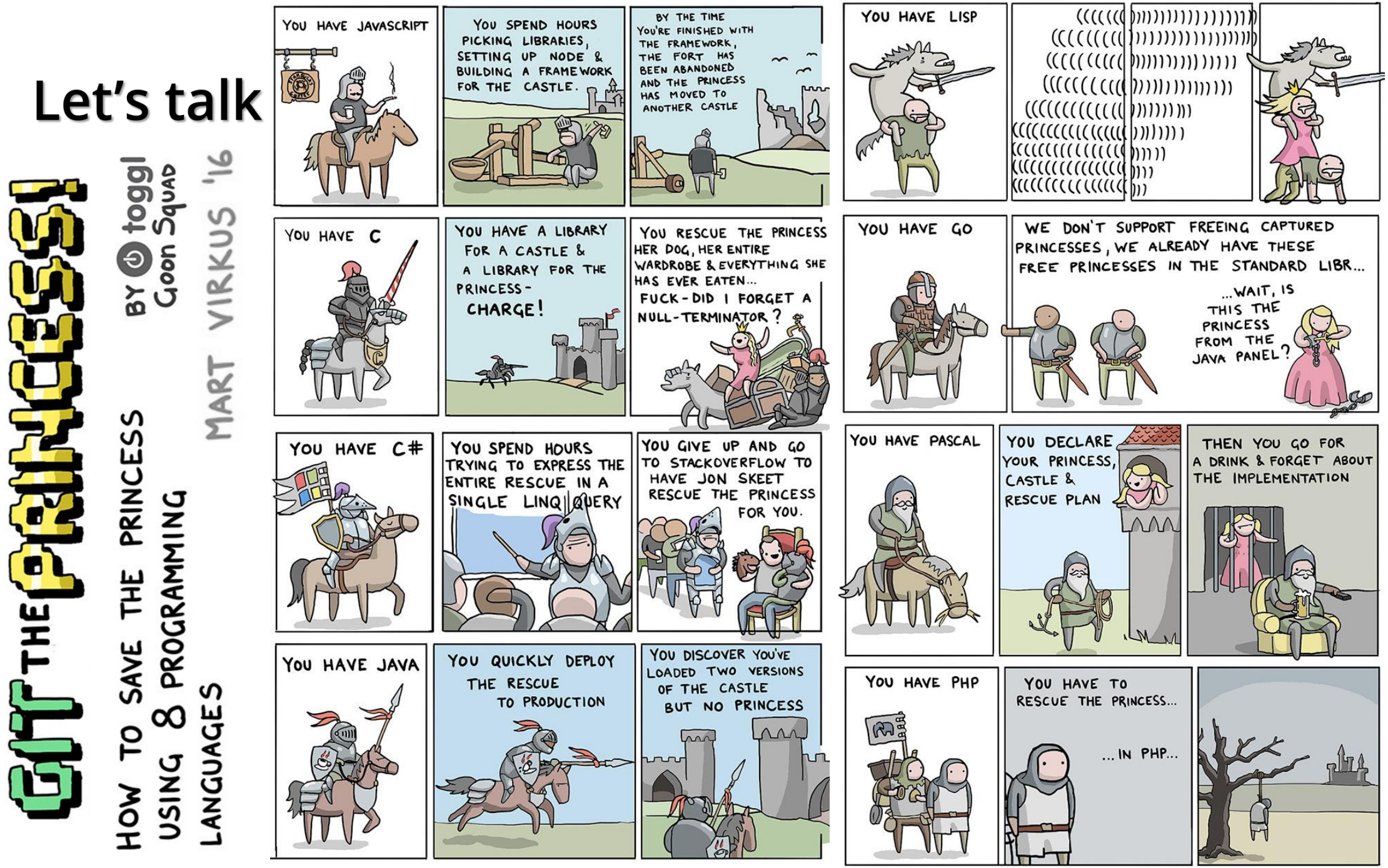
Let’s talk about Common Lisp

- Compiled multiparadigm interactive programming language
- ANSI-standardized in 1994
- Multiple conforming implementations with different qualities
- Relatively small but active community
- Continuously used for commercial projects and research

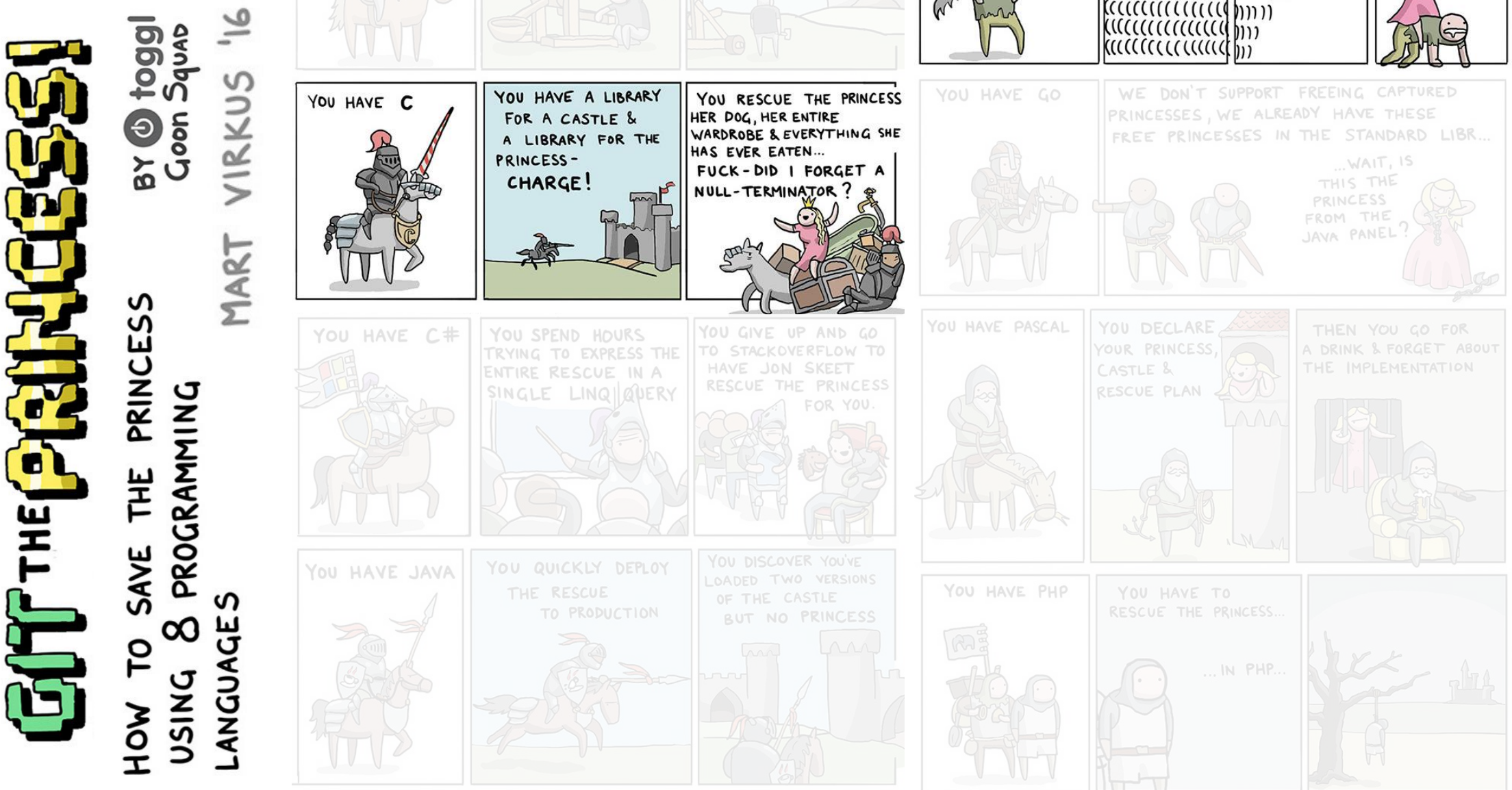
Let’s talk about Common Lisp

Let’s talk about Common Lisp

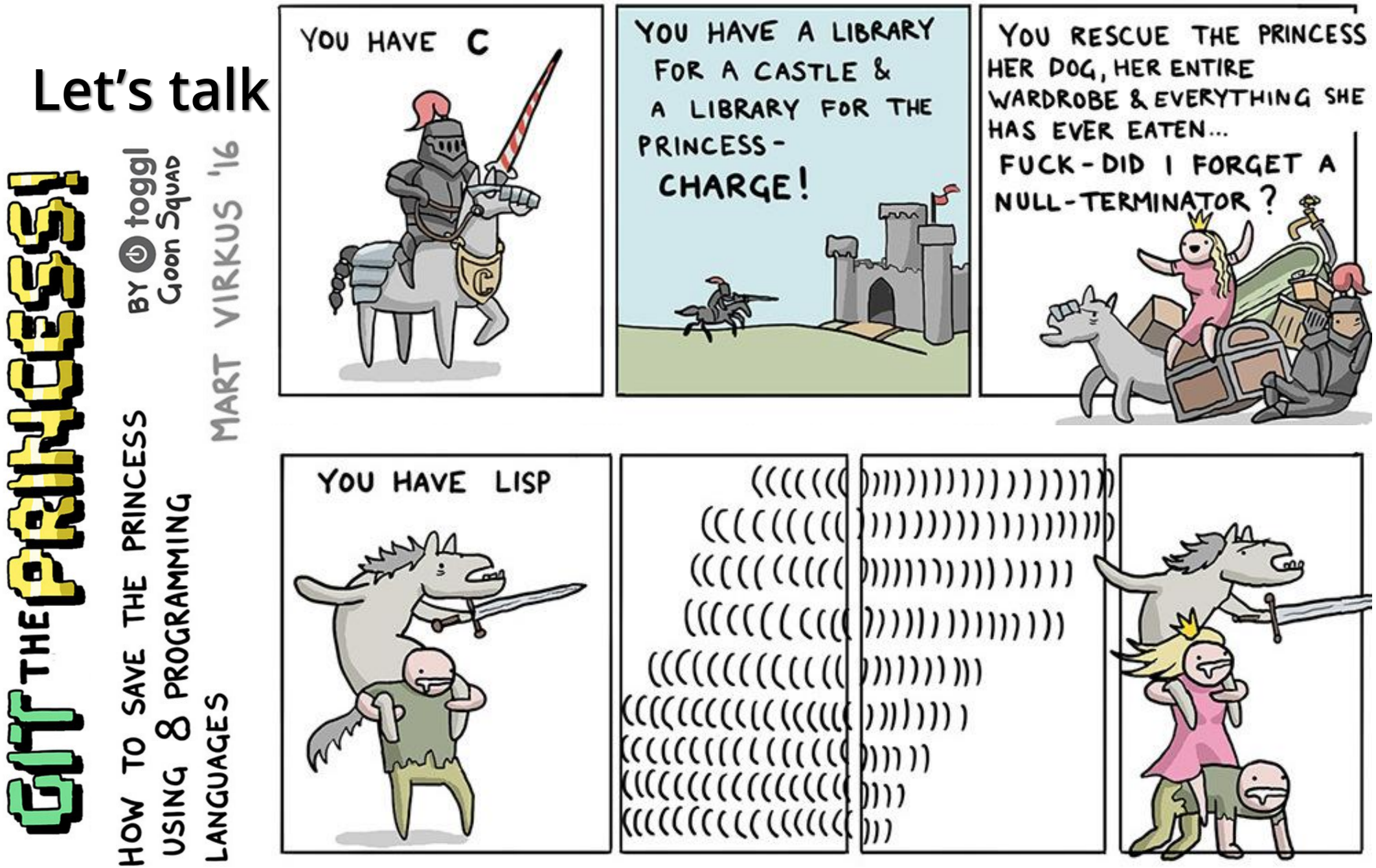
Let's talk


GIT THE PRINCESS!
HOW TO SAVE THE PRINCESS USING 8 PROGRAMMING LANGUAGES
BY toggl Goon Squad
MART VIRKUS '16
YOU HAVE JAVASCRIPT
YOU SPEND HOURS PICKING LIBRARIES, SETTING UP NODE & BUILDING A FRAMEWORK FOR THE CASTLE.
BY THE TIME YOU'RE FINISHED WITH THE FRAMEWORK, THE FORT HAS BEEN ABANDONED AND THE PRINCESS HAS MOVED TO ANOTHER CASTLE
YOU HAVE C
YOU HAVE A LIBRARY FOR A CASTLE & A LIBRARY FOR THE PRINCESS - CHARGE!
YOU RESCUE THE PRINCESS HER DOG, HER ENTIRE WARDROBE & EVERYTHING SHE HAS EVER EATEN... FUCK - DID I FORGET A NULL-TERMINATOR?
YOU HAVE C#
YOU SPEND HOURS TRYING TO EXPRESS THE ENTIRE RESCUE IN A SINGLE LINQ QUERY
YOU GIVE UP AND GO TO STACKOVERFLOW TO HAVE JON SKEET RESCUE THE PRINCESS FOR YOU.
YOU HAVE JAVA
YOU QUICKLY DEPLOY THE RESCUE TO PRODUCTION
YOU DISCOVER YOU'VE LOADED TWO VERSIONS OF THE CASTLE BUT NO PRINCESS
YOU HAVE LISP
YOU HAVE GO
WE DON'T SUPPORT FREEING CAPTURED PRINCESSES, WE ALREADY HAVE THESE FREE PRINCESSES IN THE STANDARD LIBR...
...WAIT, IS THIS THE PRINCESS FROM THE JAVA PANEL?
YOU HAVE PASCAL
YOU DECLARE YOUR PRINCESS, CASTLE & RESCUE PLAN
THEN YOU GO FOR A DRINK & FORGET ABOUT THE IMPLEMENTATION
YOU HAVE PHP
YOU HAVE TO RESCUE THE PRINCESS...
...IN PHP...

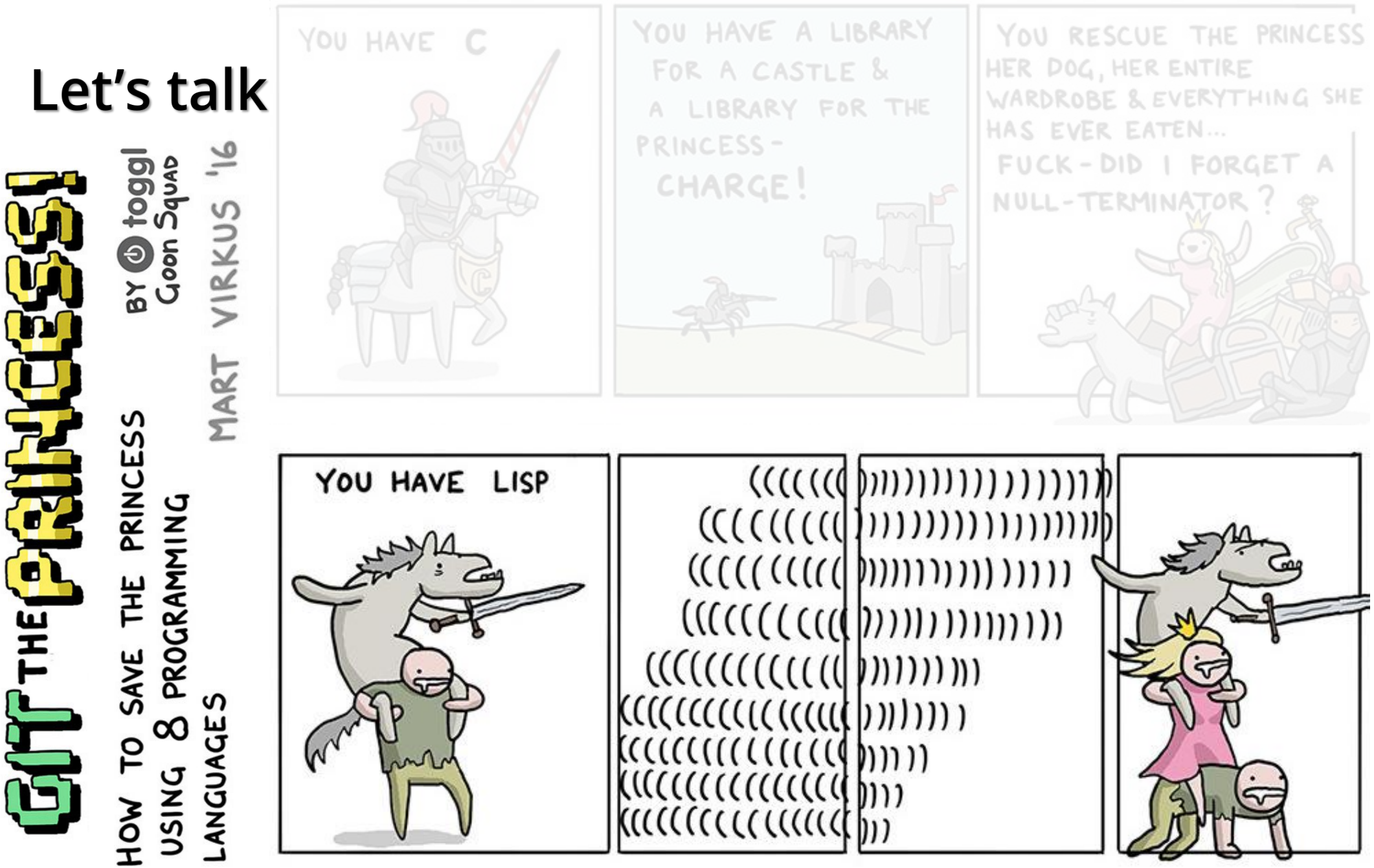
Let's talk


GIT THE PRINCESS!
HOW TO SAVE THE PRINCESS USING 8 PROGRAMMING LANGUAGES
BY toggl Goon Squad
MART VIRKUS '16
YOU HAVE JAVASCRIPT
YOU SPEND HOURS PICKING LIBRARIES, SETTING UP NODE & BUILDING A FRAMEWORK FOR THE CASTLE.
BY THE TIME YOU'RE FINISHED WITH THE FRAMEWORK, THE FORT HAS BEEN ABANDONED AND THE PRINCESS HAS MOVED TO ANOTHER CASTLE
YOU HAVE LISP
YOU HAVE C
YOU HAVE A LIBRARY FOR A CASTLE & A LIBRARY FOR THE PRINCESS - CHARGE!
YOU RESCUE THE PRINCESS HER DOG, HER ENTIRE WARDROBE & EVERYTHING SHE HAS EVER EATEN... FUCK - DID I FORGET A NULL-TERMINATOR?
YOU HAVE GO
WE DON'T SUPPORT FREEING CAPTURED PRINCESSES, WE ALREADY HAVE THESE FREE PRINCESSES IN THE STANDARD LIBR...
...WAIT, IS THIS THE PRINCESS FROM THE JAVA PANEL?
YOU HAVE C#
YOU SPEND HOURS TRYING TO EXPRESS THE ENTIRE RESCUE IN A SINGLE LINQ QUERY
YOU GIVE UP AND GO TO STACKOVERFLOW TO HAVE JON SKEET RESCUE THE PRINCESS FOR YOU.
YOU HAVE PASCAL
YOU DECLARE YOUR PRINCESS, CASTLE & RESCUE PLAN
THEN YOU GO FOR A DRINK & FORGET ABOUT THE IMPLEMENTATION
YOU HAVE JAVA
YOU QUICKLY DEPLOY THE RESCUE TO PRODUCTION
YOU DISCOVER YOU'VE LOADED TWO VERSIONS OF THE CASTLE BUT NO PRINCESS
YOU HAVE PHP
YOU HAVE TO RESCUE THE PRINCESS...
...IN PHP...

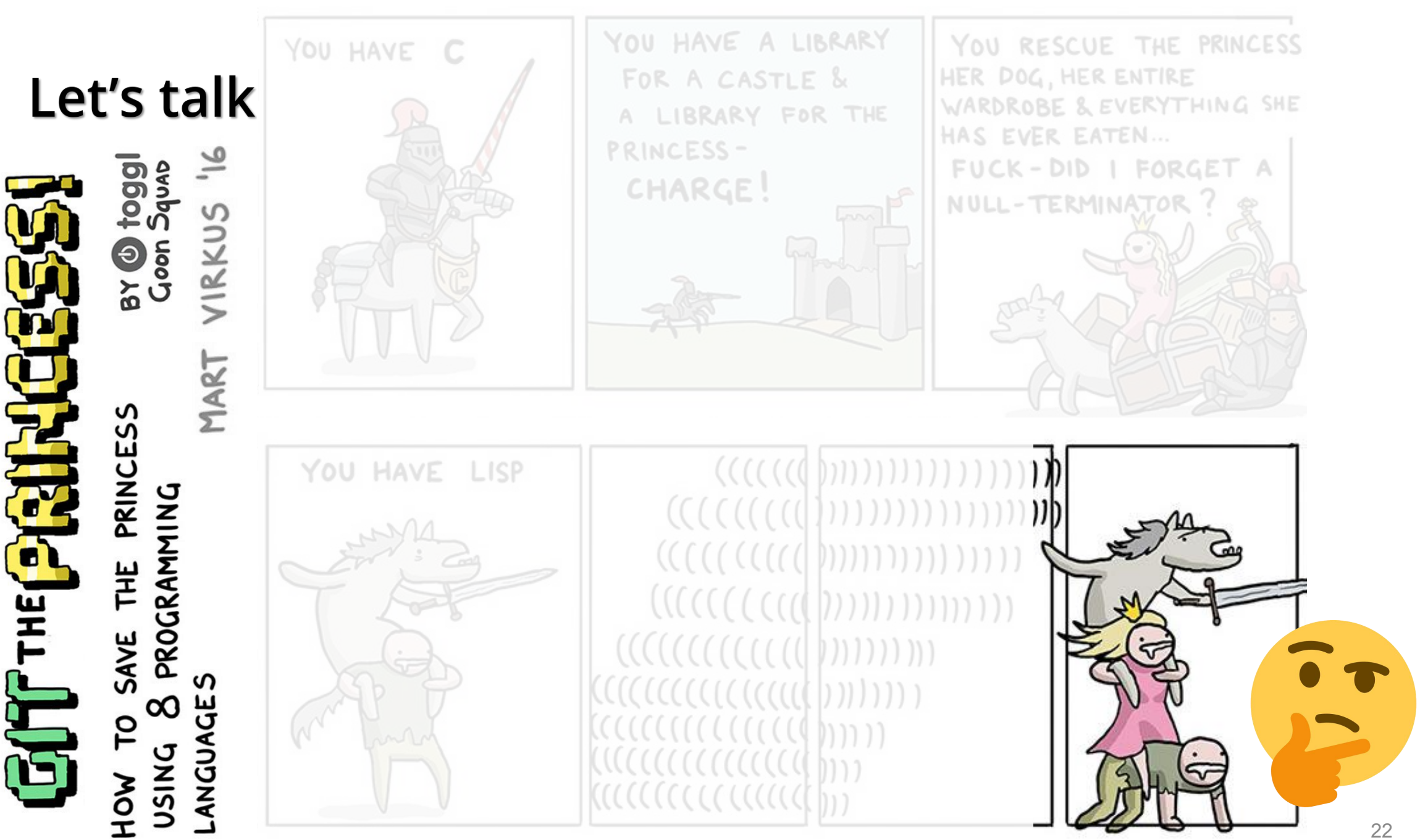
Let's talk


GIT THE PRINCESS!
HOW TO SAVE THE PRINCESS USING 8 PROGRAMMING LANGUAGES
BY toggl Goon Squad
MART VIRKUS '16
YOU HAVE C
YOU HAVE A LIBRARY FOR A CASTLE & A LIBRARY FOR THE PRINCESS - CHARGE!
YOU RESCUE THE PRINCESS HER DOG, HER ENTIRE WARDROBE & EVERYTHING SHE HAS EVER EATEN...
FUCK - DID I FORGET A NULL-TERMINATOR?
YOU HAVE LISP

Let's talk


GIT THE PRINCESS!
HOW TO SAVE THE PRINCESS USING 8 PROGRAMMING LANGUAGES
BY toggl Goon Squad
MART VIRKUS '16
YOU HAVE C
YOU HAVE A LIBRARY FOR A CASTLE & A LIBRARY FOR THE PRINCESS - CHARGE!
YOU RESCUE THE PRINCESS HER DOG, HER ENTIRE WARDROBE & EVERYTHING SHE HAS EVER EATEN... FUCK - DID I FORGET A NULL-TERMINATOR?
YOU HAVE LISP

Let's talk


GIT THE PRINCESS!
HOW TO SAVE THE PRINCESS USING 8 PROGRAMMING LANGUAGES
BY toggl Goon Squad
MART VIRKUS '16
YOU HAVE C
YOU HAVE A LIBRARY FOR A CASTLE & A LIBRARY FOR THE PRINCESS-
CHARGE!
YOU RESCUE THE PRINCESS HER DOG, HER ENTIRE WARDROBE & EVERYTHING SHE HAS EVER EATEN...
FUCK - DID I FORGET A NULL-TERMINATOR?
YOU HAVE LISP

Let’s talk about me

Let’s talk about me

- Software engineer at Ericsson (C/C++/Erlang)

Let’s talk about me

- Software engineer at Ericsson (C/C++/Erlang)
- Common Lisp enthusiast (`#lisp`, Lisp Discord, Reddit)

Let’s talk about me

- Software engineer at Ericsson (C/C++/Erlang)
- Common Lisp enthusiast (`#lisp`, Lisp Discord, Reddit)
  - Author of *The Common Lisp Condition System* (Apress, 2020)

Let’s talk about me

- Software engineer at Ericsson (C/C++/Erlang)
- Common Lisp enthusiast (`#lisp`, Lisp Discord, Reddit)
  - Author of *The Common Lisp Condition System* (Apress, 2020)
- Somewhat capable of writing other programming languages

Let’s talk about me

- Software engineer at Ericsson (C/C++/Erlang)
- Common Lisp enthusiast (`#lisp`, Lisp Discord, Reddit)
  - Author of *The Common Lisp Condition System* (Apress, 2020)
- Somewhat capable of writing other programming languages
- I kinda like the topic of control flow

Let’s talk about me

- I kinda like the topic of control flow

Let’s talk about control flow

- I kinda like the topic of control flow

Let’s talk about control flow in Common Lisp

- I kinda like the topic of control flow

Let’s talk about control flow in Common Lisp

Let’s talk about control flow in Common Lisp

- `if`

```
(if (foo)
    (bar)
    (baz))
```

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`

```
(tagbody
 10 (print “hello”)
 20 (go 10))
```

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`

```
(block my-block
  (...)
  (... (return-from my-block 42))
  (...))
```

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`

```
(catch ‘quux
  (...)
  (... (foo))
  (...)))
```

```
(defun foo ()
  (throw ‘quux 42))
```

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`

```
(let ((thing (make-thing)))
  (unwind-protect (frob thing)
    (cleanup thing)))
```

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

```
(let ((fn (lambda ...))
      (args ...))
  (apply fn 1 2 3 args))
```

Let's talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply ; and funcall`

```
(let ((fn (lambda ...))
      (args ...))
  (apply fn 1 2 3 args))
```

```
(let ((fn (lambda ...)))
  (funcall fn 1 2 3))
```

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about closures

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about closures

Let’s talk about closures

```
(let ((x 42))
  (lambda () x))
```

Let’s talk about closures

```
(let ((x 42))
  (lambda () x))
;; #<FUNCTION (LAMBDA ())>
```

Let’s talk about closures

```
(let ((x 42))
  (lambda () x))
;; #<FUNCTION (LAMBDA ())>

(funcall *) ; (funcall #<FUNCTION (LAMBDA ())>)
```

Let’s talk about closures

```
(let ((x 42))
  (lambda () x))
;; #<FUNCTION (LAMBDA ())>

(funcall *) ; (funcall #<FUNCTION (LAMBDA ())>)
;; => 42
```

Let's talk about closures

```
(let ((x 42))
  (lambda () x))
;; #<FUNCTION (LAMBDA ())>

(funcall *) ; (funcall #<FUNCTION (LAMBDA ())>)
;; => 42

;; but we can close over more
;; than just lexical variables!
```

Let's talk about non-local control flow in Common Lisp

```
(let ((x 42))
  (lambda () x))
;; #<FUNCTION (LAMBDA ())>

(funcall *) ; (funcall #<FUNCTION (LAMBDA ())>)
;; => 42

;; but we can close over more
;; than just lexical variables!
```

Let’s talk about non-local control flow in Common Lisp

Let’s talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))
```

Let’s talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar () ; block bar
  ...)
```

Let's talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    ...))
```

Let's talk about non-local control flow in Common Lisp

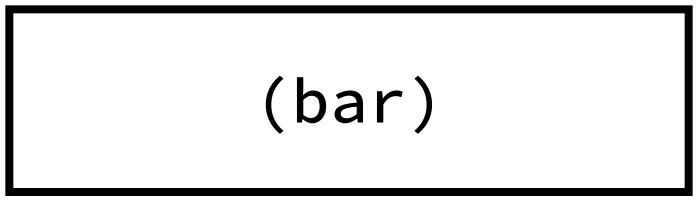
```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))
```

Let's talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
```

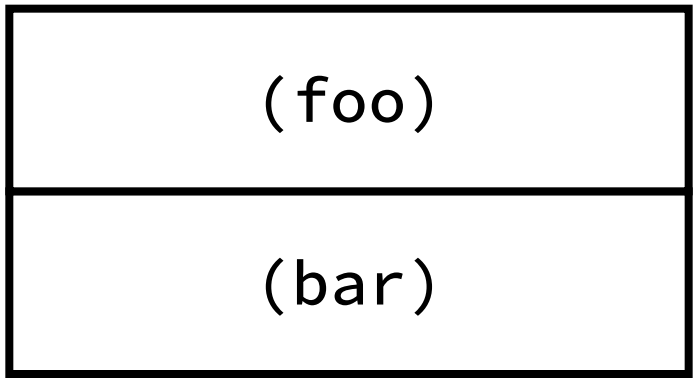
Let's talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
```



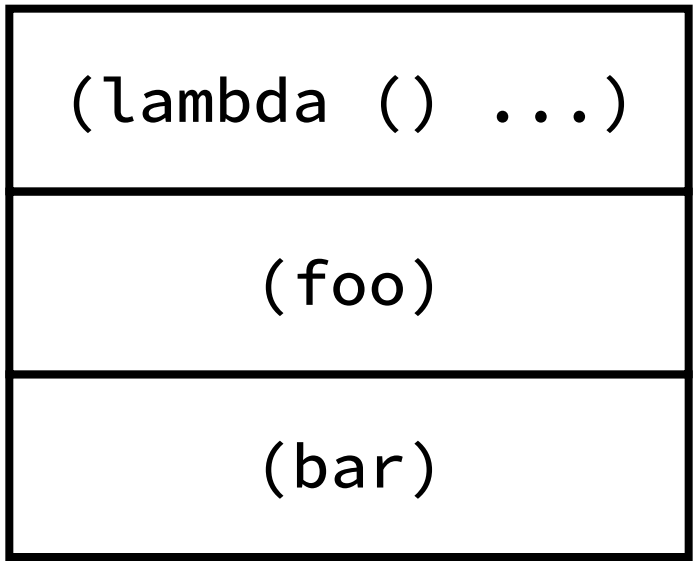
Let's talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
```

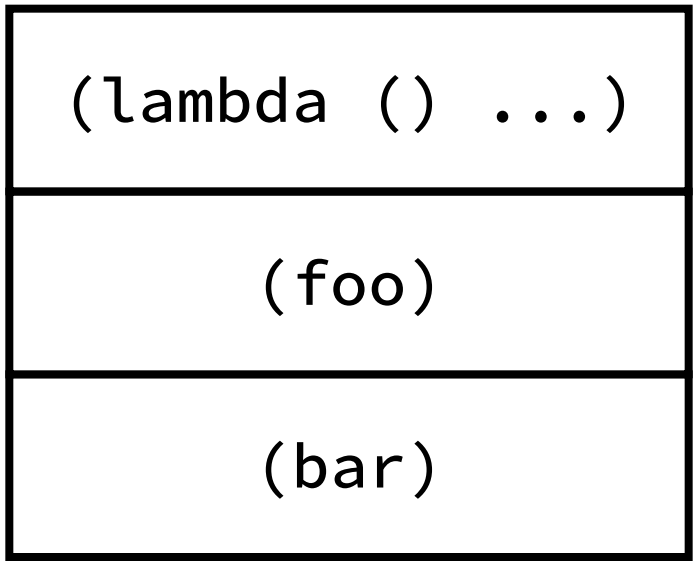


Let's talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
```

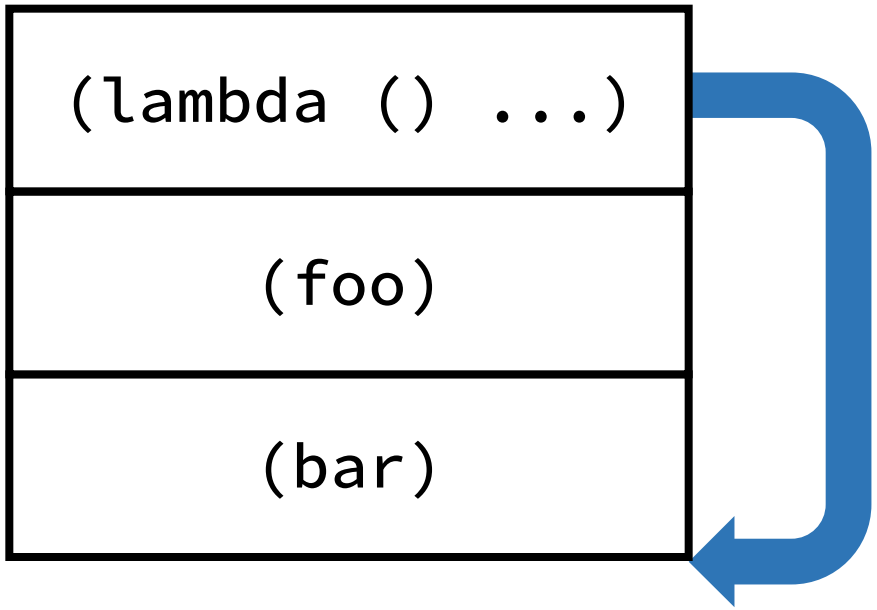


Let's talk about non-local control flow in Common Lisp




```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
```

Let's talk about non-local control flow in Common Lisp




```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
```

Let's talk about non-local control flow in Common Lisp

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
;; => 42
```

Let's talk about non-local control flow in Common Lisp

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply

```
(defun foo (x) (funcall x))

(defun bar ()
  (let ((fn (lambda ()
              (return-from
               bar 42))))
    (foo fn)))

(bar)
;; => 42
```

Let’s talk about non-local control flow in Common Lisp

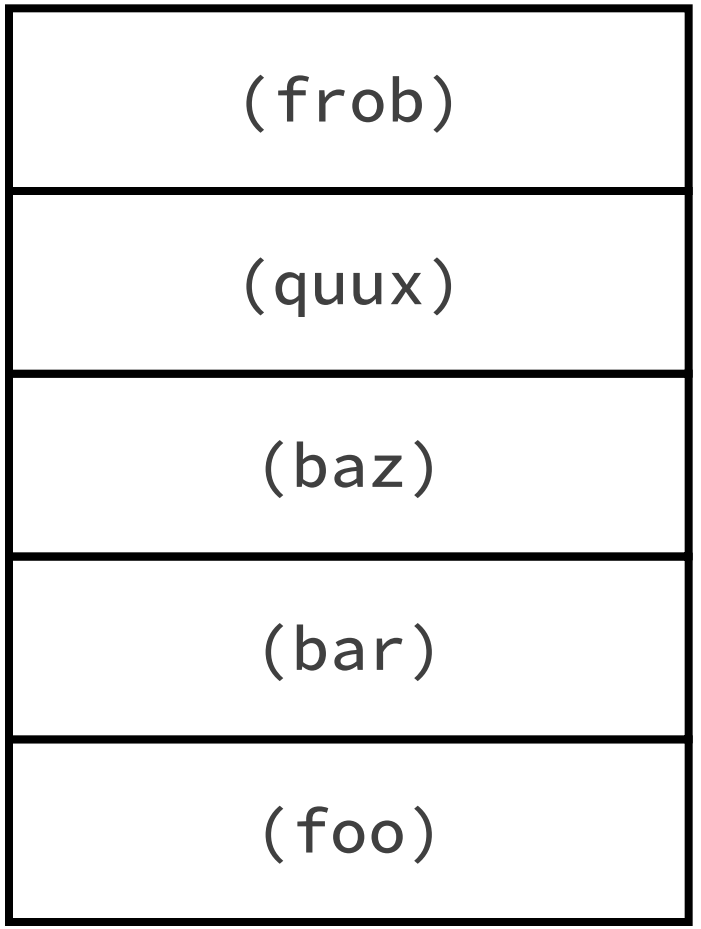
- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

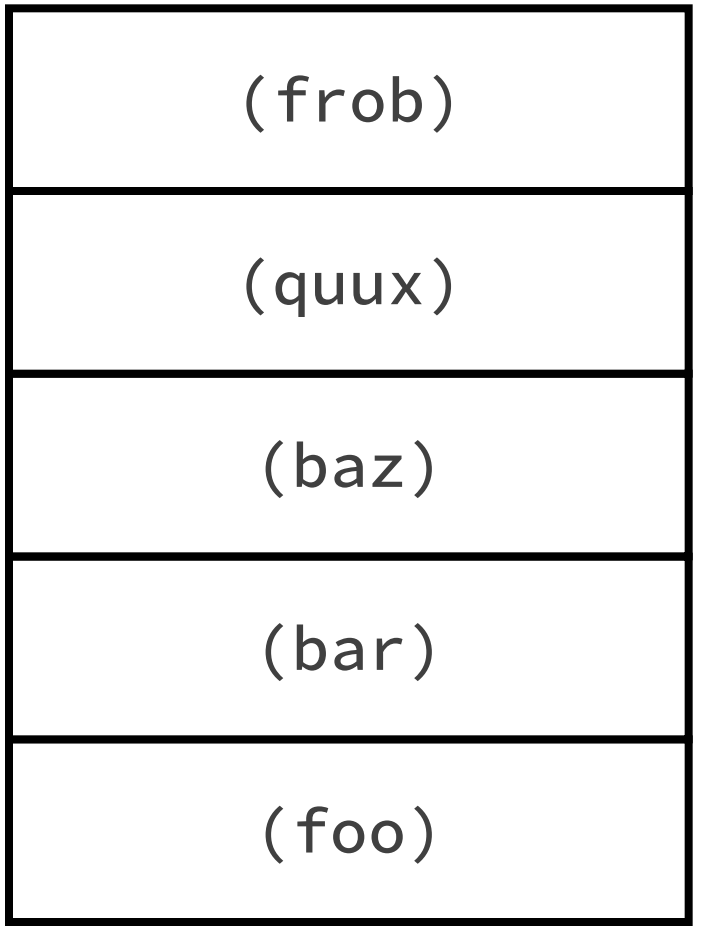
Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



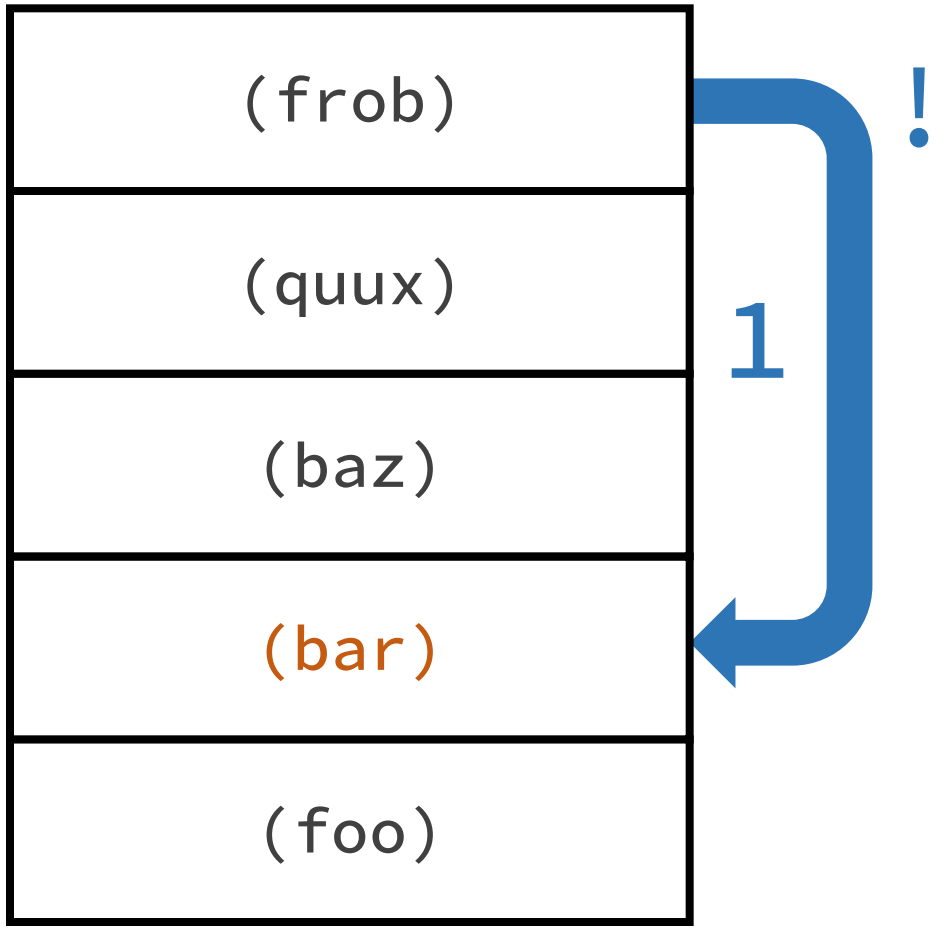
Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



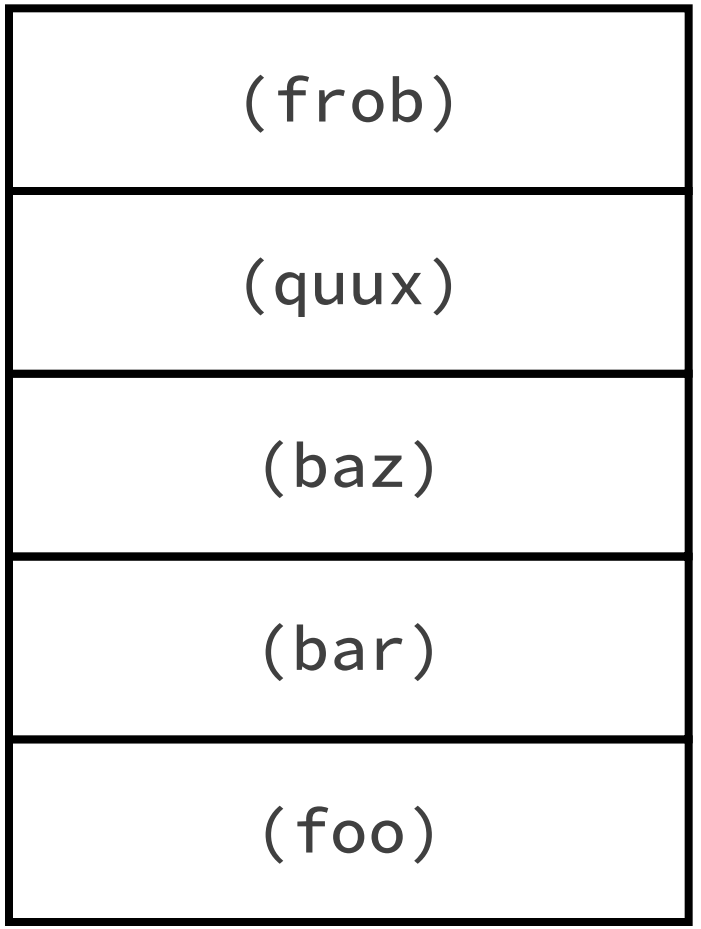
Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



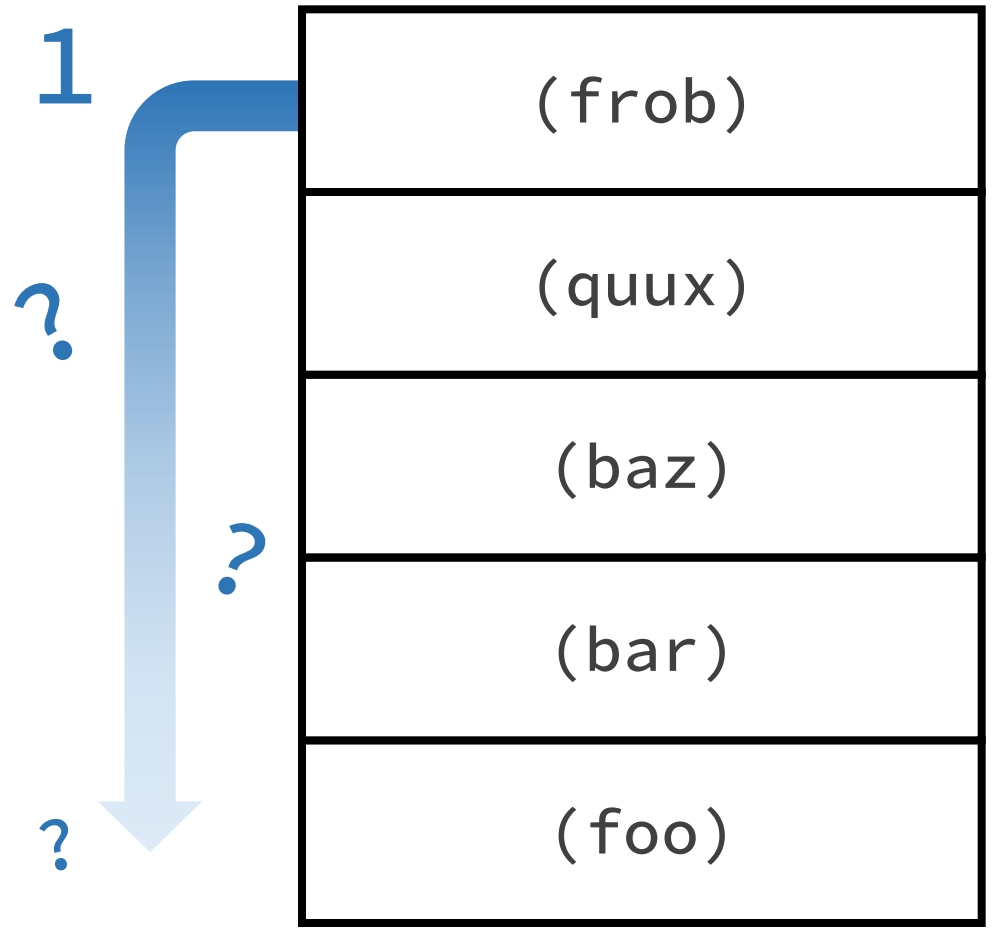
Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



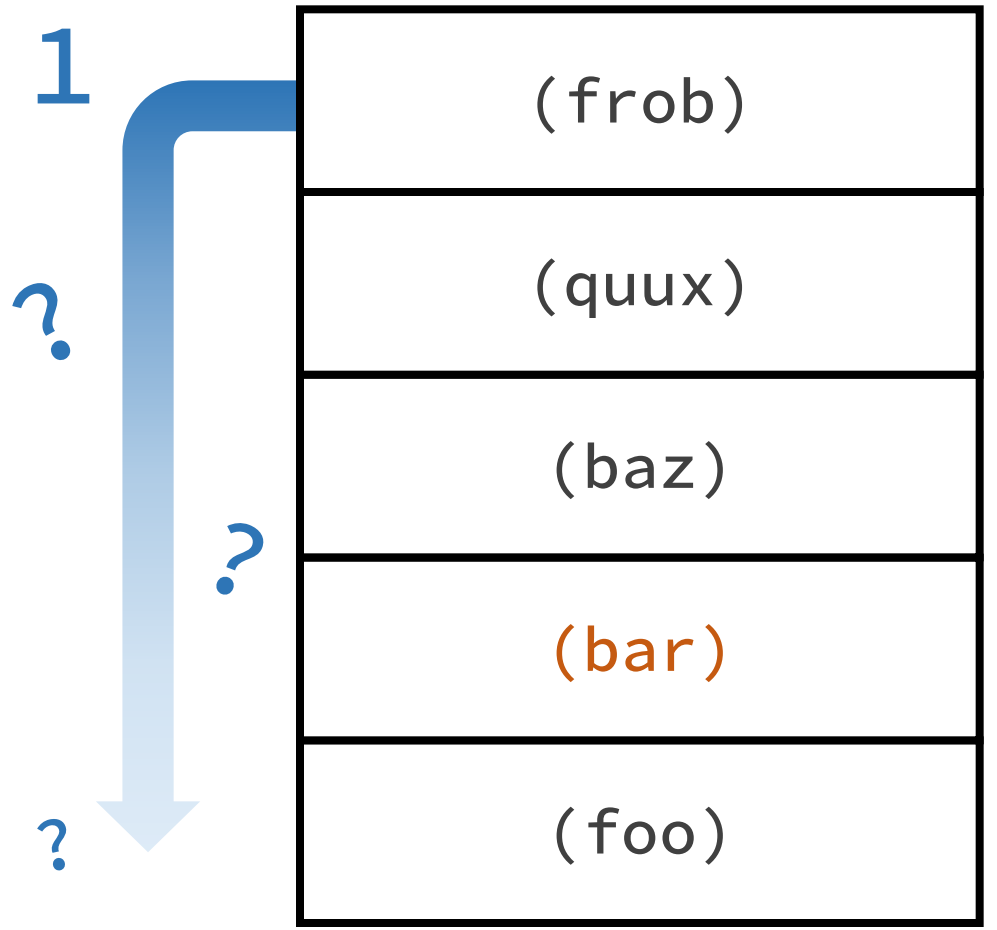
Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



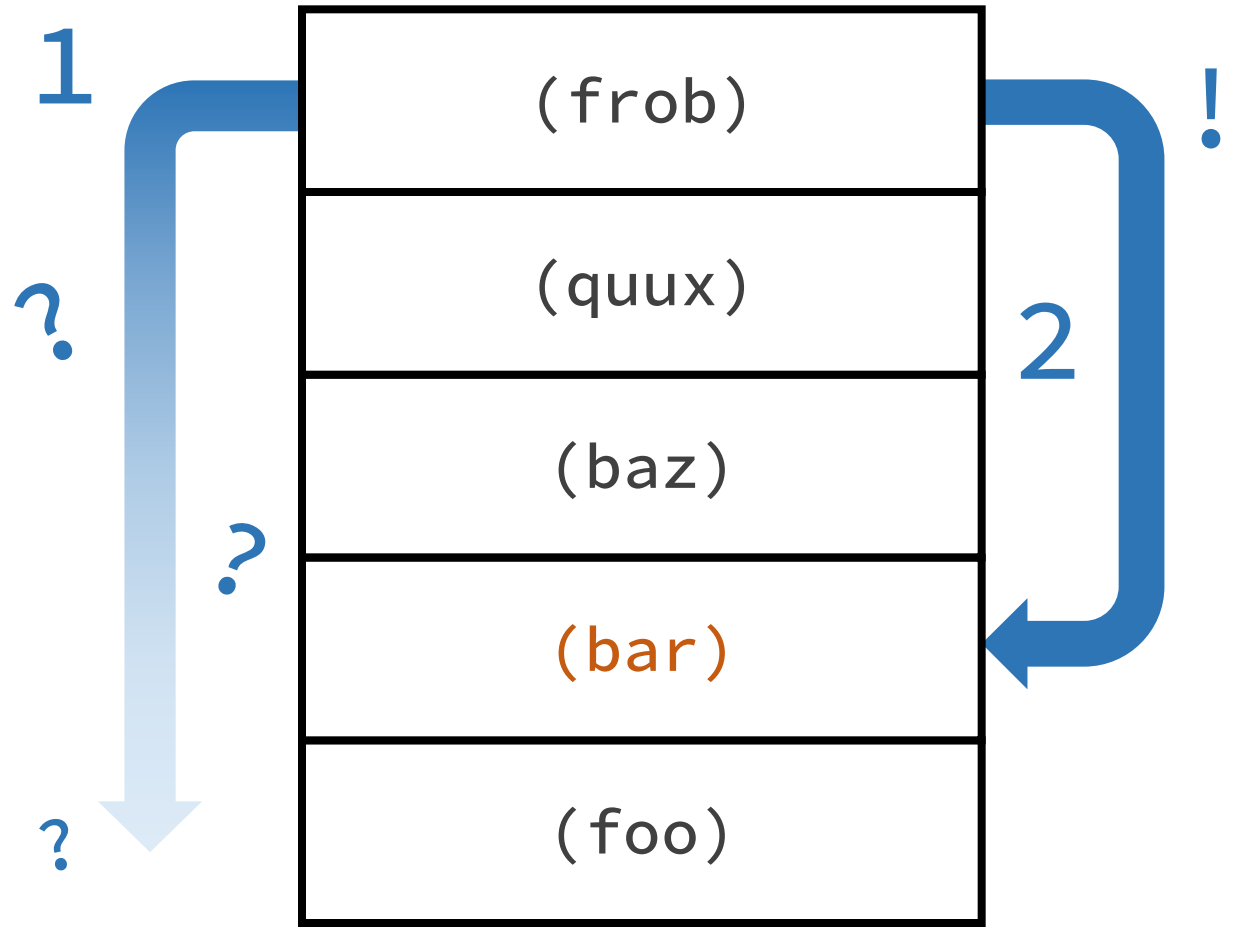
Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go` ; 1-phase unwind (no search)
- `block/return-from` ; 1-phase unwind (no search)
- `catch/throw` ; 2-phase unwind (search)
- `unwind-protect`
- `lambda/apply`

Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives

Let's talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, ...)

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, …)
  - switches (`cond`, `case`, `typecase`, …)

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, …)
  - switches (`cond`, `case`, `typecase`, …)
  - error handling (`handler-case`, …)

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, …)
  - switches (`cond`, `case`, `typecase`, …)
  - error handling (`handler-case`, …)
  - restarts (`restart-case`, …)

Let's talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, ...)
  - switches (`cond`, `case`, `typecase`, ...)
  - error handling (`handler-case`, ...)
  - restarts (`restart-case`, ...)
- This list includes use cases that are *not related* to exception handling

Let's talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, ...)
  - switches (`cond`, `case`, `typecase`, ...)
  - error handling (`handler-case`, ...)
  - restarts (`restart-case`, ...)
- This list includes use cases that are *not related* to exception handling

- **Control flow ≠ exception handling**

Let's talk about control flow in general

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- All other CL control flow operators are derivatives of those primitives
  - loops (`do`, `dolist`, `loop`, …)
  - switches (`cond`, `case`, `typecase`, …)
  - error handling (`handler-case`, …)
  - restarts (`restart-case`, …)
- This list includes use cases that are *not related* to exception handling

- **Control flow ≠ exception handling**

Let’s talk about control flow in general

- Control flow ≠ exception handling

Let’s talk about control flow in general

- Do not conflate unwinding with throwing exceptions

- Control flow ≠ exception handling

Let’s talk about control flow in general

- **Do not conflate unwinding with throwing exceptions**
  - Throwing exceptions is *a subset* of control flow

- **Control flow ≠ exception handling**

Let’s talk about control flow in general

- **Do not conflate unwinding with throwing exceptions**
  - Throwing exceptions is *a subset* of control flow
  - Throwing exceptions is *not* synonymous with unwinding

- **Control flow ≠ exception handling**

Let’s talk about control flow in general

- **Do not conflate unwinding with throwing exceptions**
  - Throwing exceptions is *a subset* of control flow
  - Throwing exceptions is *not* synonymous with unwinding
  - Throwing exceptions is *not* a primitive operation

- **Control flow ≠ exception handling**

Let’s talk about control flow in general

- **Do not conflate unwinding with throwing exceptions**
  - Throwing exceptions is *a subset* of control flow
  - Throwing exceptions is *not* synonymous with unwinding
  - Throwing exceptions is *not* a primitive operation

  - Proofs: Common Lisp, Dylan, Smalltalk

- **Control flow ≠ exception handling**

Let’s talk about control flow in WebAssembly

- **Do not conflate unwinding with throwing exceptions**
  - Throwing exceptions is *a subset* of control flow
  - Throwing exceptions is *not* synonymous with unwinding
  - Throwing exceptions is *not* a primitive operation

  - Proofs: Common Lisp, Dylan, Smalltalk

- **Control flow ≠ exception handling**

Let’s talk about control flow in WebAssembly

- Control flow ≠ exception handling

Let’s talk about control flow in WebAssembly

- Suggestion: unwinding as a WebAssembly primitive
  - Name the WASM operator `unwind` instead of `throw` perhaps?

- **Control flow ≠ exception handling**

Let's talk about control flow in WebAssembly

- Suggestion: unwinding as a WebAssembly primitive
  - Name the WASM operator `unwind` instead of `throw` perhaps?
- Suggestion: WebAssembly terminology change
  - 1PEH → 1PU = one-phase unwinding (`unwind`)
  - 2PEH → 2PU = two-phase unwinding (`stack-search` + `unwind`)

- **Control flow ≠ exception handling**

Let's talk about control flow in WebAssembly

- Suggestion: unwinding as a WebAssembly primitive
  - Name the WASM operator `unwind` instead of `throw` perhaps?
- Suggestion: WebAssembly terminology change
  - 1PEH → 1PU = one-phase unwinding (`unwind`)
  - 2PEH → 2PU = two-phase unwinding (`stack-search` + `unwind`)
- Suggestion: support for dynamic (fluid) variables?
  - Basis for implementing a condition system
  - Otherwise, we will need some other stack-searching operator
  - …or we'll need to reimplement dynamic variables

- **Control flow ≠ exception handling**

Let’s talk about control flow in WebAssembly

Let’s talk about control flow in WebAssembly

- https://github.com/phoe/portable-condition-system
  Common Lisp condition system implemented in Common Lisp

Let’s talk about control flow in WebAssembly

- https://github.com/phoe/portable-condition-system
  Common Lisp condition system implemented in Common Lisp
- https://github.com/phoe/cafe-latte/
  Common Lisp control flow operators and condition system implemented in Java

Let’s talk about control flow in WebAssembly

- https://github.com/phoe/portable-condition-system
  Common Lisp condition system implemented in Common Lisp
- https://github.com/phoe/cafe-latte/
  Common Lisp control flow operators and condition system implemented in Java
- https://www.youtube.com/watch?v=V4P9lFK79hQ
  Control Flow in Common Lisp - Online Lisp Meeting #11,
  a recording of material presented in this talk,
  including the differences between conditions and exceptions

<3

<3

but wait hold on for just one moment

Control Flow in Common Lisp

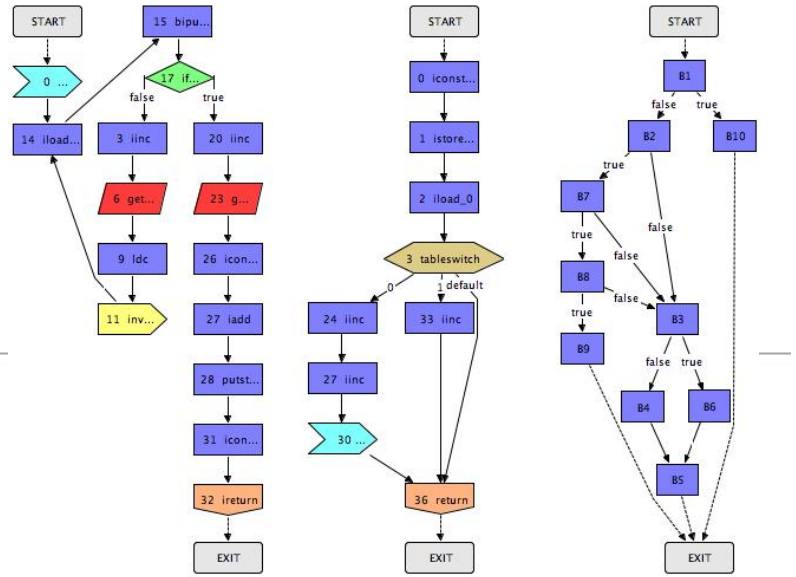
aka Why Lisp Doesn't Need To Throw Exceptions

Appendix A

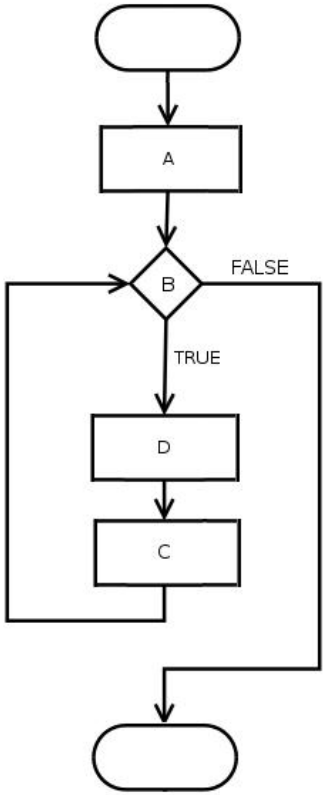
Differences between conditions and exceptions

Let’s talk about control flow

Let’s talk about control flow



for(A;B;C)
D;



Control flow

From Wikipedia, the free encyclopedia

Not to be confused with Flow control (data).

In computer science, **control flow** (or **flow of control**) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

Let’s talk about non-local control flow

Structured non-local control flow [edit]

Many programming languages, especially those favoring more dynamic styles of programming, offer constructs for *non-local control flow*. These cause the flow of execution to jump out of a given context and resume at some predeclared point. *Conditions*, *exceptions* and *continuations* are three common sorts of non-local control constructs; more exotic ones also exist, such as generators, coroutines and the async keyword.

Let’s talk about non-local control flow

Structured non-local control flow [edit]

Many programming languages, especially those favoring more dynamic styles of programming, offer constructs for *non-local control flow*. These cause the flow of execution to jump out of a given context and resume at some predeclared point. *Conditions*, *exceptions* and *continuations* are three common sorts of non-local control constructs; more exotic ones also exist, such as generators, coroutines and the async keyword.

Let’s talk about exceptions

Exception support in programming languages [edit]

See also: Exception handling syntax

Many computer languages have built-in support for exceptions and exception handling. This includes ActionScript, Ada, BlitzMax, C++, C#, Clojure, COBOL, D, ECMAScript, Eiffel, Java, ML, Next Generation Shell, Object Pascal (e.g. Delphi, Free Pascal, and the like), PowerBuilder, Objective-C, OCaml, PHP (as of version 5), PL/I, PL/SQL, Prolog, Python, REALbasic, Ruby, Scala, Seed7, Smalltalk, Tcl, Visual Prolog and most .NET languages. Exception handling is commonly not resumable in those languages, and when an exception is thrown, the program searches back through the stack of function calls until an exception handler is found.

Some languages call for unwinding the stack as this search progresses. That is, if function f, containing a handler H for exception E, calls function g, which in turn calls function h, and an exception E occurs in h, then functions h and g may be terminated, and H in f will handle E.

Let’s talk about exceptions

Exception support in programming languages [edit]

See also: Exception handling syntax

Many computer languages have built-in support for exceptions and exception handling. This includes ActionScript, Ada, BlitzMax, C++, C#, Clojure, COBOL, D, ECMAScript, Eiffel, Java, ML, Next Generation Shell, Object Pascal (e.g. Delphi, Free Pascal, and the like), PowerBuilder, Objective-C, OCaml, PHP (as of version 5), PL/I, PL/SQL, Prolog, Python, REALbasic, Ruby, Scala, Seed7, Smalltalk, Tcl, Visual Prolog and most .NET languages. Exception handling is commonly not resumable in those languages, and when an exception is thrown, the program searches back through the stack of function calls until an exception handler is found.

Some languages call for unwinding the stack as this search progresses. That is, if function f, containing a handler H for exception E, calls function g, which in turn calls function h, and an exception E occurs in h, then functions h and g may be terminated, and H in f will handle E.

Let’s talk about exceptions

Let’s talk about exceptions

Let’s talk about exceptions

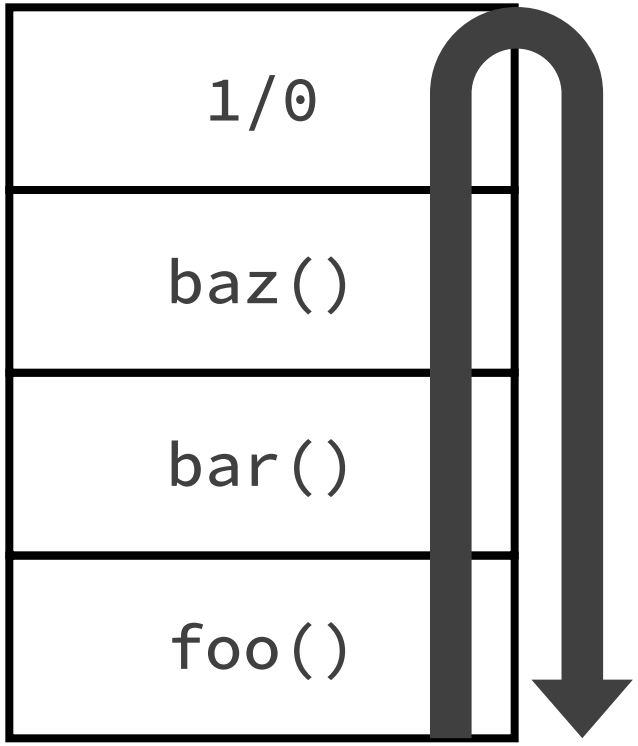
Let’s talk about exceptions

`ArithmeticException`

```
1/0
baz()
bar()
foo()
```

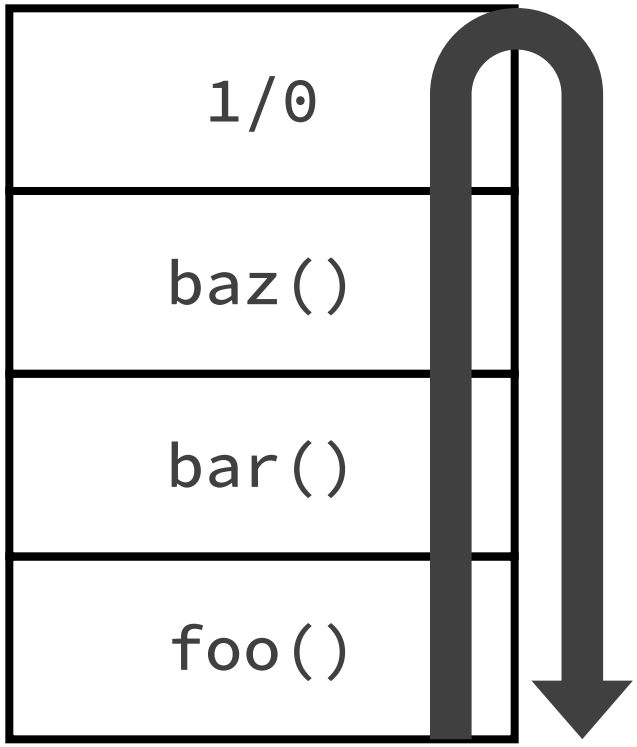
Let’s talk about exceptions

ArithmeticException



Let’s talk about exceptions

`ArithmeticException`



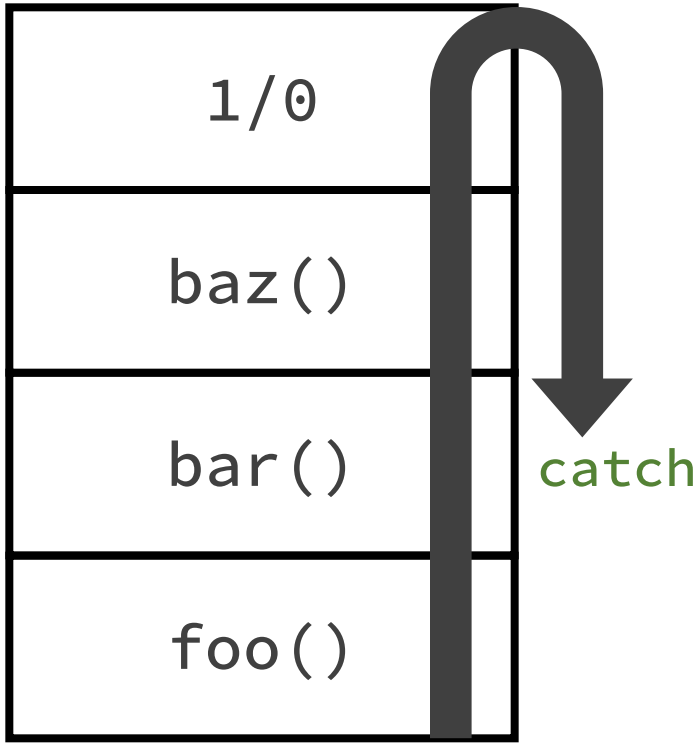
Let’s talk about exceptions

`ArithmeticException`

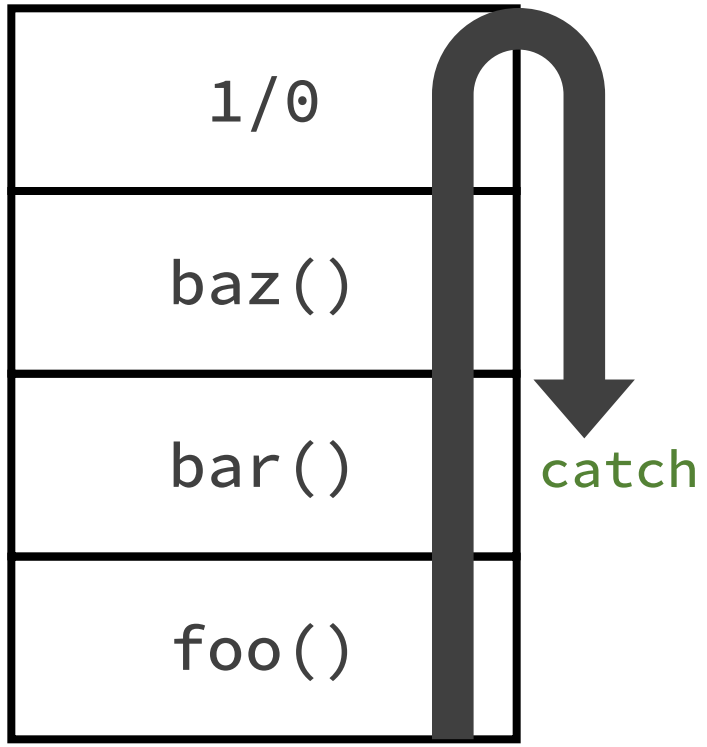
```
1/0
baz()
bar()
foo()
```

Let’s talk about exceptions

ArithmeticException

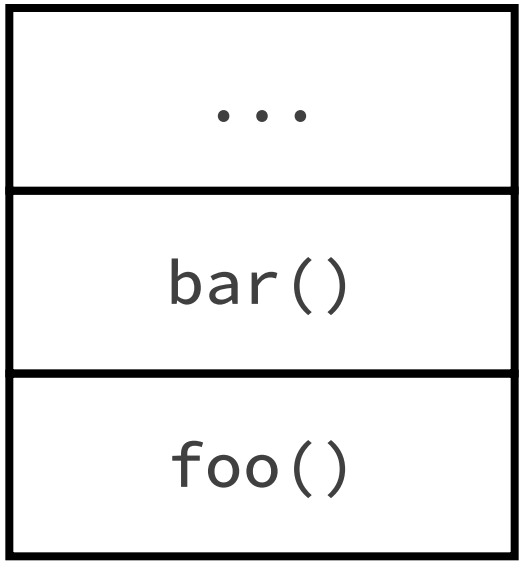


Let’s talk about exceptions



Let’s talk about exceptions

Let’s talk about exceptions




```
// execution continues
```

Let’s talk about conditions

Condition systems [edit]

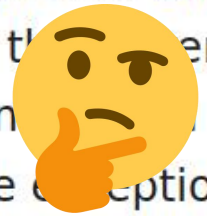
Common Lisp, Dylan and Smalltalk have a condition system[53] (see Common Lisp Condition System) that encompasses the aforementioned exception handling systems. In those languages or environments the advent of a condition (a "generalisation of an error" according to Kent Pitman) implies a function call, and only late in the exception handler the decision to unwind the stack may be taken.

Conditions are a generalization of exceptions. When a condition arises, an appropriate condition handler is searched for and selected, in stack order, to handle the condition. Conditions that do not represent errors may safely go unhandled entirely; their only purpose may be to propagate hints or warnings toward the user.[54]

Let’s talk about conditions

Condition systems [edit]

Common Lisp, Dylan and Smalltalk have a condition system[53] (see Common Lisp Condition System) that encompasses t ementioned exception handling systems. In those languages or environments the advent of a con "generalisation of an error" according to Kent Pitman) implies a function call, and only late in the e ception handler the decision to unwind the stack may be taken.



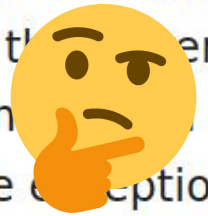
Conditions are a generalization of exceptions. When a condition arises, an appropriate condition handler is searched for and selected, in stack order, to handle the condition. Conditions that do not represent errors may safely go unhandled entirely; their only purpose may be to propagate hints or warnings toward the user.[54]

Let's talk about conditions

Condition systems [edit]

Common Lisp, Dylan and Smalltalk have a condition system[53] (see Common Lisp Condition System) that encompasses t ementioned exception handling systems. In those languages or environments the advent of a con "generalisation of an error" according to Kent Pitman) implies a function call, and only late in the e ception handler the decision to unwind the stack may be taken.

Conditions are a generalization of exceptions. When a condition arises, an appropriate condition handler is searched for and selected, in stack order, to handle the condition. Conditions that do not represent errors may safely go unhandled entirely; their only purpose may be to propagate hints or warnings toward the user.[54]



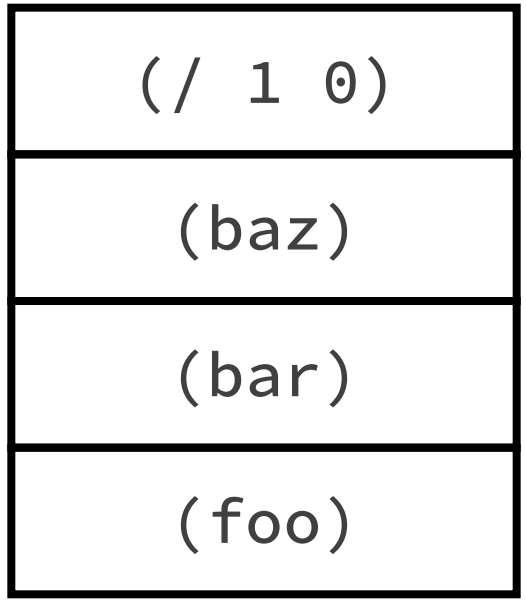
Let’s talk about signaling an error

Let’s talk about signaling an error

Let’s talk about signaling an error

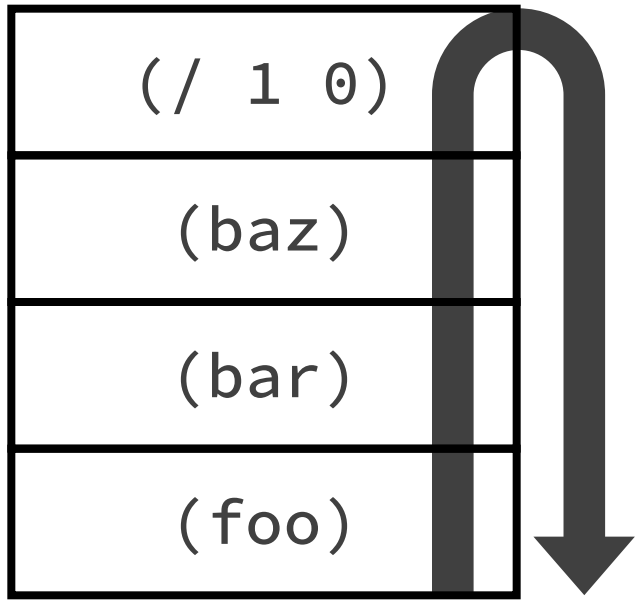
Let’s talk about signaling an error

division-by-zero



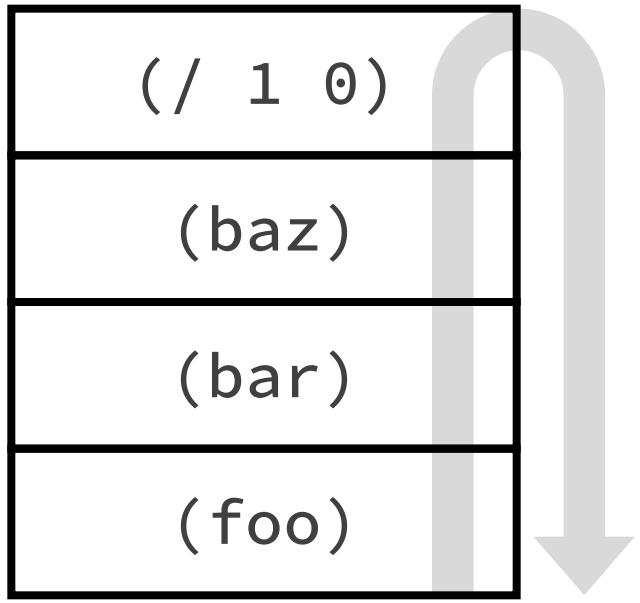
Let’s talk about signaling an error

division-by-zero



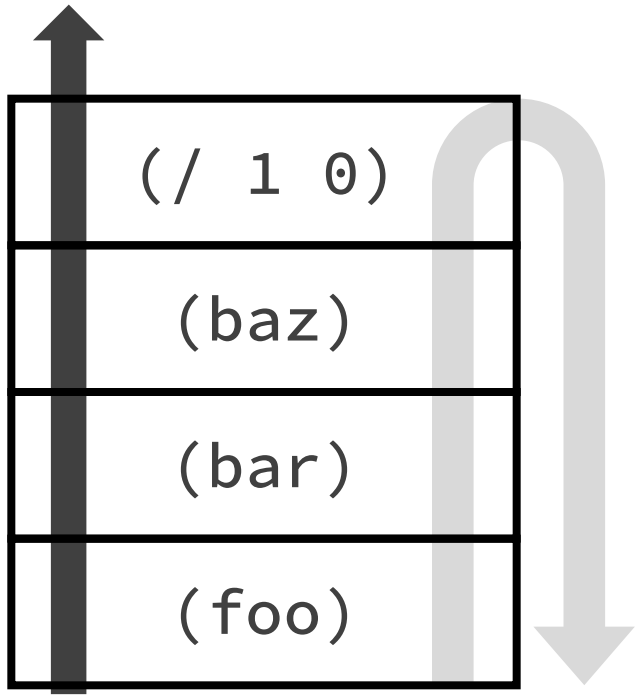
Let’s talk about signaling an error

division-by-zero

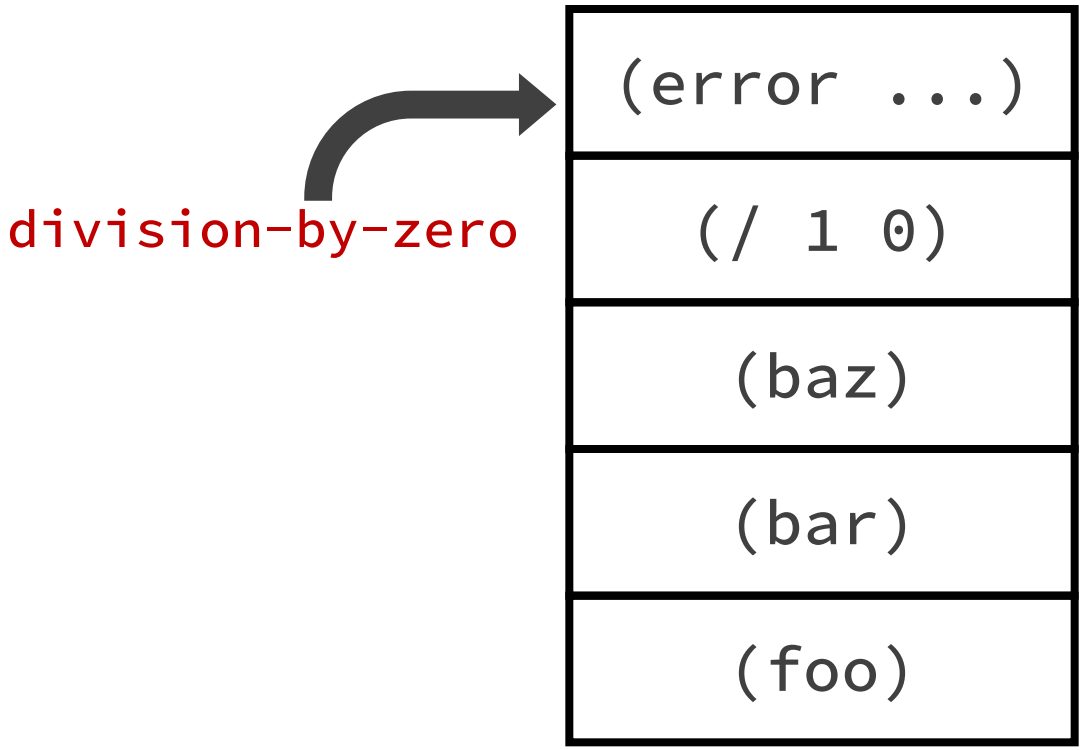


Let’s talk about signaling an error

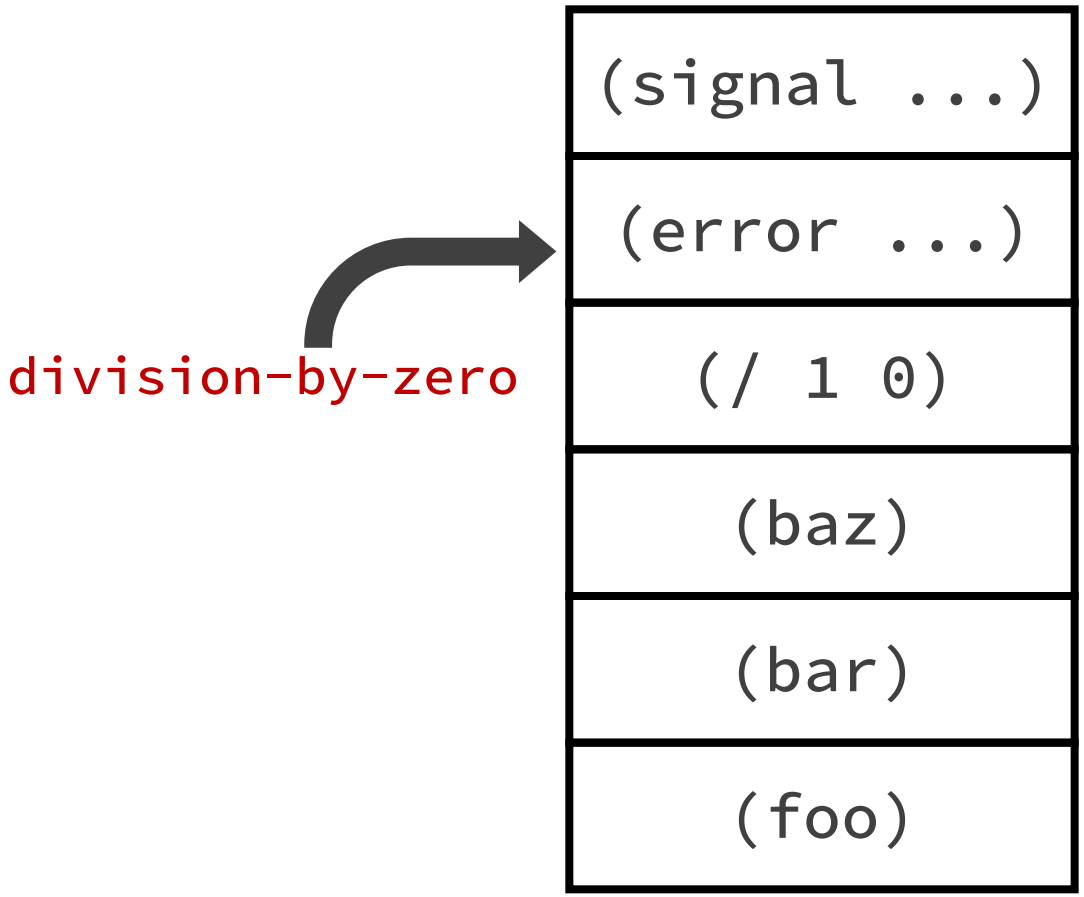
division-by-zero



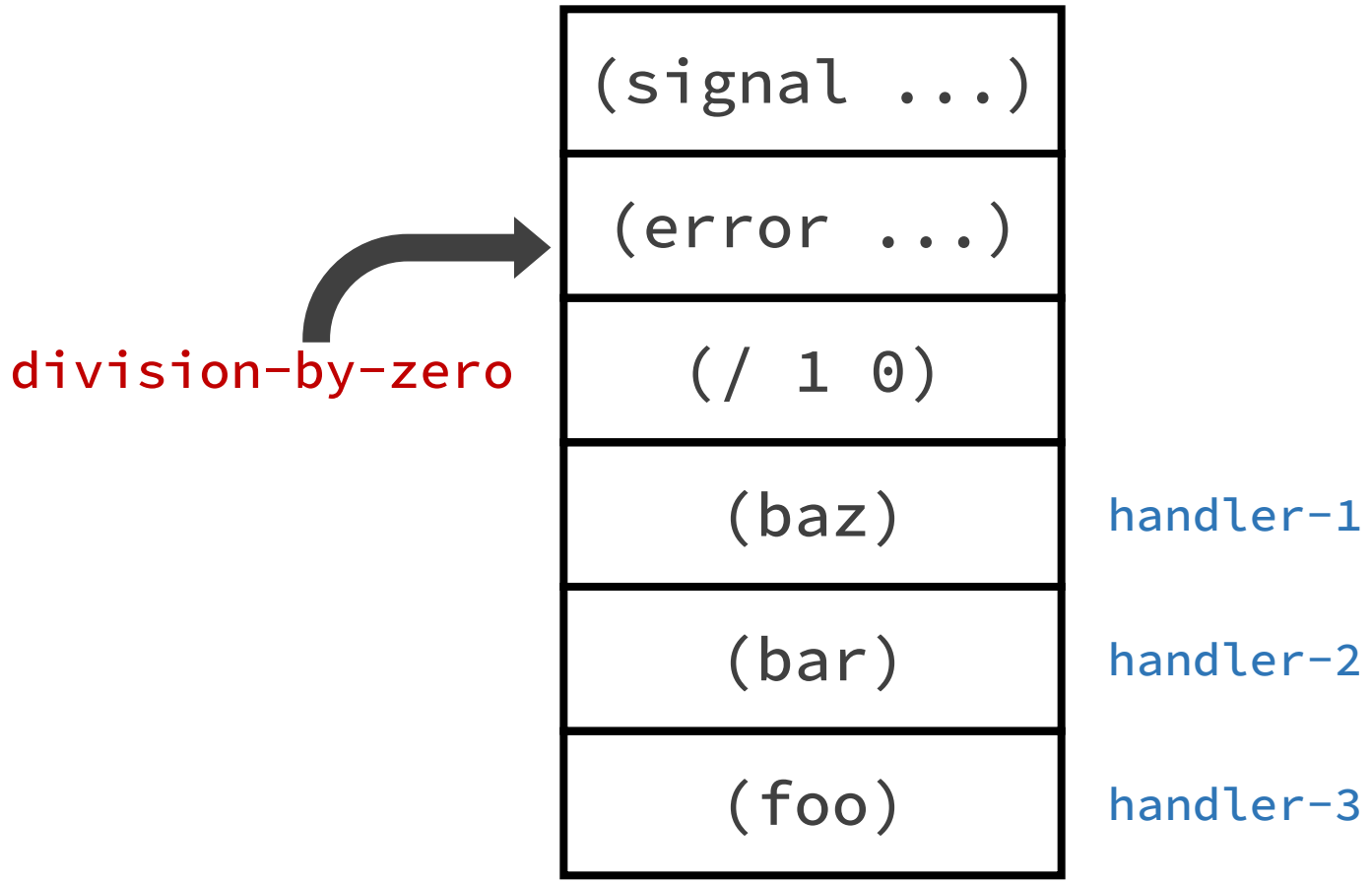
Let’s talk about signaling an error



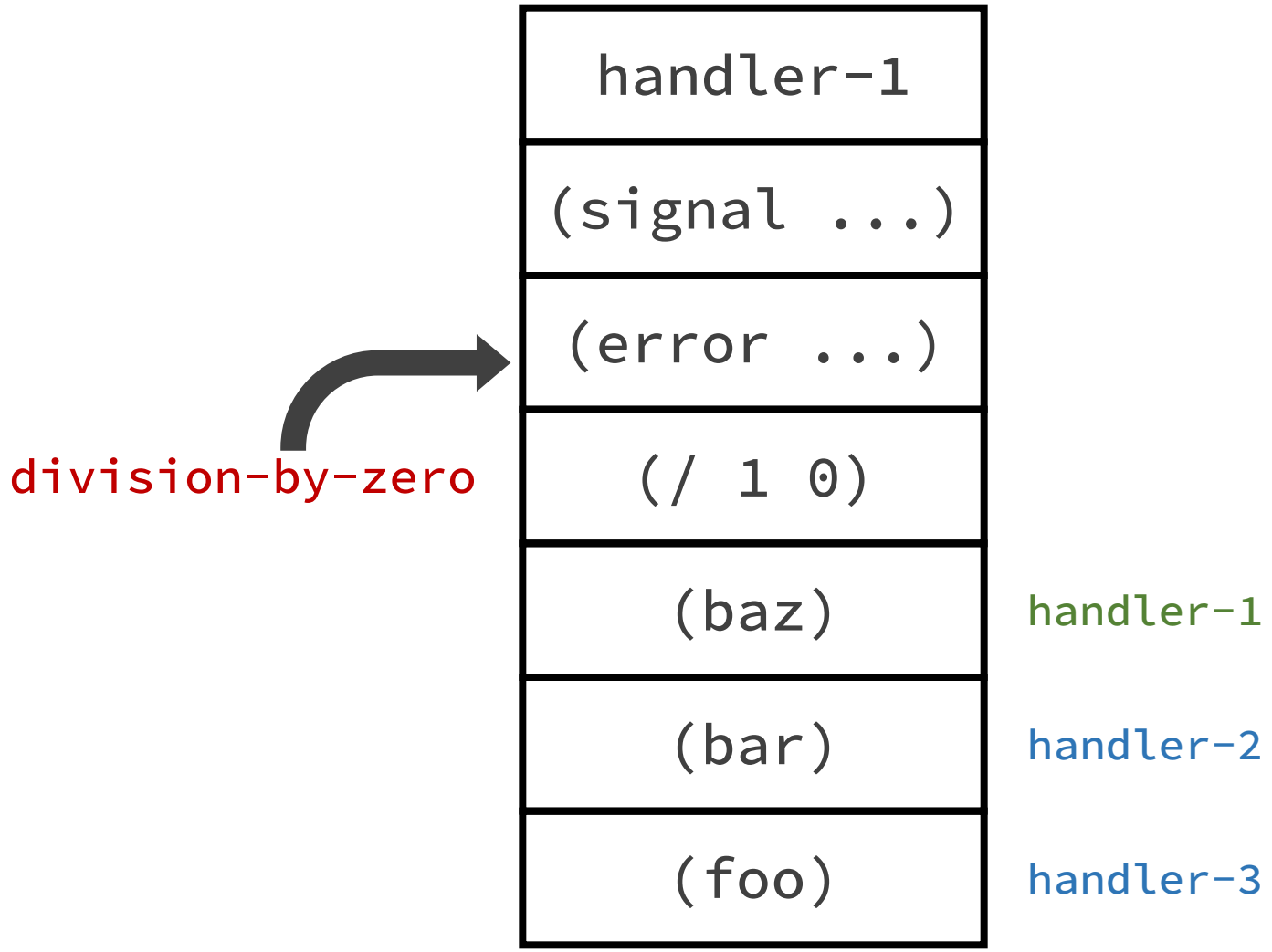
Let’s talk about signaling an error



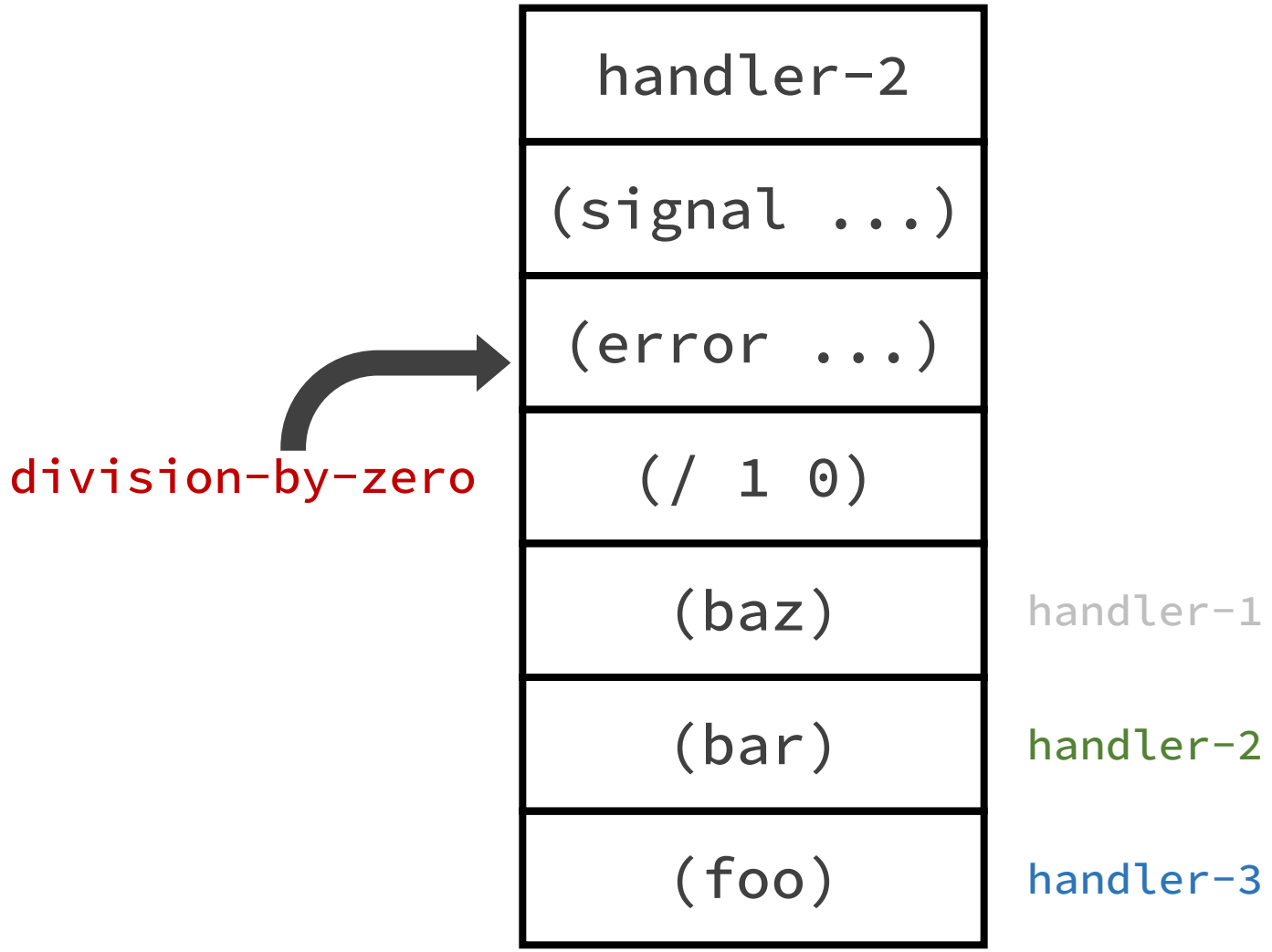
Let’s talk about signaling an error



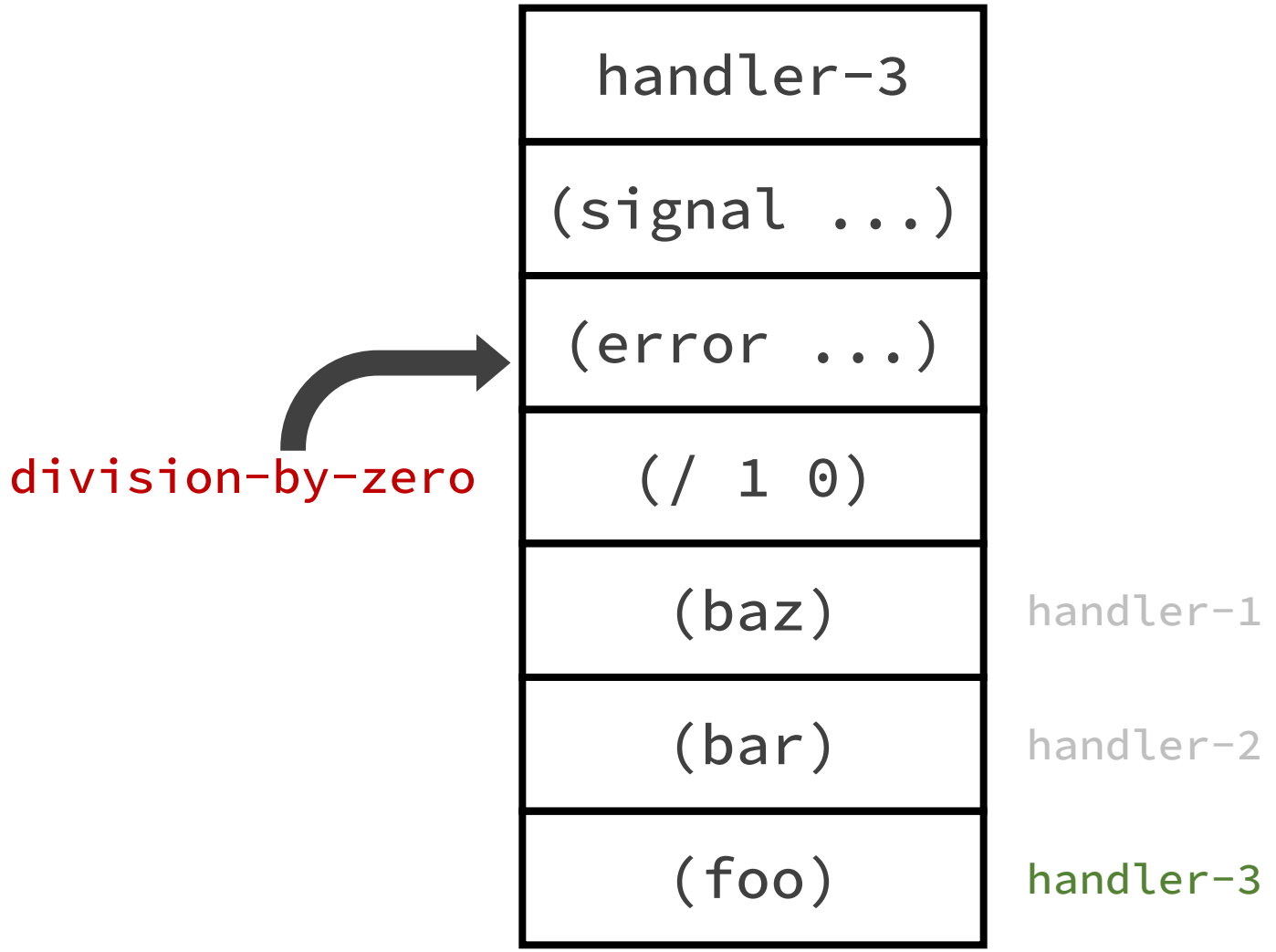
Let’s talk about signaling an error



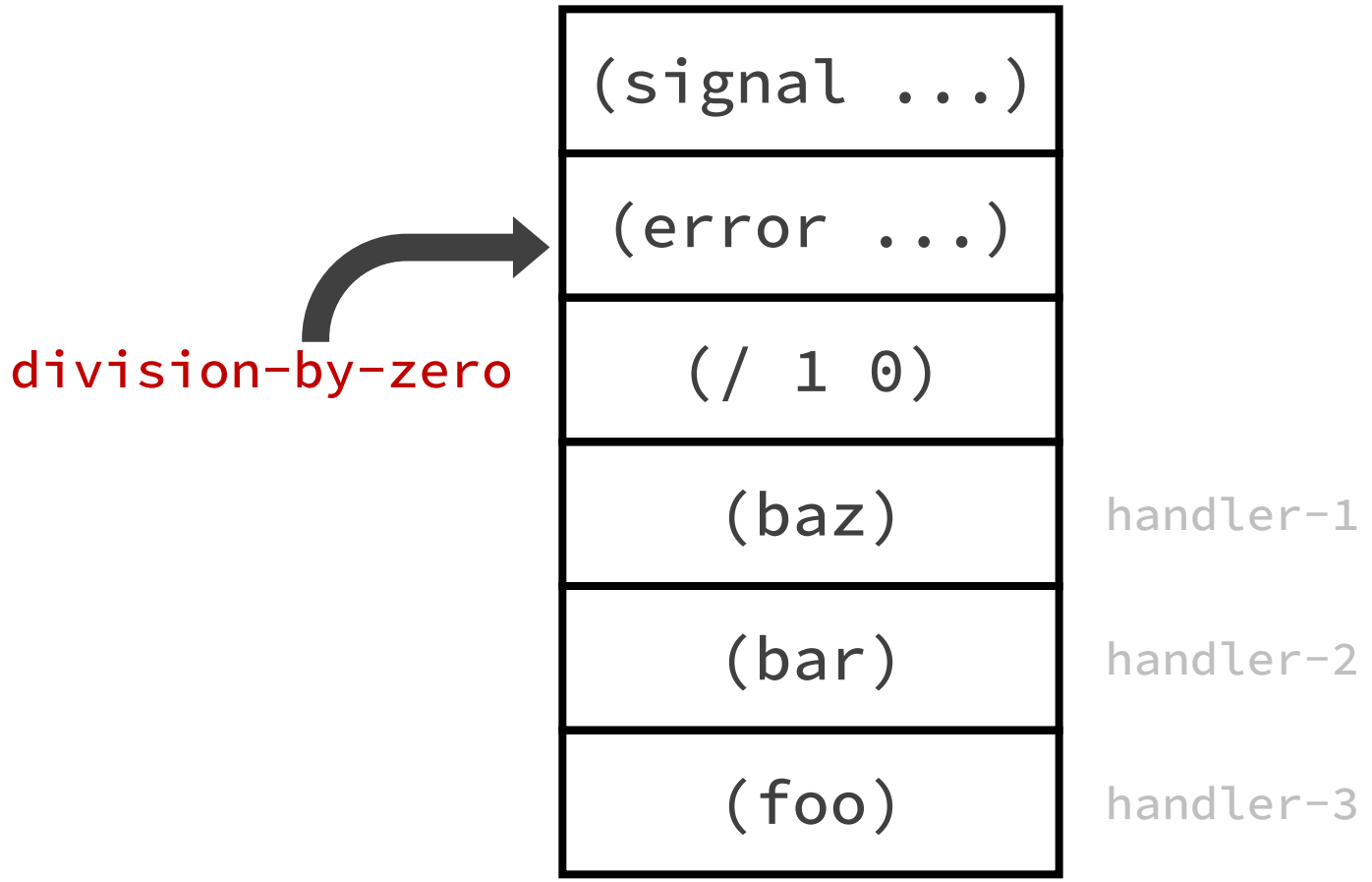
Let’s talk about signaling an error



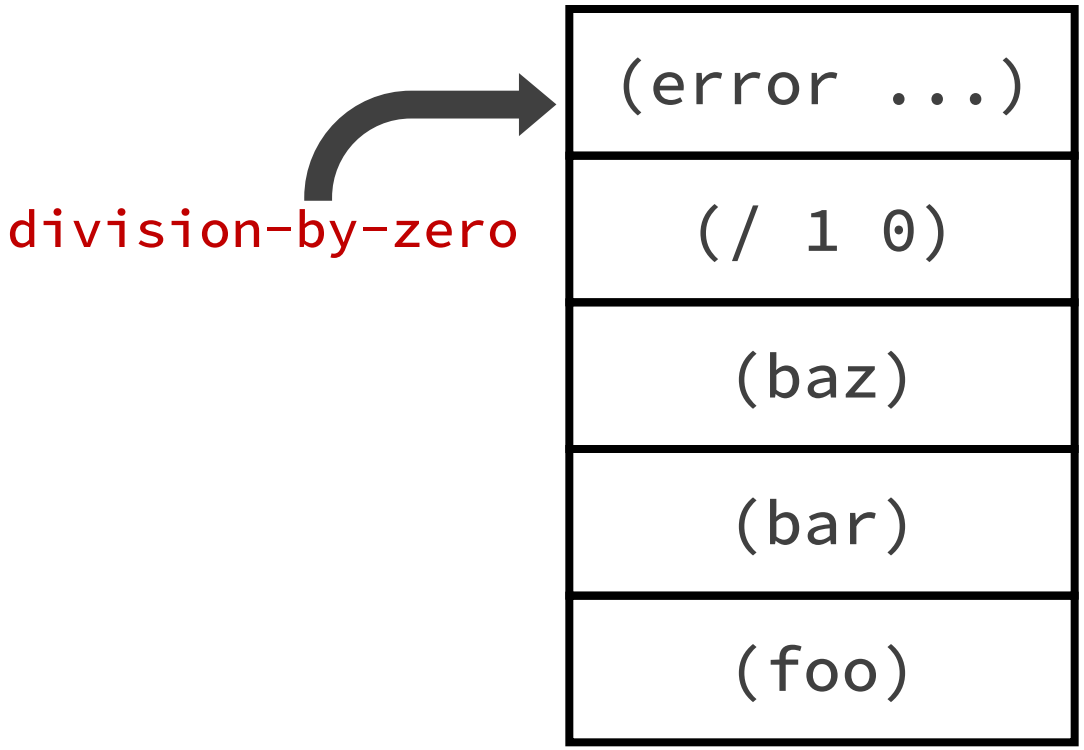
Let’s talk about signaling an error



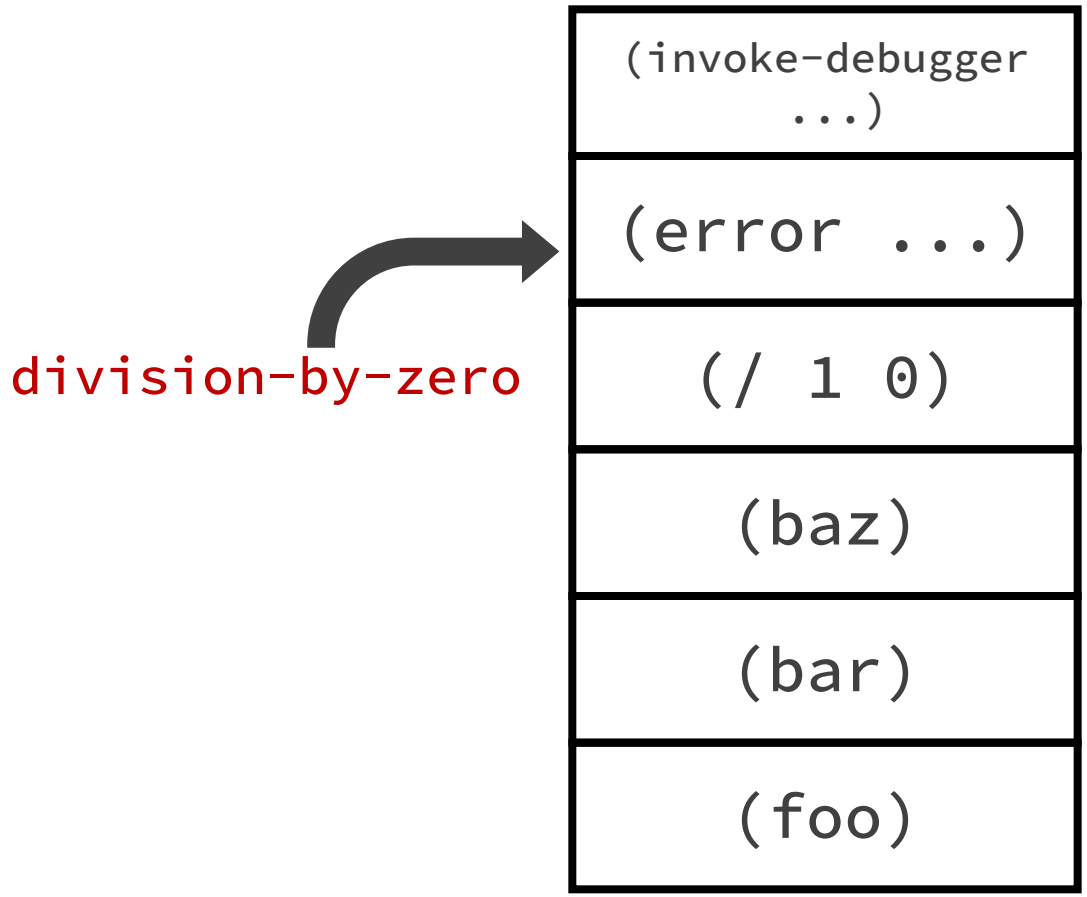
Let’s talk about signaling an error



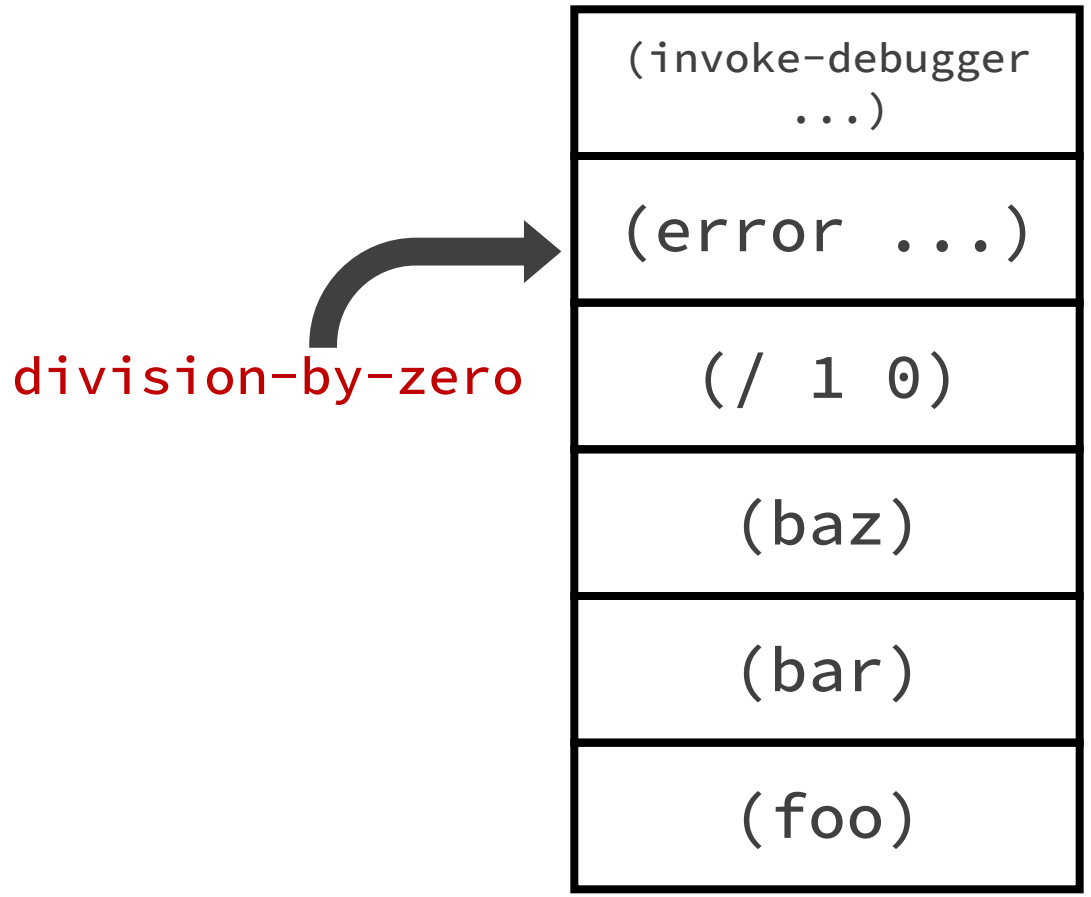
Let’s talk about signaling an error



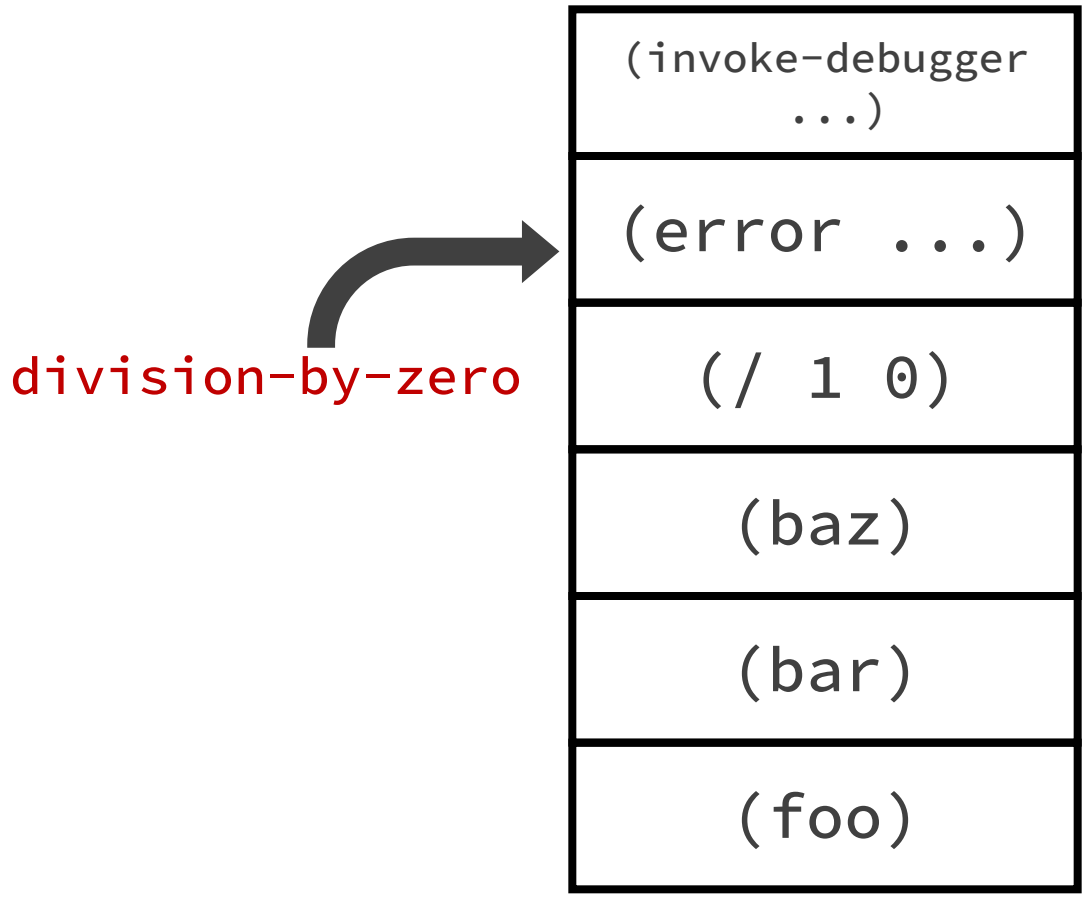
Let’s talk about signaling an error



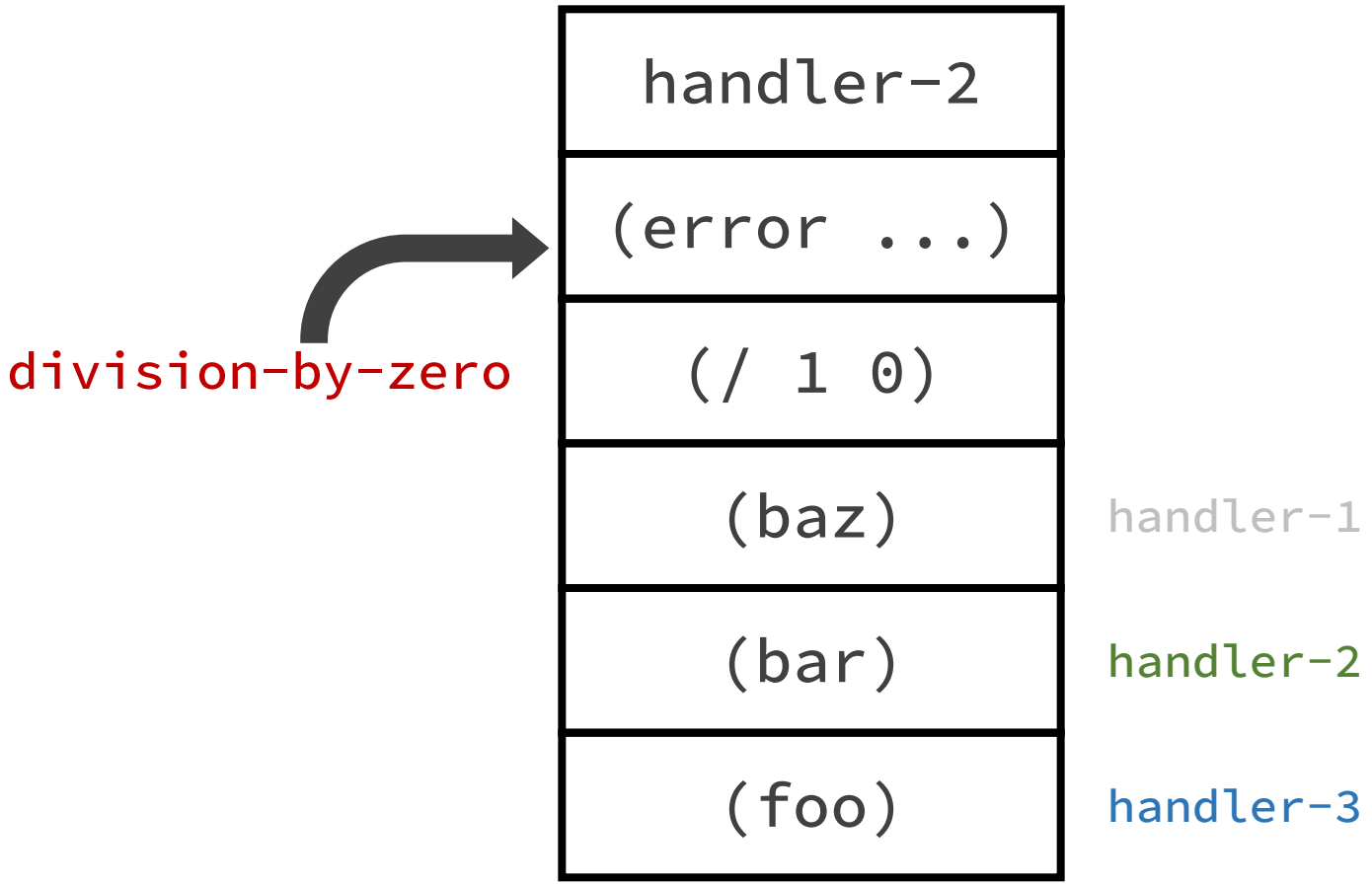
Let's talk about signaling an error



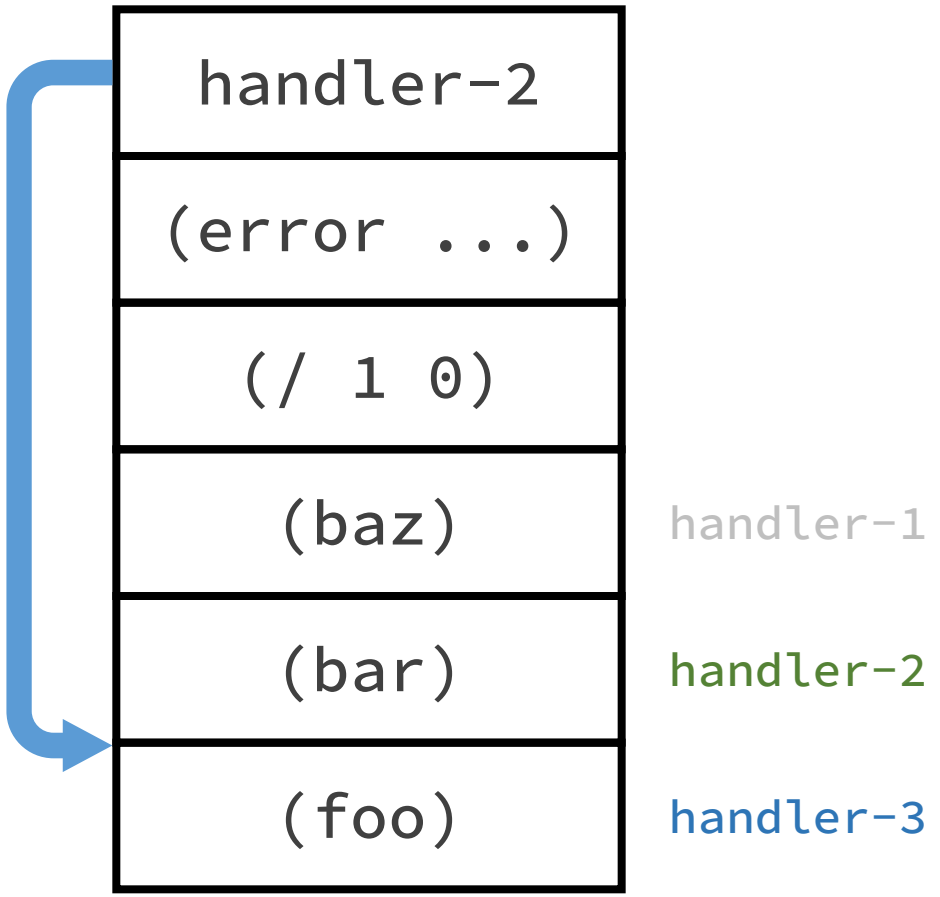
Let's talk about unwinding the stack



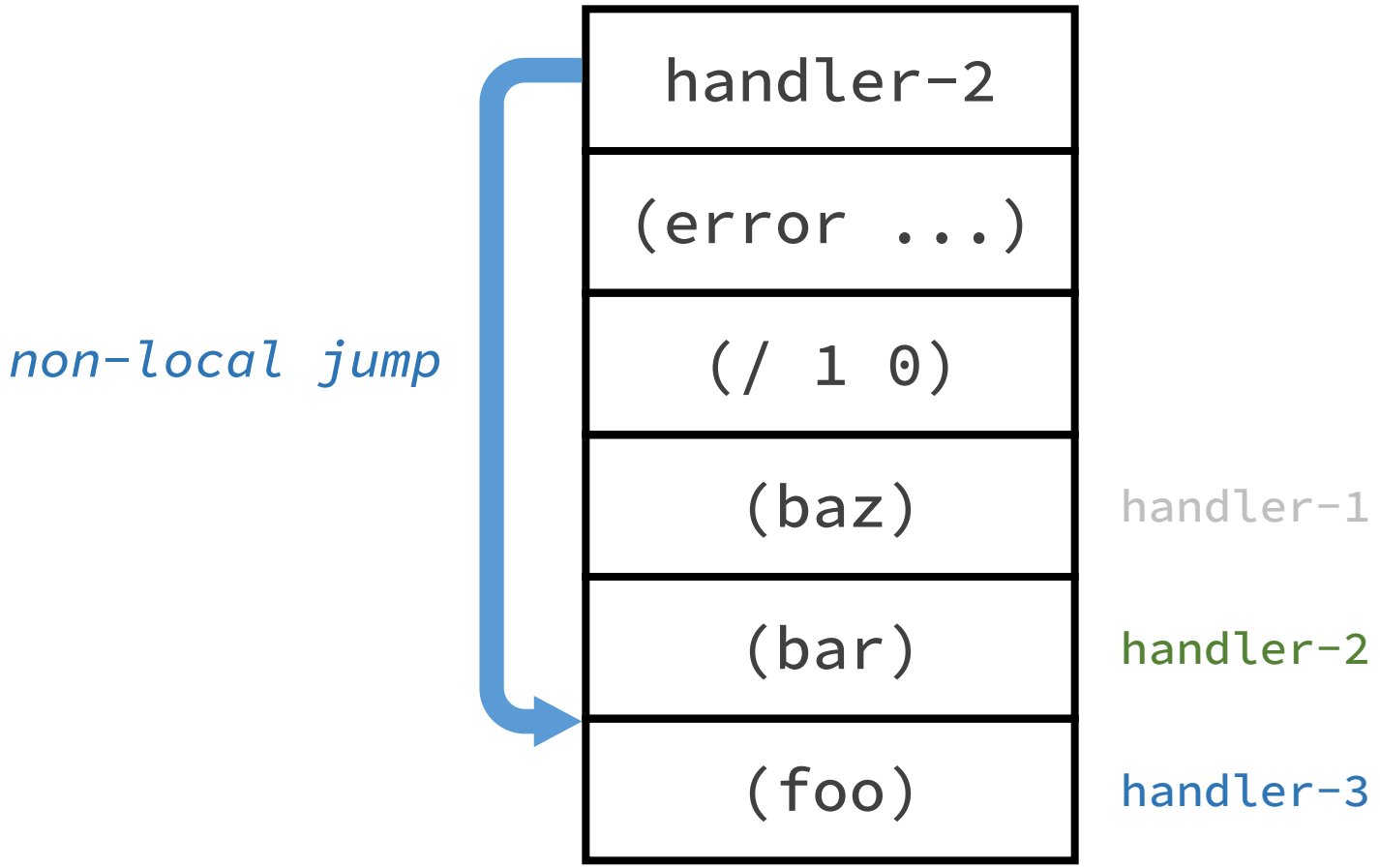
Let’s talk about unwinding the stack



Let’s talk about unwinding the stack

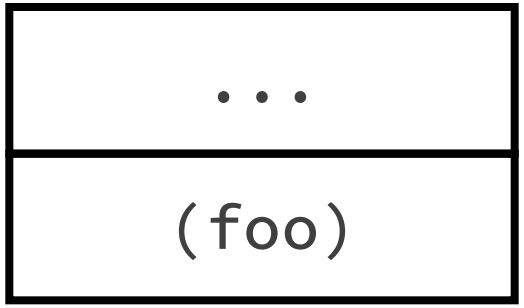


Let's talk about unwinding the stack



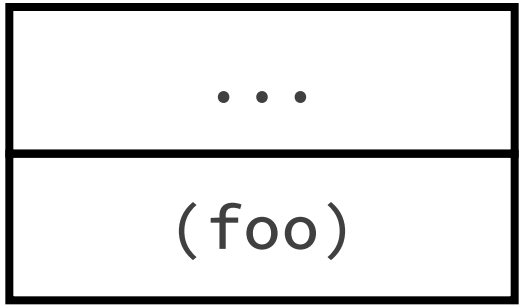
Let’s talk about unwinding the stack

Let’s talk about unwinding the stack



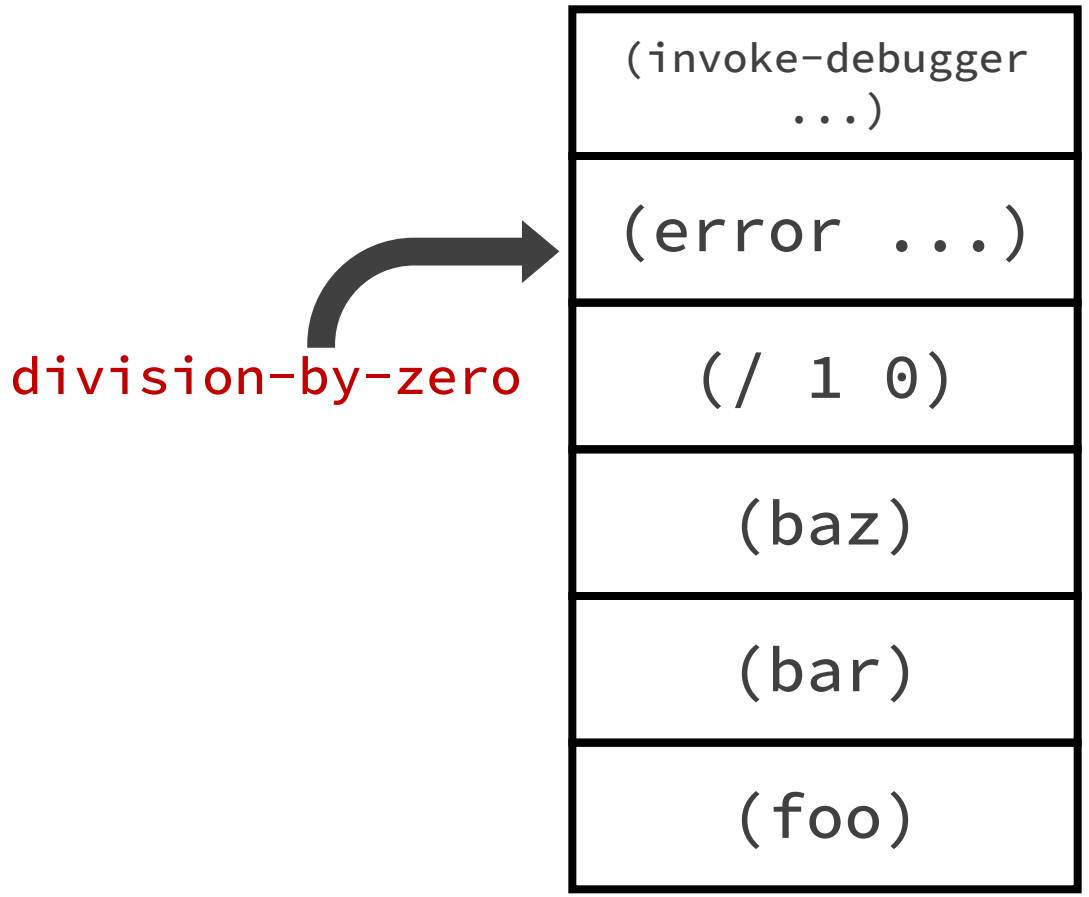

```
// execution continues
```

Let’s talk about restarts

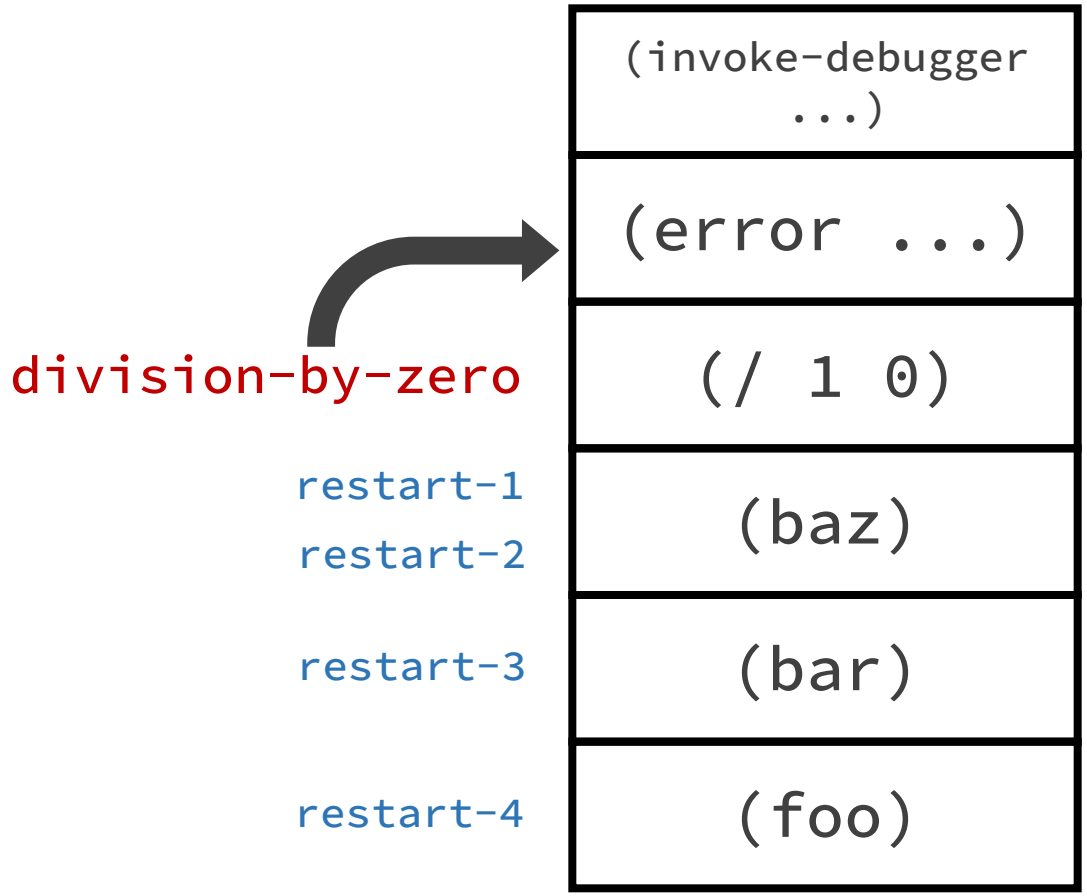



```
// execution continues
```

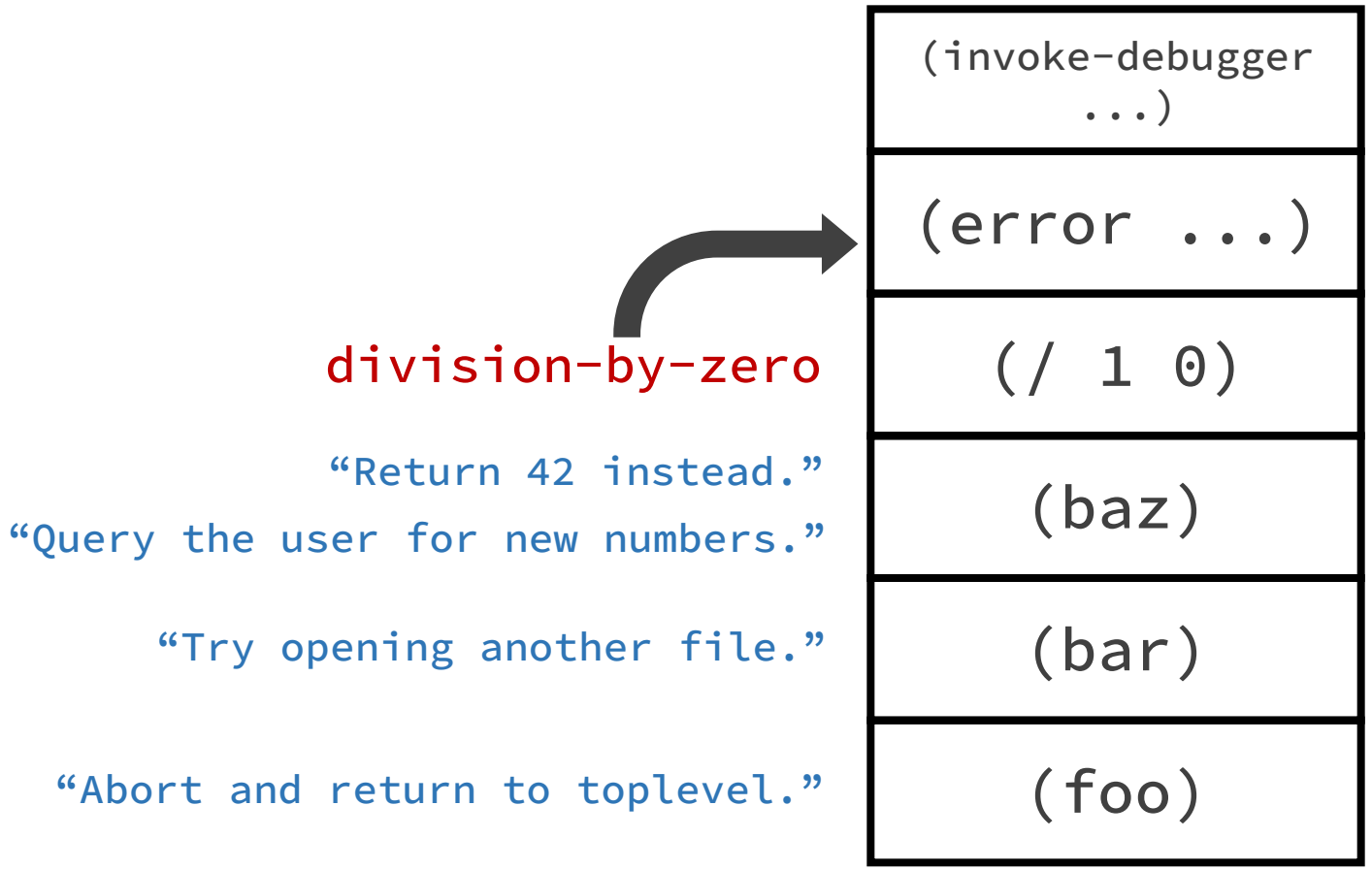
Let's talk about restarts



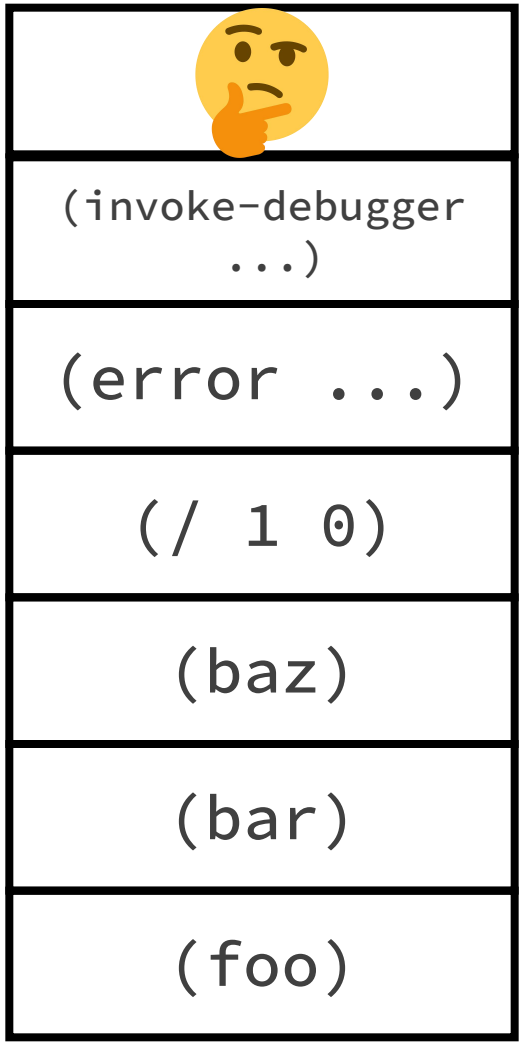
Let's talk about restarts



Let's talk about restarts



Let’s talk about restarts



division-by-zero

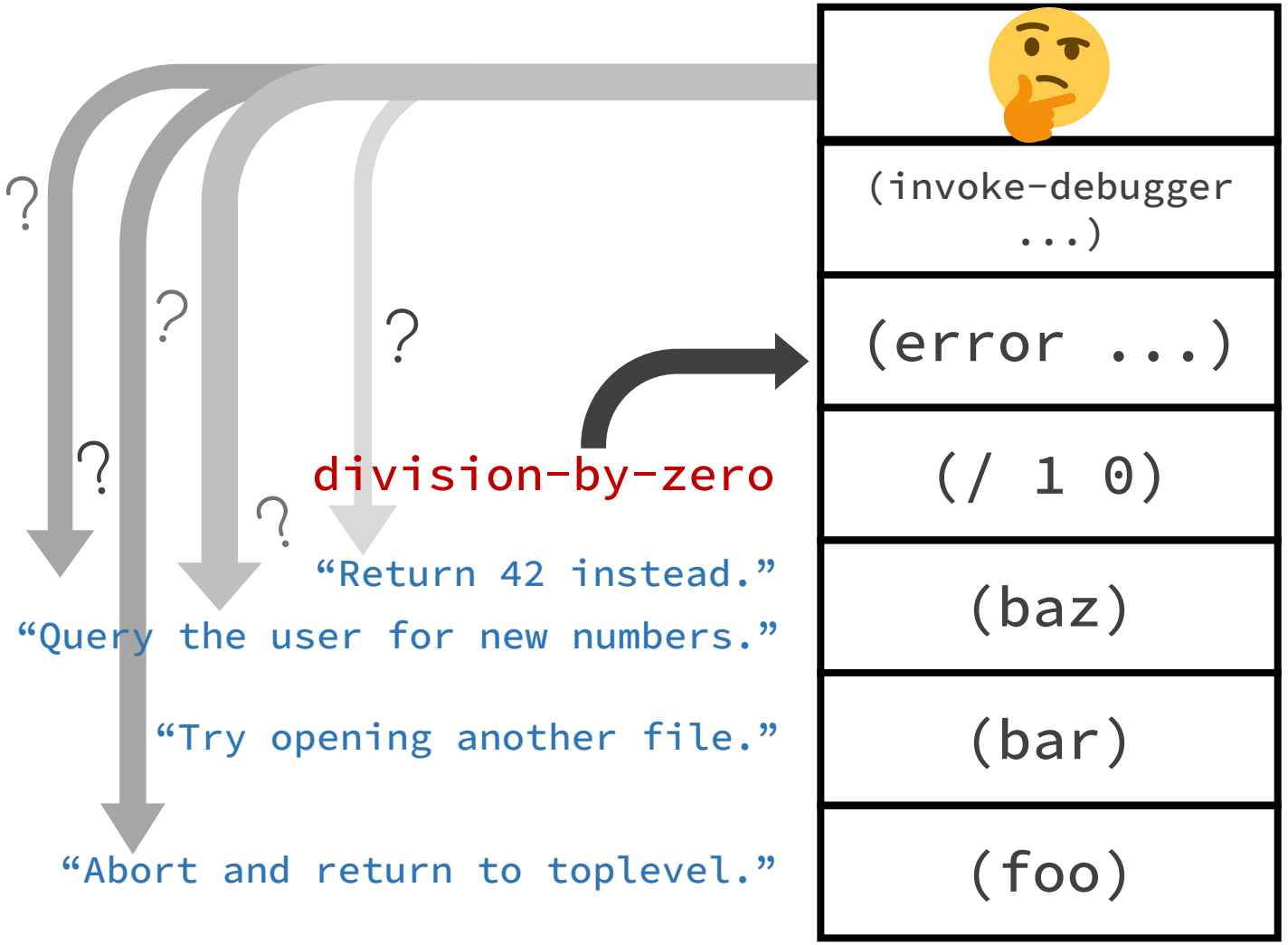
“Return 42 instead.”

“Query the user for new numbers.”

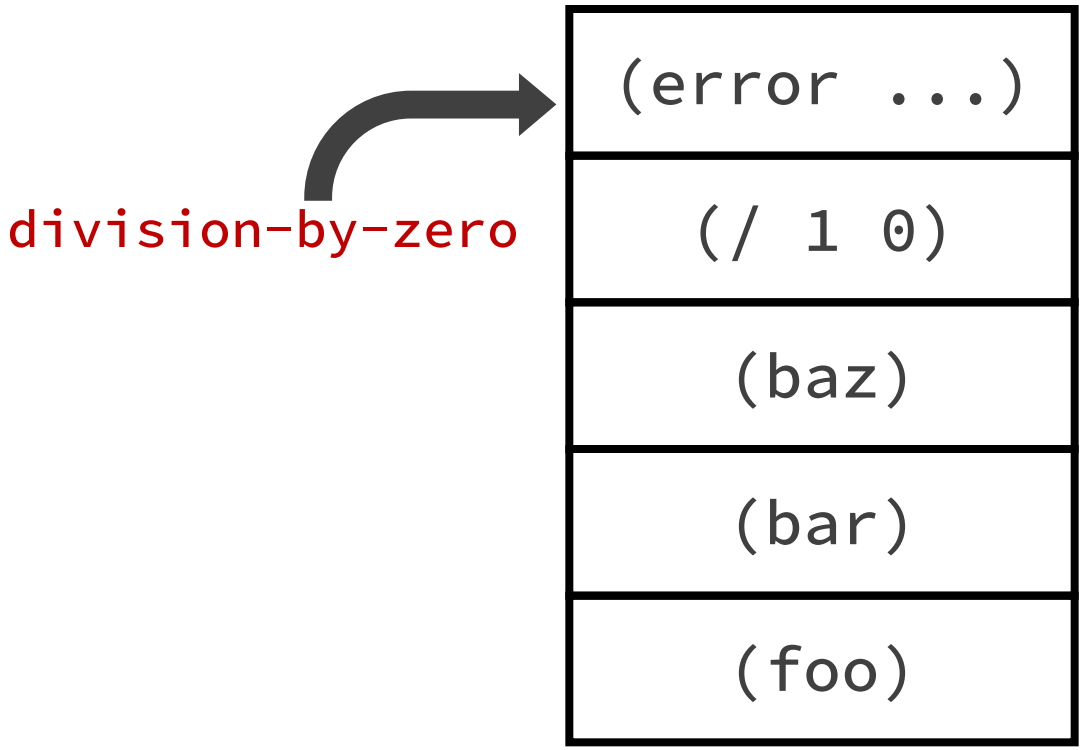
“Try opening another file.”

“Abort and return to toplevel.”

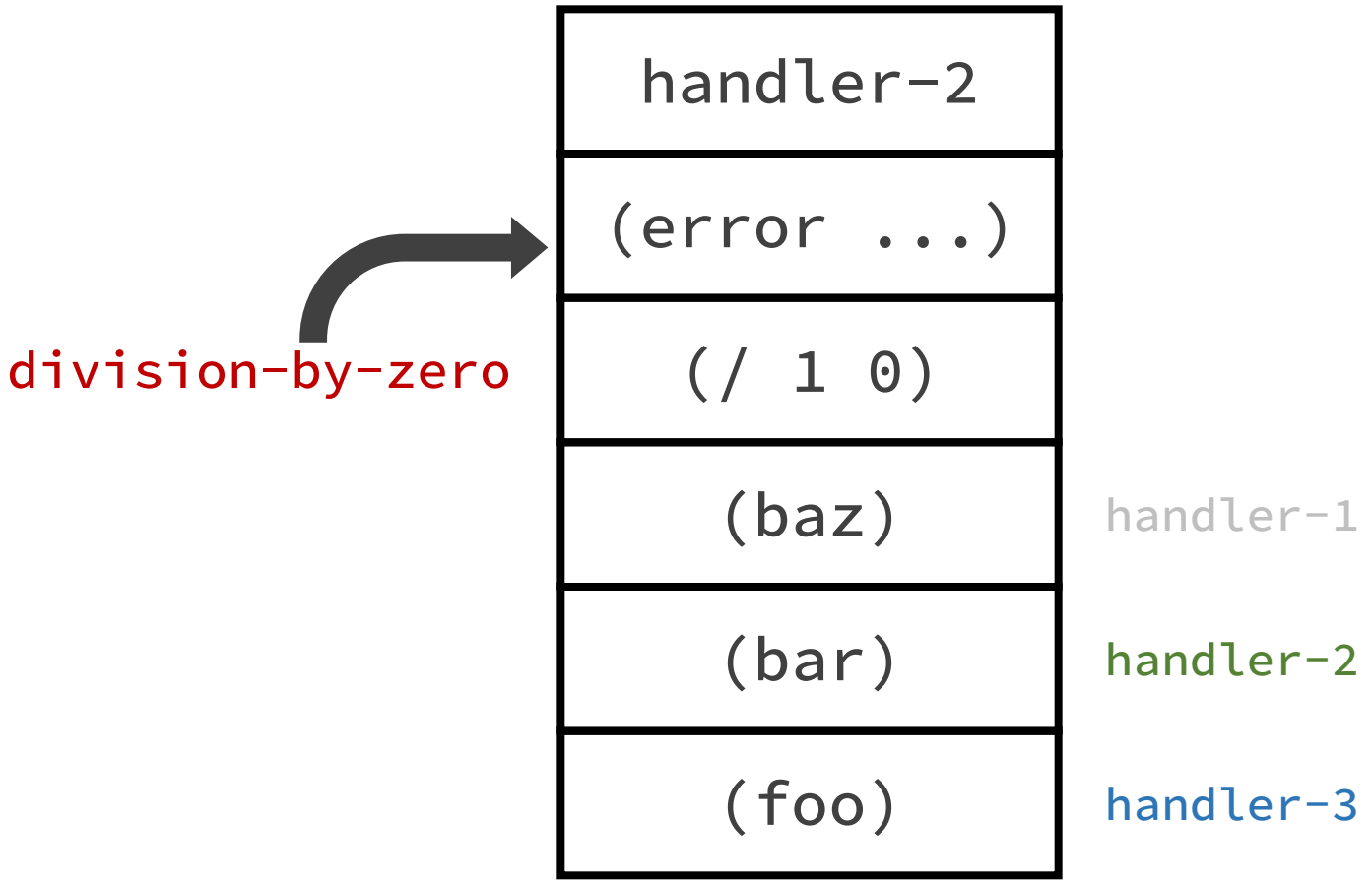
Let's talk about restarts



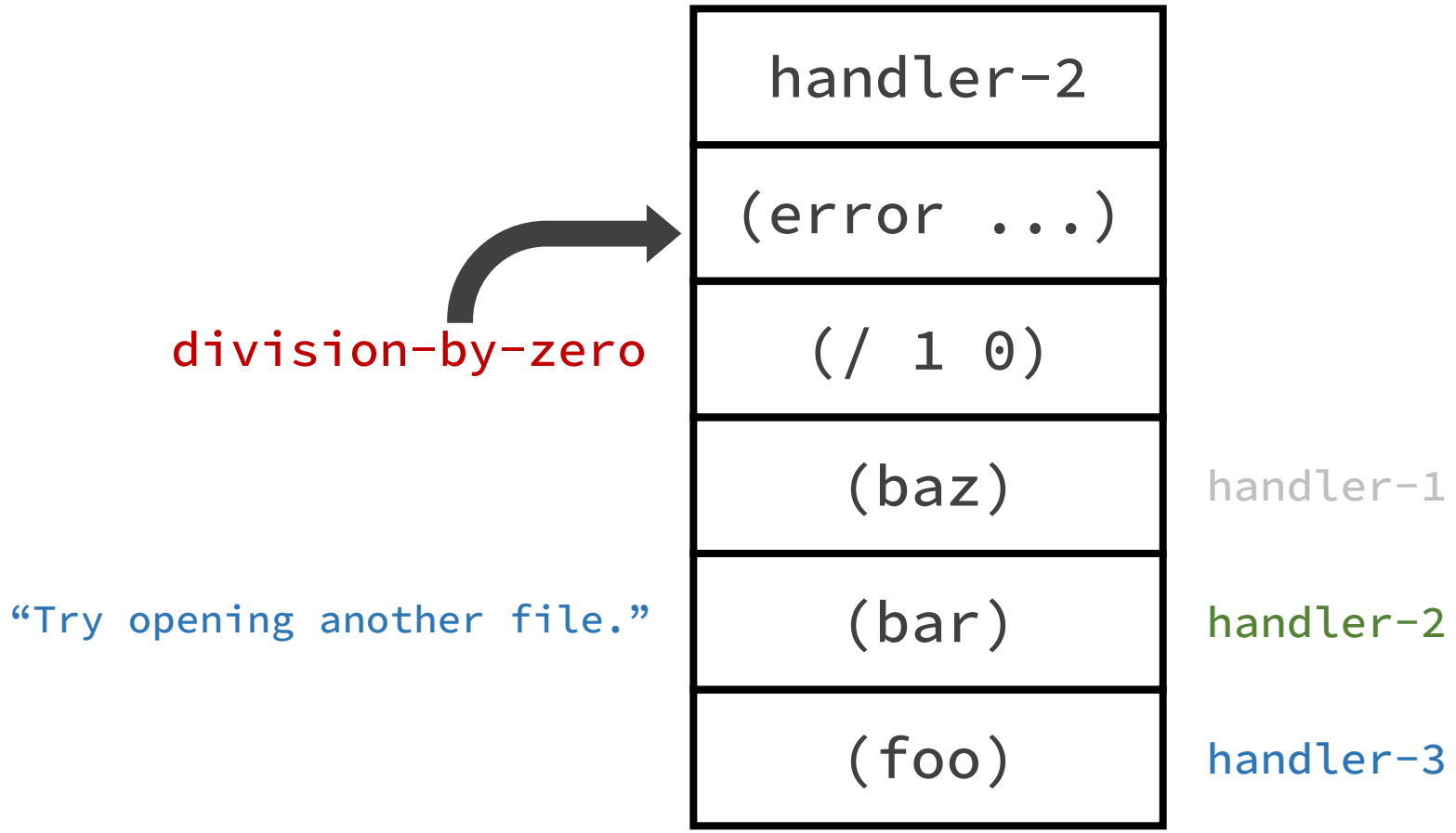
Let’s talk about restarts



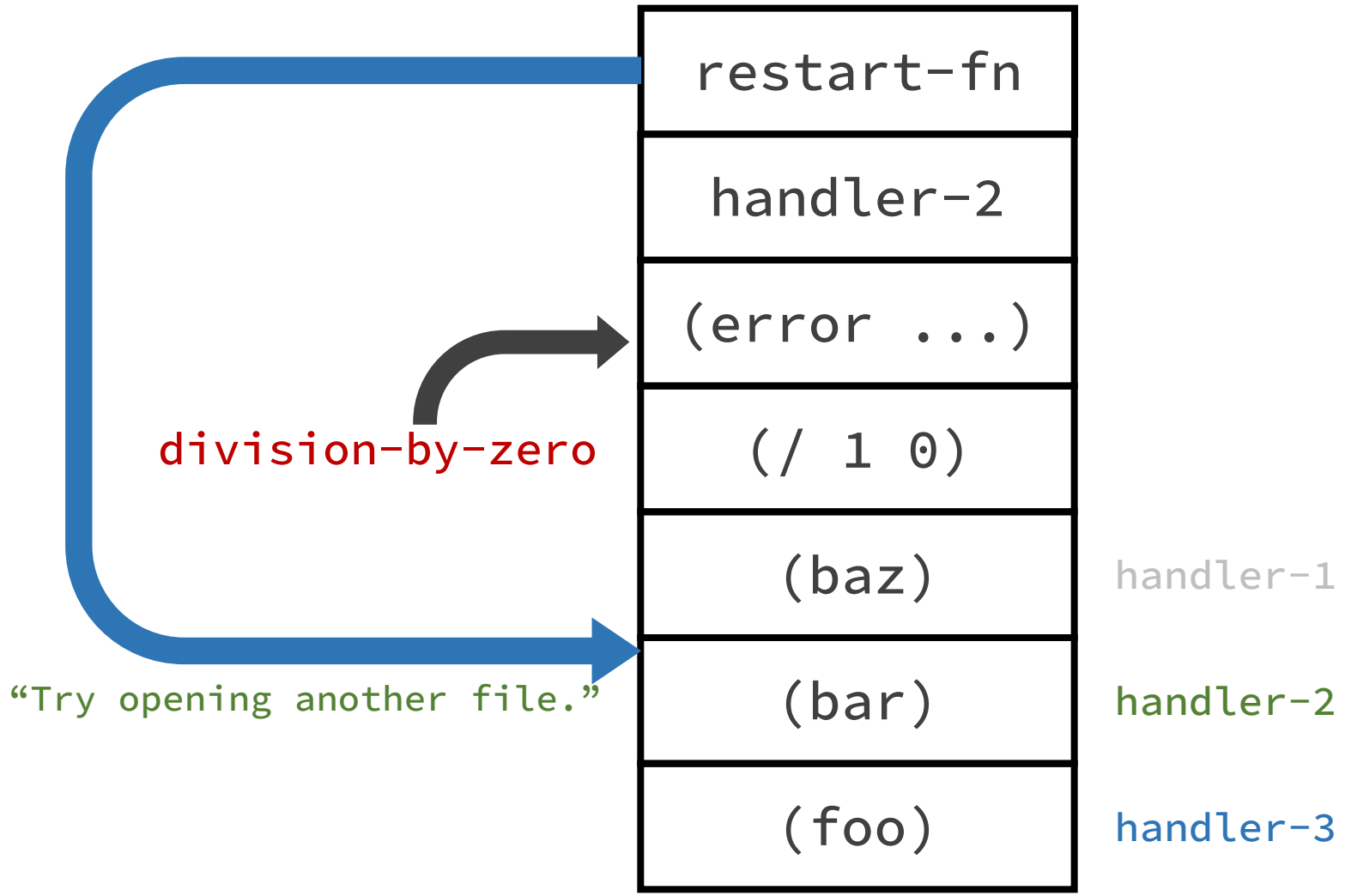
Let’s talk about restarts



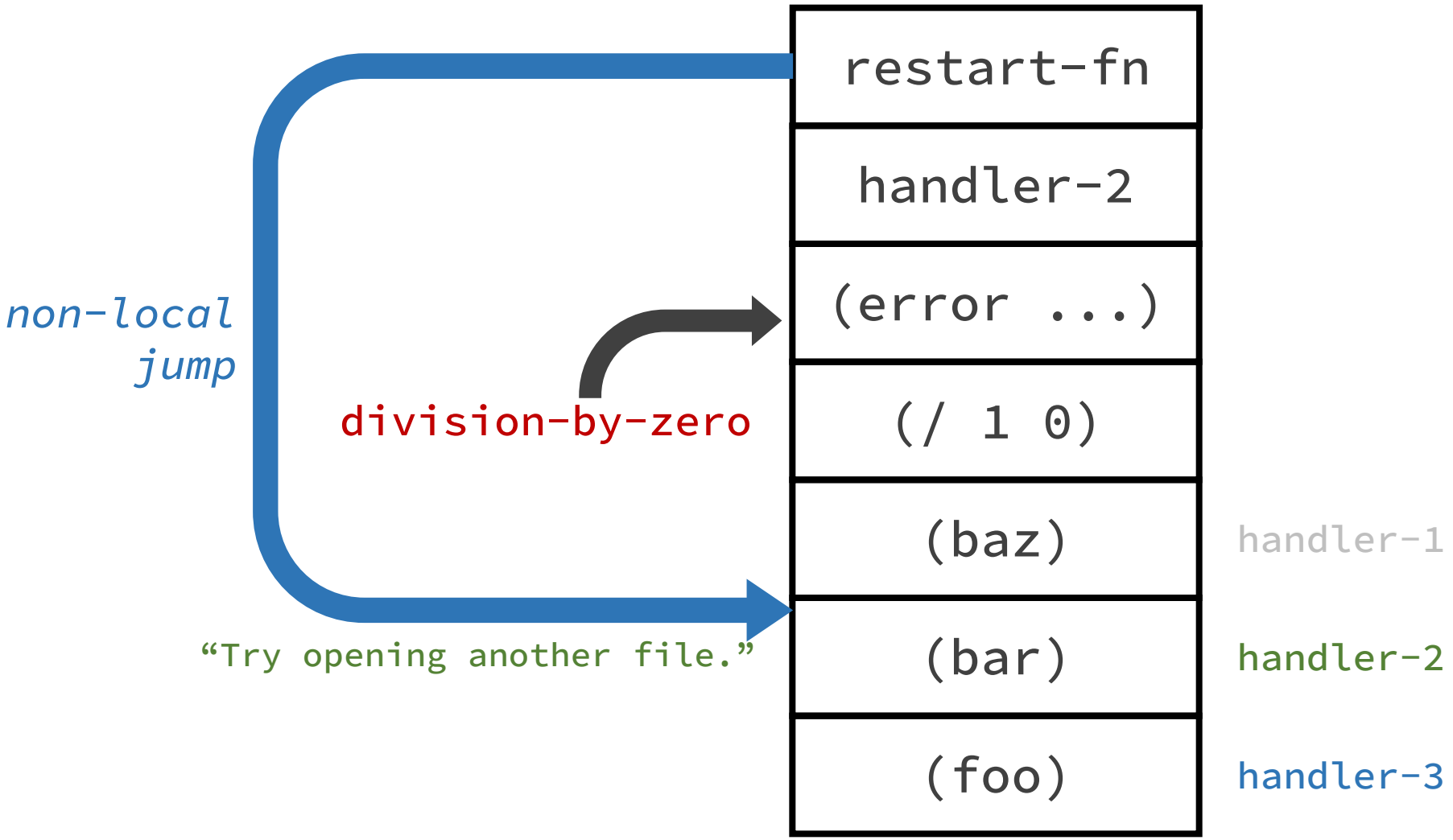
Let’s talk about restarts



Let’s talk about restarts

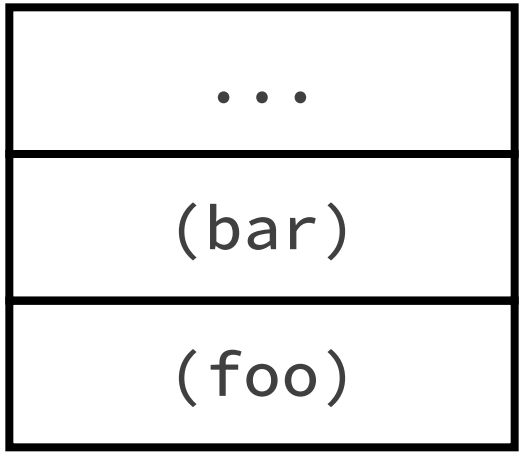


Let's talk about restarts



Let’s talk about restarts

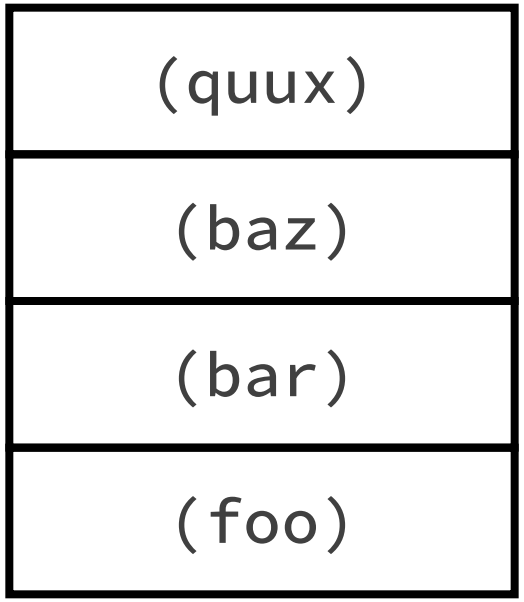
Let’s talk about restarts




```
// execution continues
```

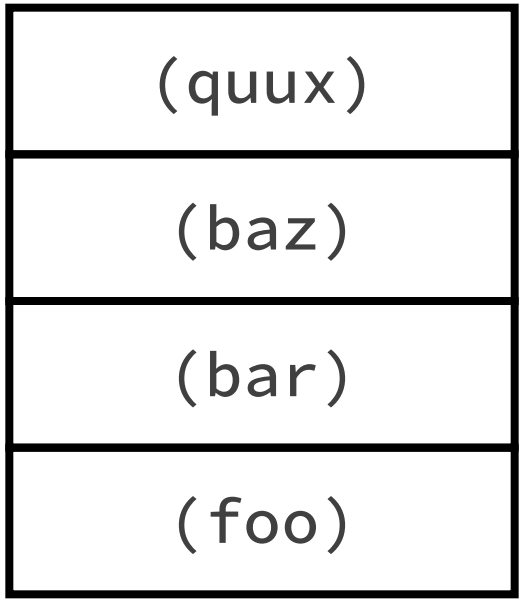
Let’s talk about signaling a non-error condition

Let’s talk about signaling a non-error condition

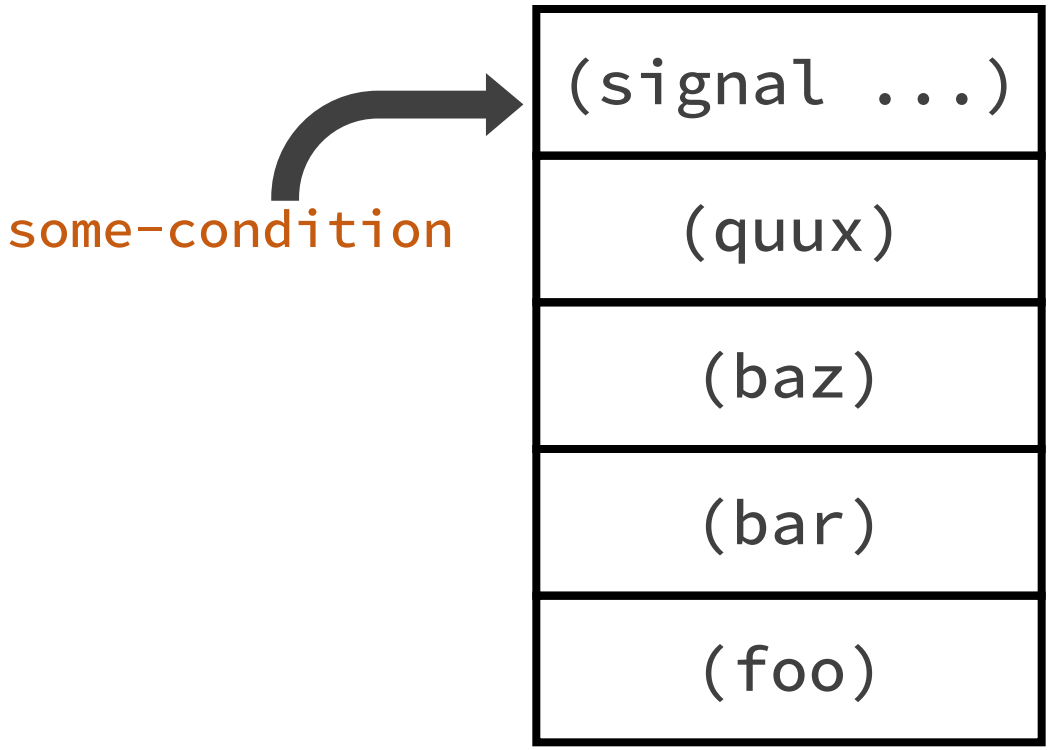


Let’s talk about signaling a non-error condition

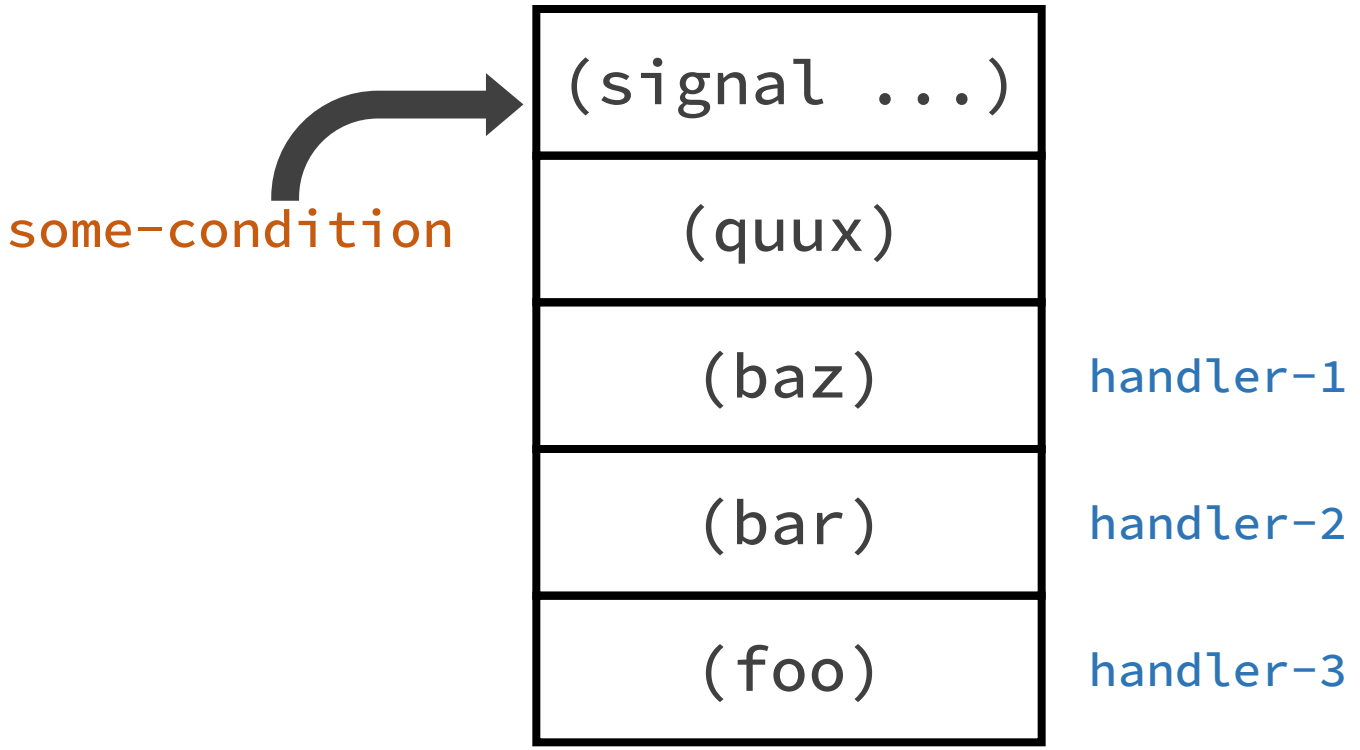
some-condition



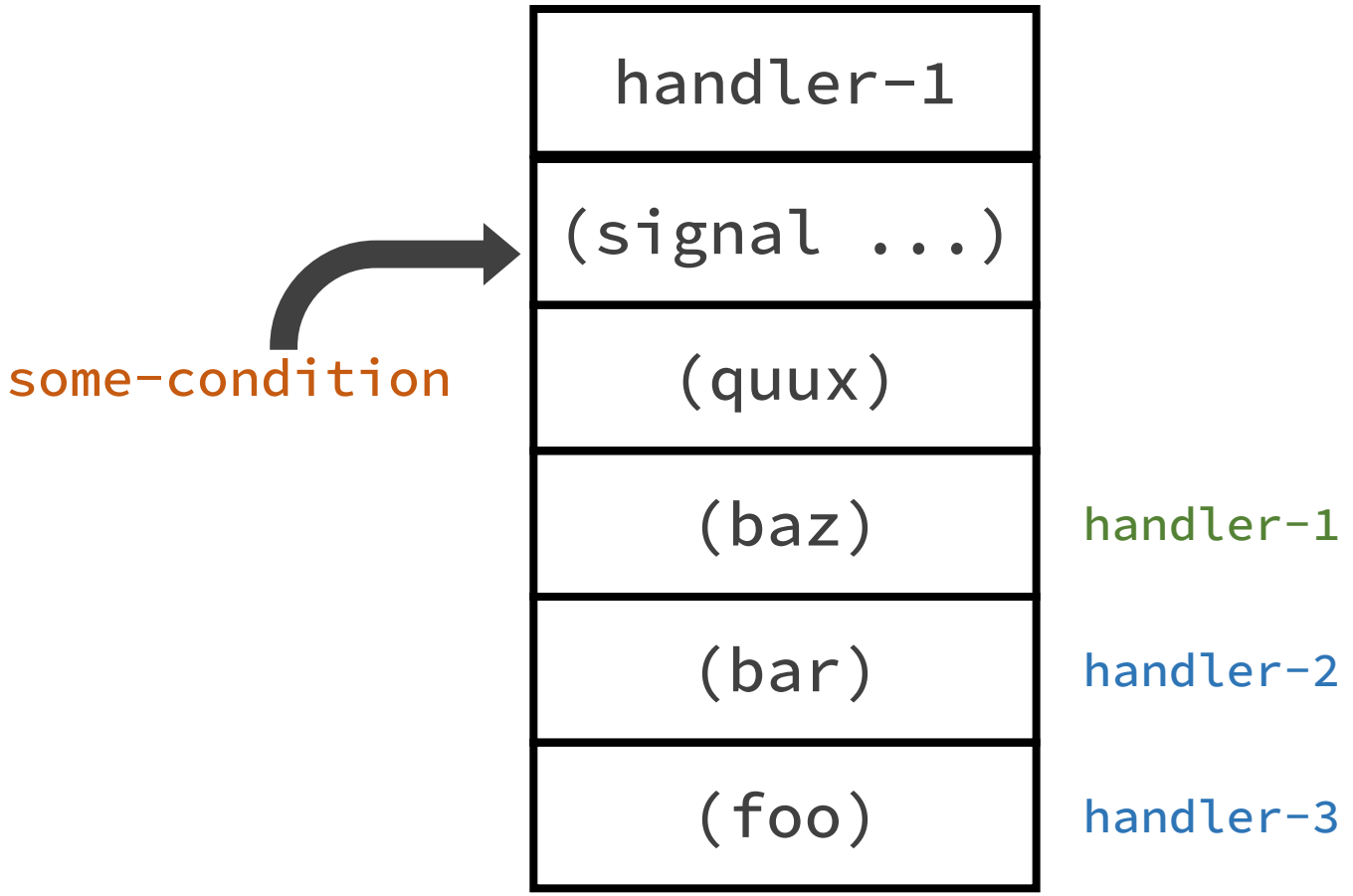
Let’s talk about signaling a non-error condition



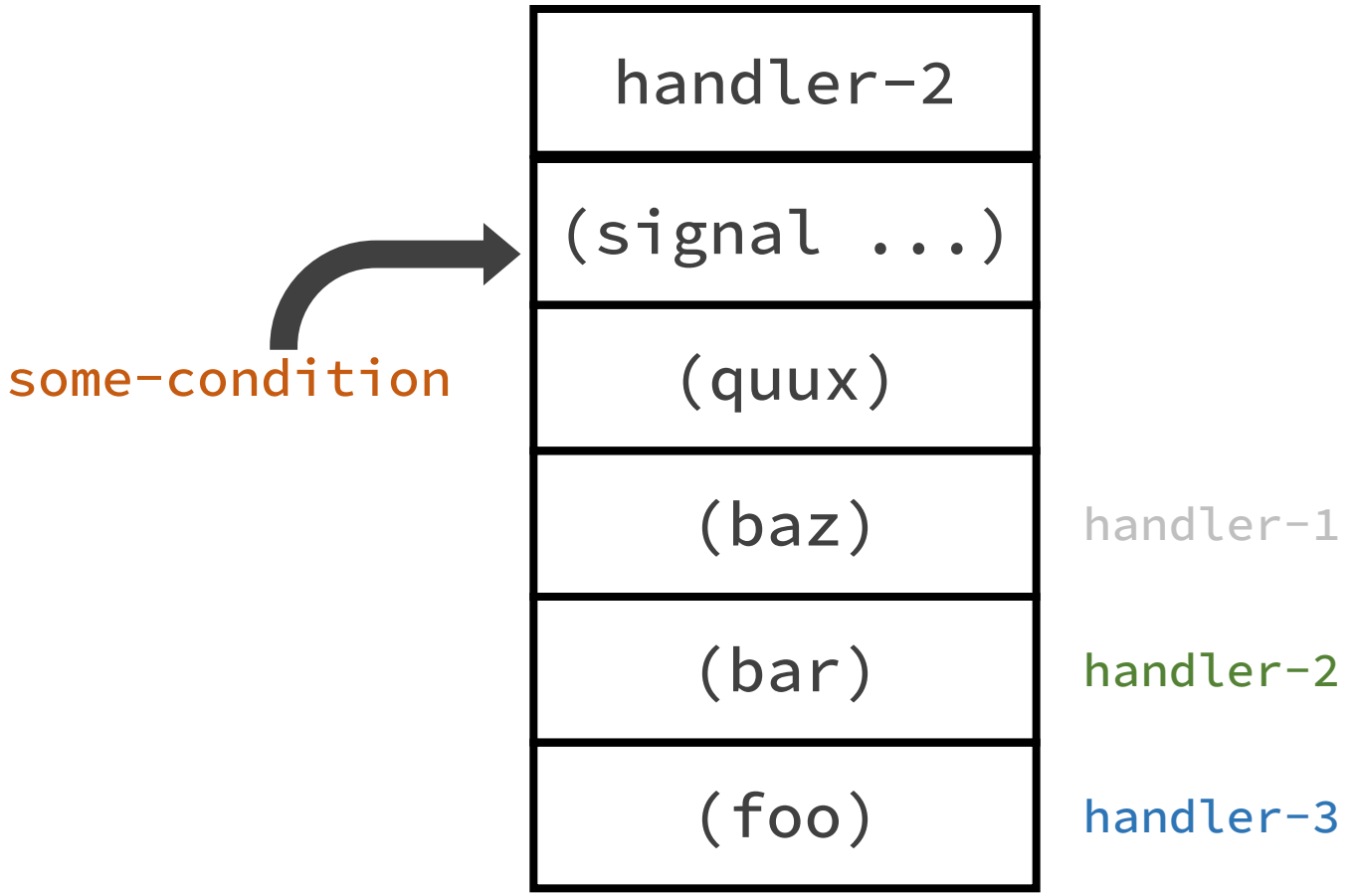
Let’s talk about signaling a non-error condition



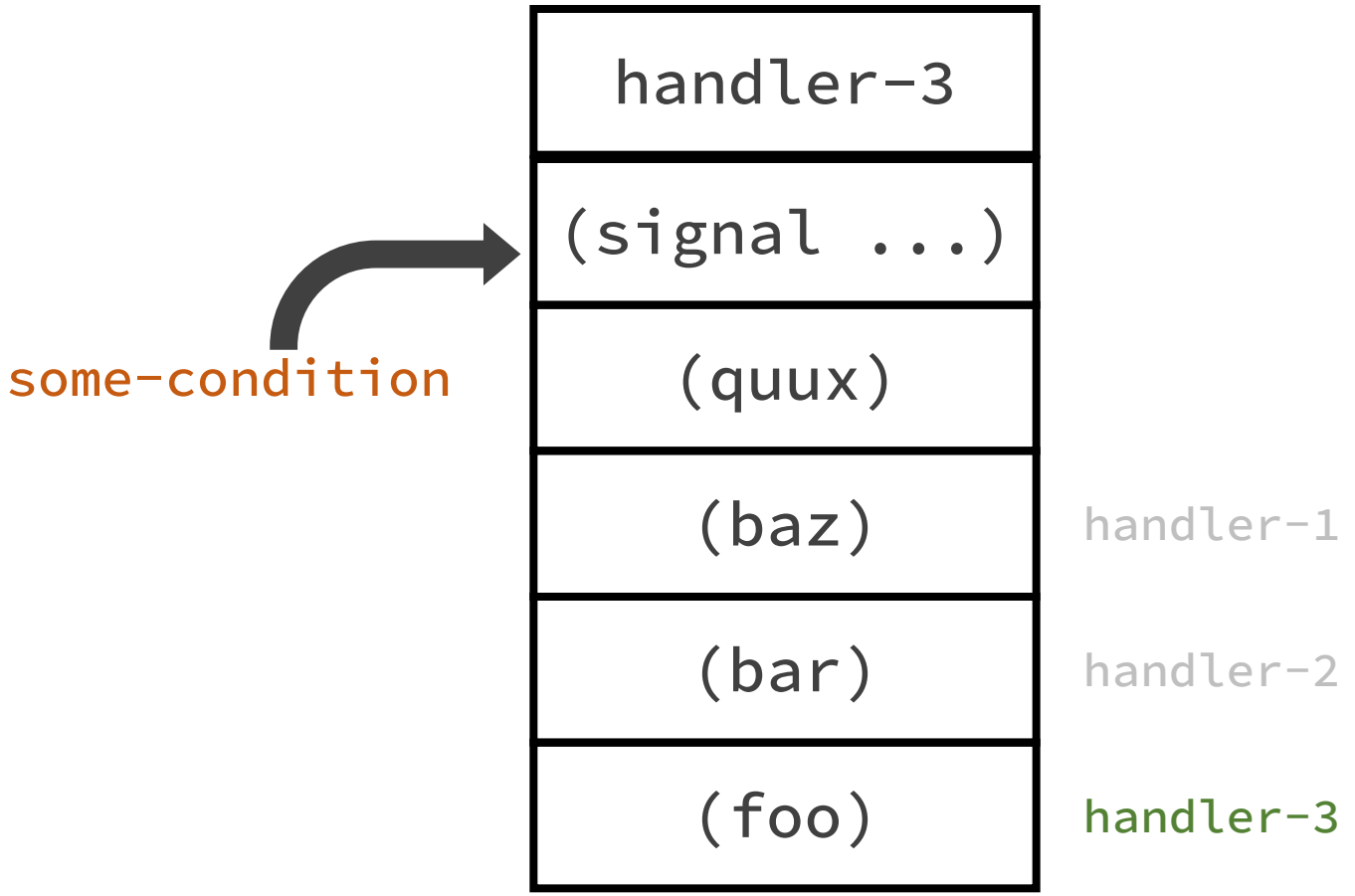
Let’s talk about signaling a non-error condition



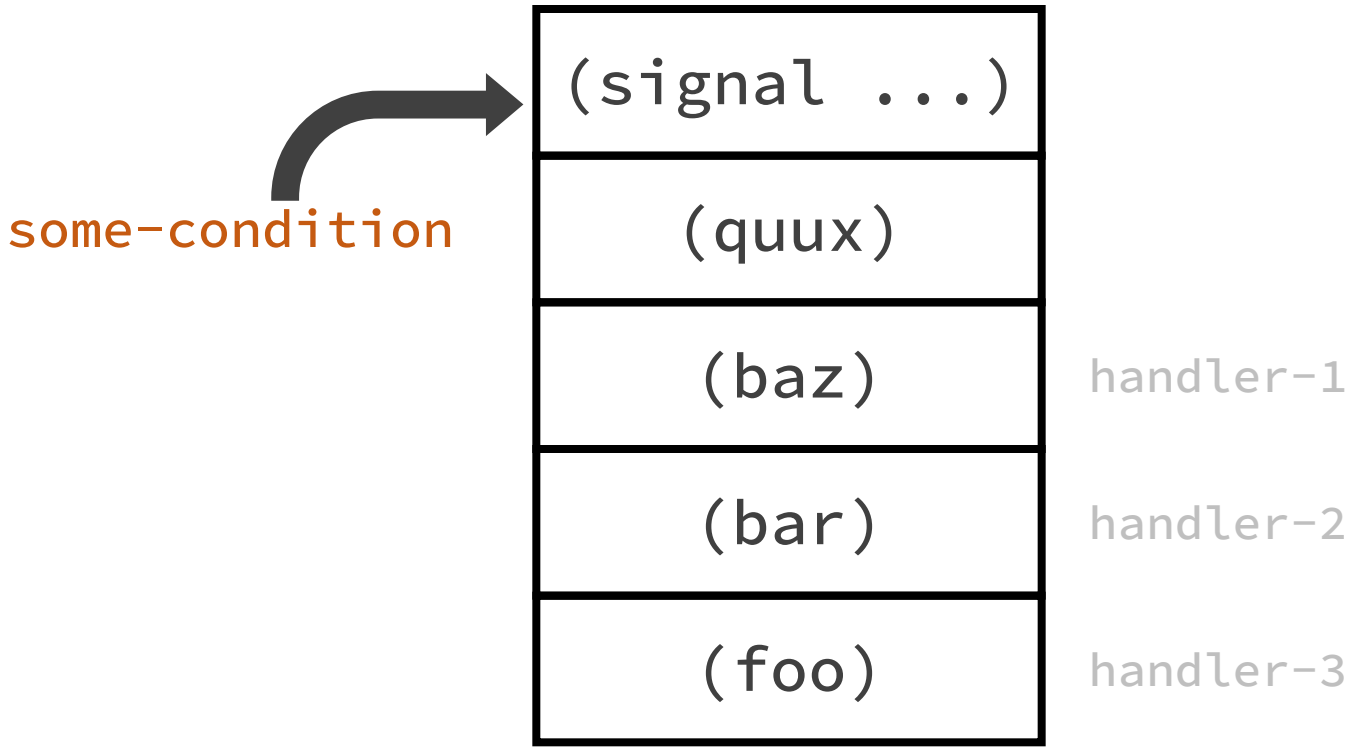
Let’s talk about signaling a non-error condition



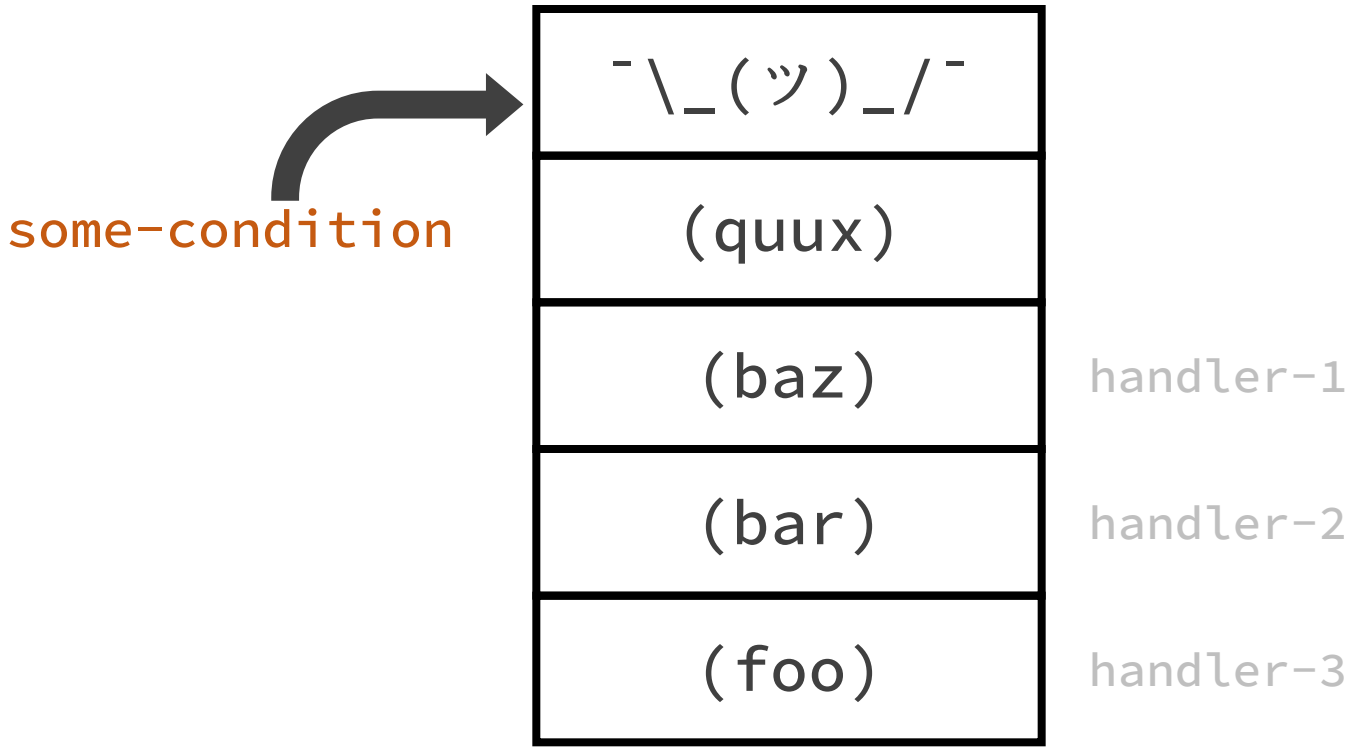
Let’s talk about signaling a non-error condition



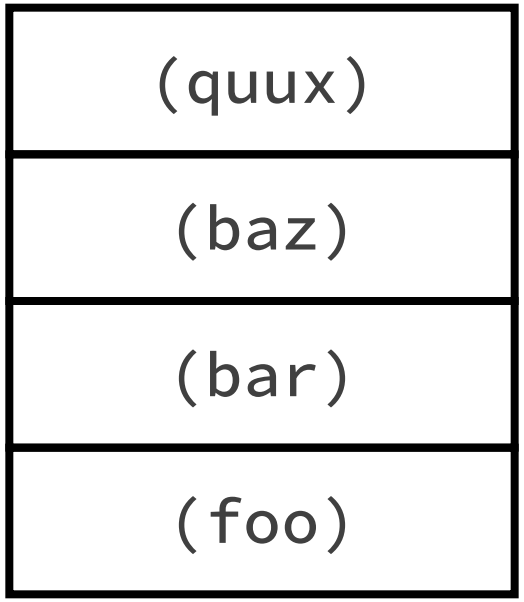
Let’s talk about signaling a non-error condition



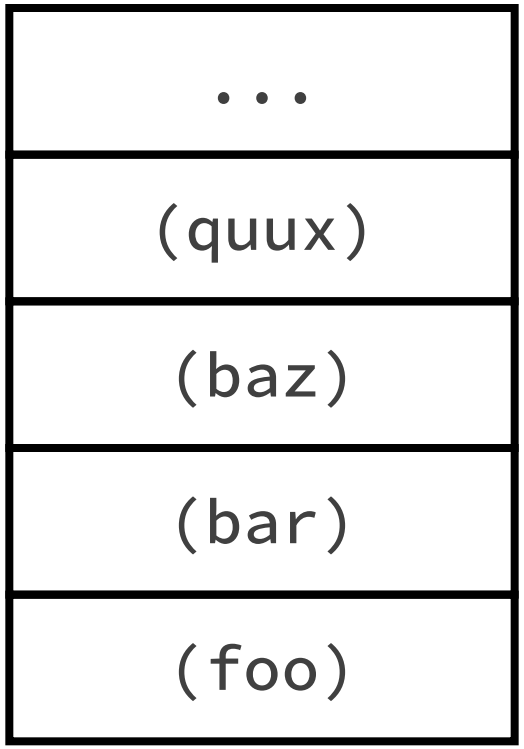
Let’s talk about signaling a non-error condition



Let’s talk about signaling a non-error condition



Let’s talk about signaling a non-error condition




```
// execution continues
```

Let’s talk about throwing exceptions

Let’s talk about throwing exceptions

- Construct the exception object

Let’s talk about throwing exceptions

- Construct the exception object
- Unwind the stack immediately
  - Stop unwinding when something catches the exception

Let’s talk about throwing exceptions

- Construct the exception object
- Unwind the stack immediately
  - Stop unwinding when something catches the exception
- Continue execution from that point

Let’s talk about signaling conditions

Let’s talk about signaling conditions

- Construct the condition object

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™

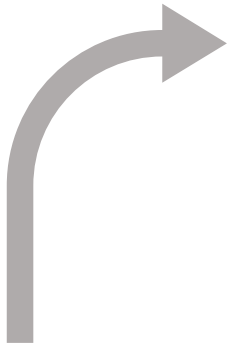
Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™

- If there was no transfer of control, return
  - …and maybe enter the debugger to halt the program

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™



- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

- If there was no transfer of control, return
  - …and maybe enter the debugger to halt the program

Let's talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she's born with it
    - Maybe it's Maybelline™
- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..

- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

Let’s talk about conditions versus exceptions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™

- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..

- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

Let’s talk about conditions versus exceptions

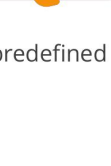
Let’s talk about throwing exceptions

- Construct the exception object
- Unwind the stack immediately
  - Stop unwinding when something catches the exception
- Continue execution from that point

77

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™
- If there was no transfer of control, return
  - …and maybe enter the debugger to halt the program



- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..

- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

91

Let’s talk about conditions versus exceptions

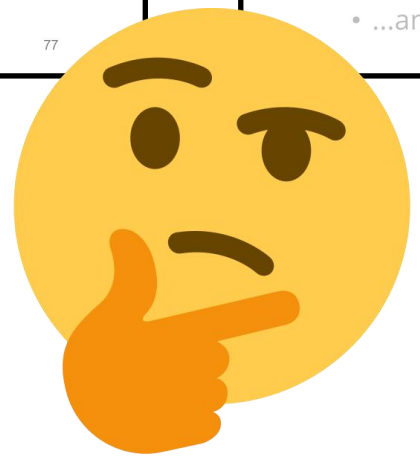
Let’s talk about throwing exceptions

- Construct the exception object
- Unwind the stack immediately
  - Stop unwinding when something catches the exception
- Continue execution from that point

77

Let’s talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™
- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program



- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..
- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

91

Let's talk about signaling conditions

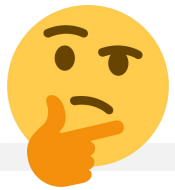
- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she's born with it
    - Maybe it's Maybelline™
- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..

- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

Let's talk about signaling conditions

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she's born with it
    - Maybe it's Maybelline™



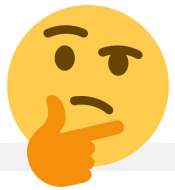
- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..

- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

Let’s talk about non-local control flow

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™



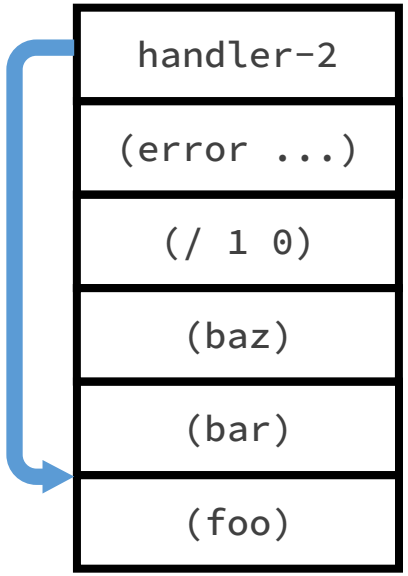
- restarts = a dynamically scoped mechanism of choices
  - context-dependent actions for interactive programming
  - context-dependent means of automated error recovery (e.g. when parsing incomplete source code)
  - etc..

- signaling = a dynamically scoped hooking mechanism
  - progress bars
  - message passing
  - calling asynchronous code
  - etc..

- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

Let’s talk about non-local control flow

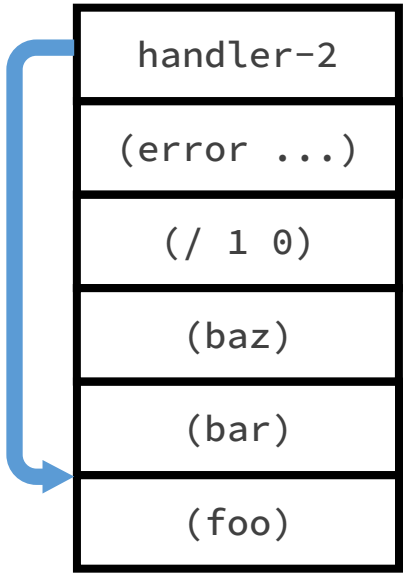
- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™



- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

Let’s talk about control flow

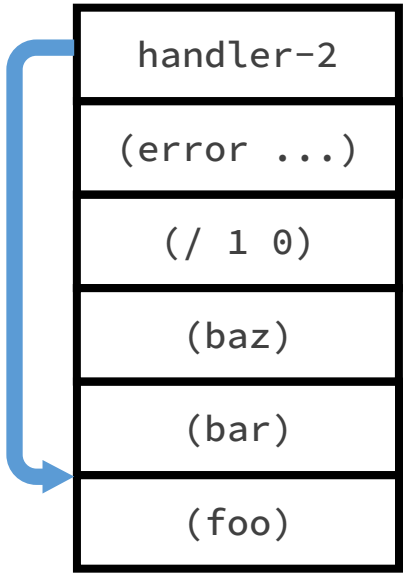
- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™



- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

Let’s talk about control flow in Common Lisp

- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she’s born with it
    - Maybe it’s Maybelline™



- If there was no transfer of control, return
  - ...and maybe enter the debugger to halt the program

Control Flow in Common Lisp

aka Why Lisp Doesn't Need To Throw Exceptions

Appendix B

Proving one-phase unwind in TAGBODY and BLOCK

Let’s talk about non-local control flow in Common Lisp

- `if`
- `tagbody/go` `; 1-phase unwind (no search)`
- `block/return-from` `; 1-phase unwind (no search)`
- `catch/throw` `; 2-phase unwind (search)`
- `unwind-protect`
- `lambda/apply`

Let’s talk about non-local control flow in Common Lisp

- `if`
- `tagbody/go` `; 1-phase unwind (no search)`
- `block/return-from` `; 1-phase unwind (no search)`
- `catch/throw` `; 2-phase unwind (search)`
- `unwind-protect`
- `lambda/apply`

Let’s talk about one-phase unwind in Common Lisp

- `if`
- `tagbody/go` `; 1-phase unwind (no search)`
- `block/return-from` `; 1-phase unwind (no search)`
- `catch/throw` `; 2-phase unwind (search)`
- `unwind-protect`
- `lambda/apply`

Let's talk about one-phase unwind in Common Lisp

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply

```
                    ; 1-phase unwind (no search)
                    ; 1-phase unwind (no search)
                    ; 2-phase unwind (search)
(block foo
  (lambda ()
    (return-from foo)))
```

Let's talk about one-phase unwind in Common Lisp

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply

```
                    ; 1-phase unwind (no search)
                    ; 1-phase unwind (no search)
                    ; 2-phase unwind (search)
(let ((fn (block foo
            (lambda ()
              (return-from foo)))))
  (funcall fn))
; ...?
```

Let's talk about one-phase unwind in Common Lisp

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply

```
                    ; 1-phase unwind (no search)
                    ; 1-phase unwind (no search)
                    ; 2-phase unwind (search)
```

```
(let ((fn (block foo
            (lambda ()
              (return-from foo)))))
  (funcall fn))
; ERROR: Condition CONTROL-ERROR
; was signaled.
```

Let's talk about one-phase unwind in Common Lisp

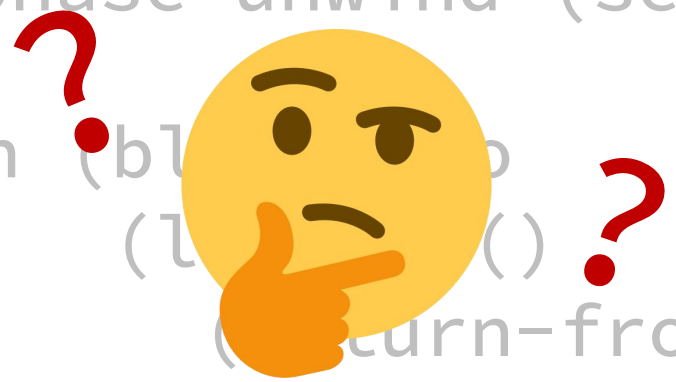
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply

```
                                ; 1-phase unwind (no search)
                                ; 1-phase unwind (no search)
                                ; 2-phase unwind (search)
(let ((fn (block foo
            (lambda ()
              (return-from foo)))))
  (funcall fn))
; ERROR: Condition CONTROL-ERROR
; was signaled.
```

Let's talk about one-phase unwind in Common Lisp

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply

```
                                ; 1-phase unwind (no search)
                                ; 1-phase unwind (no search)
                                ; 2-phase unwind (search)
(let ((fn (bl                 p
            (l                ()
                                (return-from foo)))))
  (funcall fn))
; ERROR: Condition CONTROL-ERROR
; was signaled.
```



Let’s talk about one-phase unwind in Common Lisp

```
(block foo
  (lambda ()
    (return-from foo)))
```

Let’s talk about one-phase unwind in Common Lisp

```
(block foo
  ...)
```

Let’s talk about one-phase unwind in Common Lisp

```
(let ((return-valid-p t))
  (unwind-protect
      (%unwind-tag foo
        ...)
    (setf return-valid-p nil)))
```

Let’s talk about one-phase unwind in Common Lisp

```
(let ((return-valid-p t))
  (unwind-protect
      (%unwind-tag foo
        ...) ;; let’s expand the lambda!
    (setf return-valid-p nil)))
```

Let’s talk about one-phase unwind in Common Lisp

```
(let ((return-valid-p t))
  (unwind-protect
      (%unwind-tag foo
        (lambda ()
          (if return-valid-p
              (%1-phase-unwind-to-tag foo)
              (error ‘control-error))))
    (setf return-valid-p nil)))
```

Let’s talk about one-phase unwind in Common Lisp

```
(let ((return-valid-p t))
  (unwind-protect
      (%unwind-tag foo
        (lambda ()
          (if return-valid-p
              (%1-phase-unwind-to-tag foo)
              (error ‘control-error))))
    (setf return-valid-p nil)))
```

Let’s talk about one-phase unwind in Common Lisp

```
(let ((return-valid-p t))
  (unwind-protect
      (%unwind-tag foo
        (lambda ()
          (if return-valid-p
              (%1-phase-unwind-to-tag foo)
              (error ‘control-error))))
    (setf return-valid-p nil)))
```

Let’s talk about one-phase unwind in Common Lisp

```
(let ((return-valid-p t))
  (unwind-protect
      (%unwind-tag foo
        (lambda ()
          (if return-valid-p
              (%1-phase-unwind-to-tag foo)
              (error ‘control-error))))
    (setf return-valid-p nil)))

;; similar validation scheme applies for TAGBODY/GO
```

Control Flow in Common Lisp

aka Why Lisp Doesn't Need To Throw Exceptions

Appendix C

Describing UNWIND-PROTECT

Let’s talk about non-local control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about non-local control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about unwinding in Common Lisp

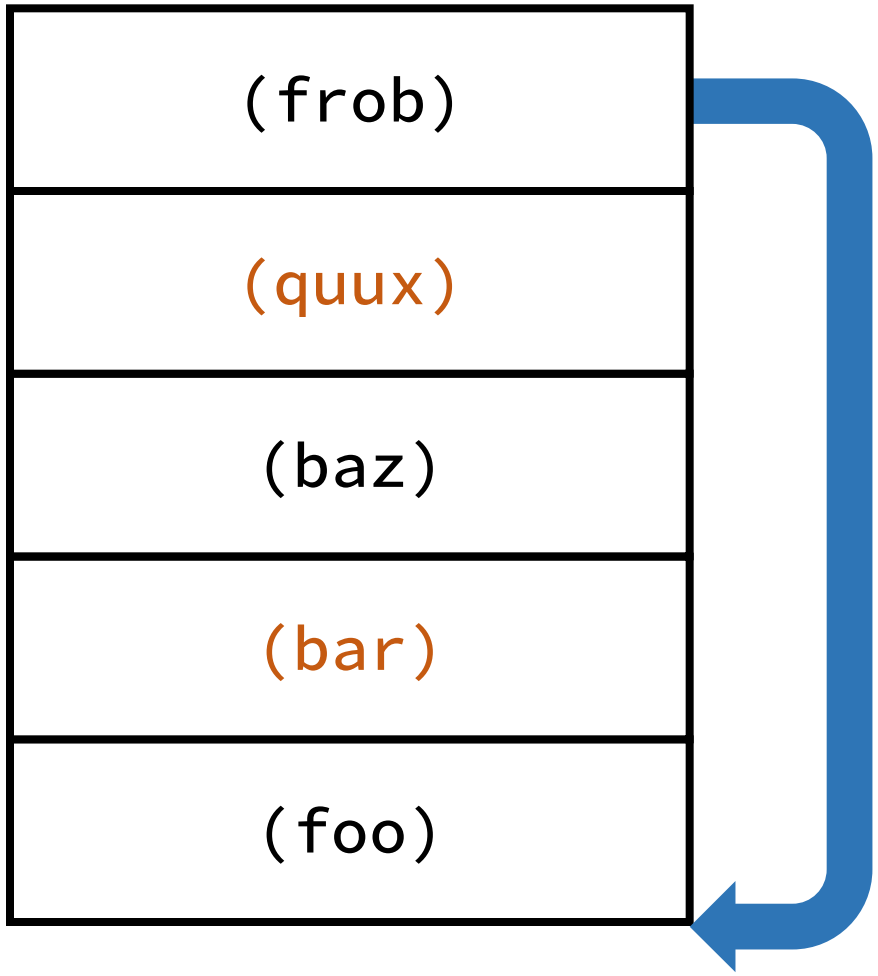
- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

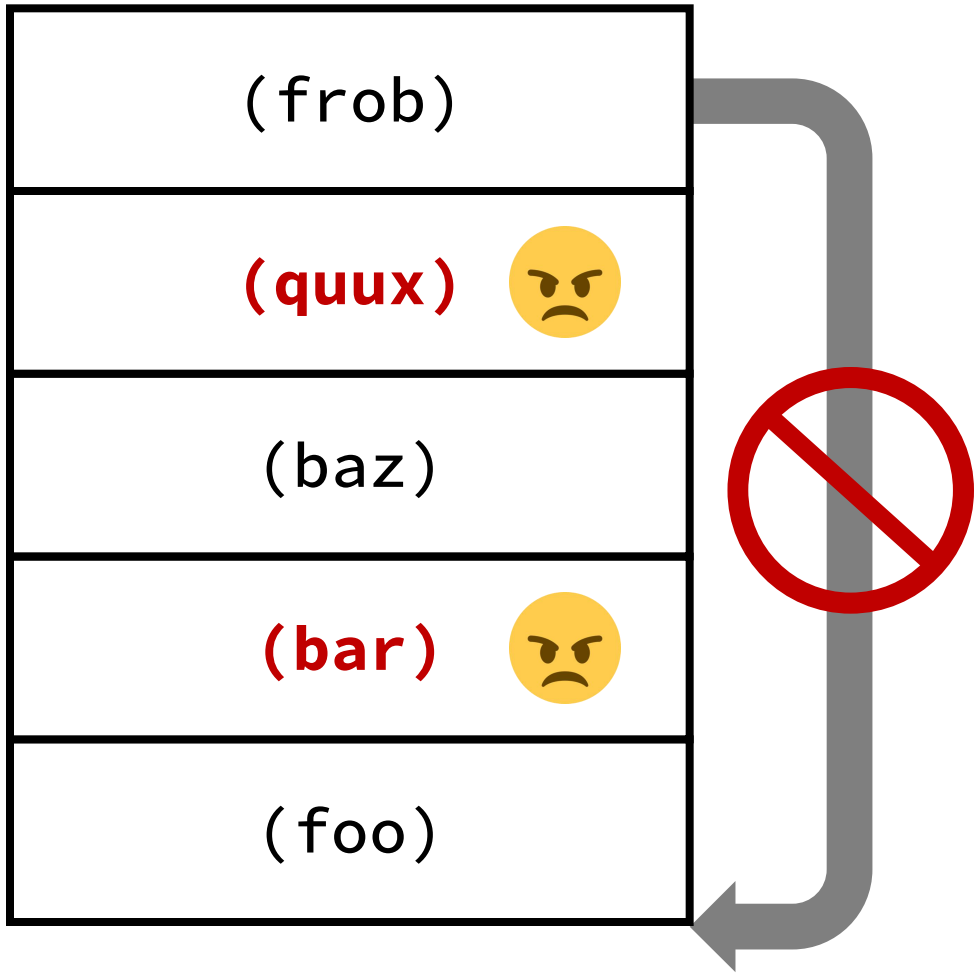
Let's talk about unwinding in Common Lisp

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

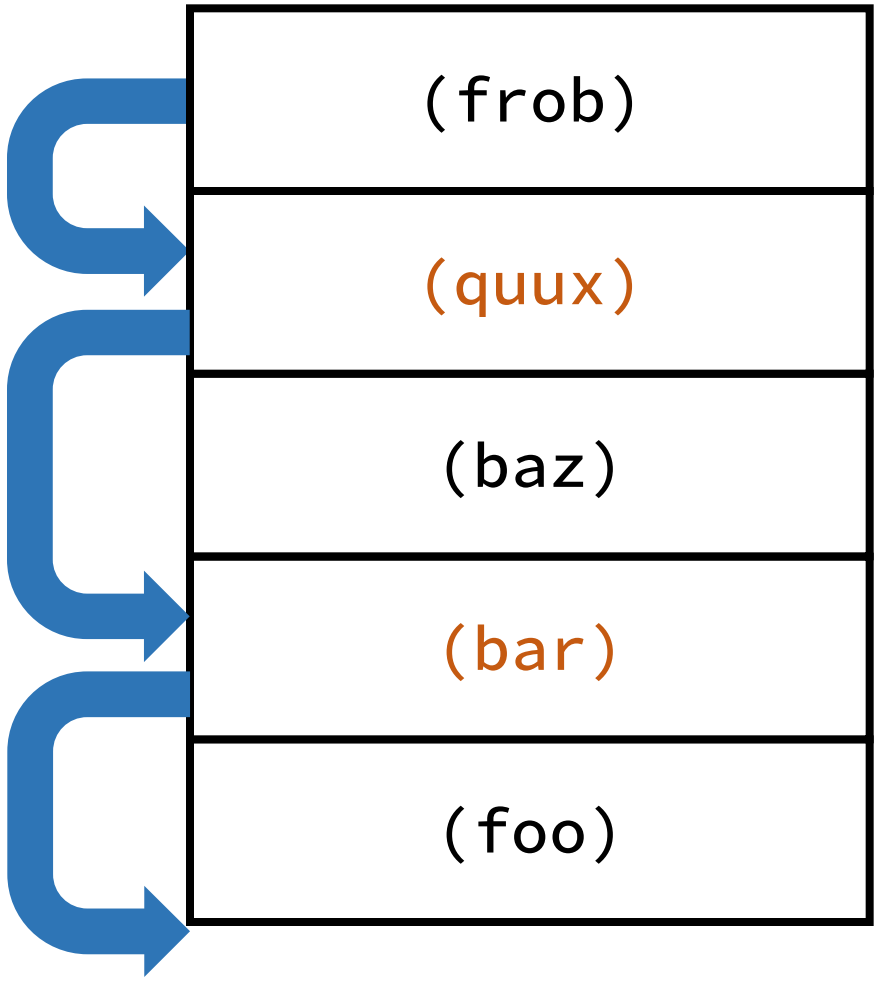


Let’s talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

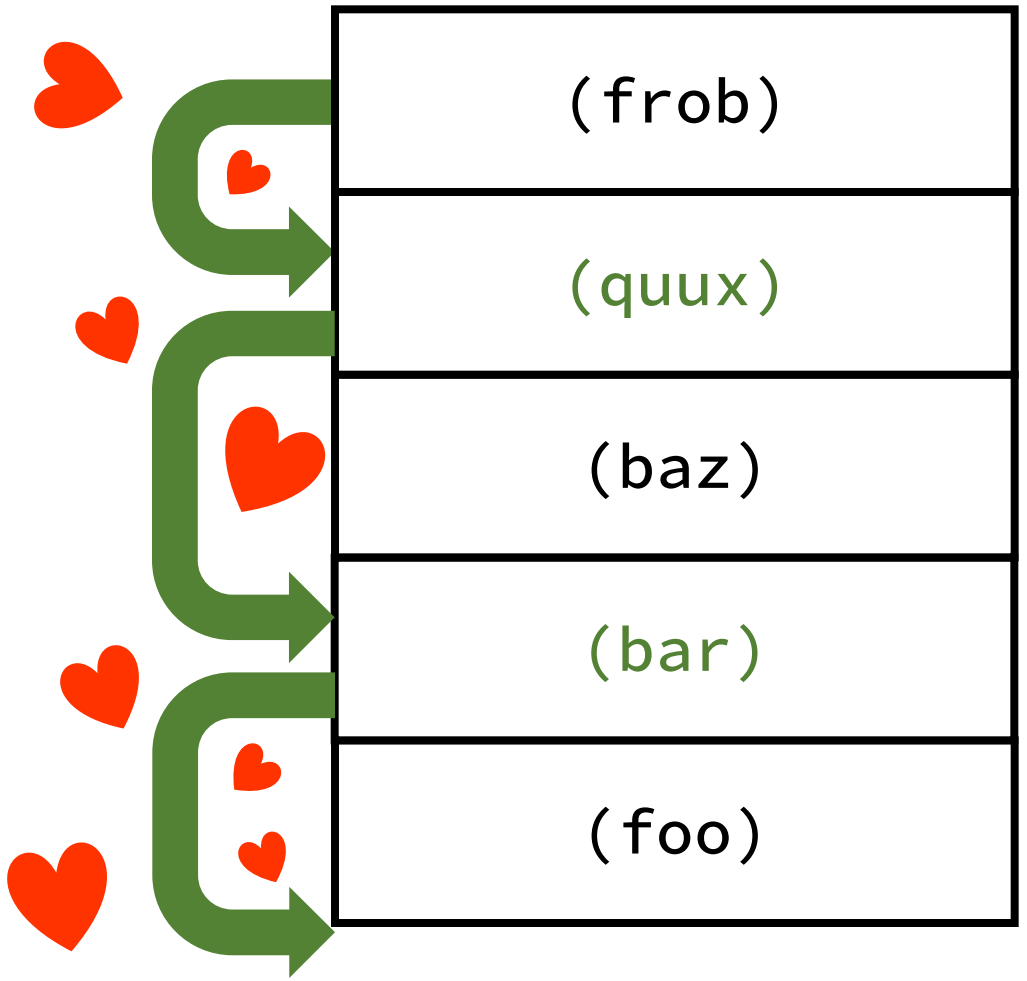
Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



Let's talk about unwinding in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`



Control Flow in Common Lisp

aka Why Lisp Doesn't Need To Throw Exceptions

Appendix D

Common Lisp condition system without Common Lisp

(this is the last one I promise)

Let’s talk about control flow in Common Lisp

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

Let’s talk about control flow in Common Lisp and Java

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `try/finally`
- `new/.apply()`
- `throw exception`

Let’s talk about control flow in Common Lisp and Java

Metacircular Semantics for Common Lisp Special Forms

Henry G. Baker
Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

McCarthy's metacircular interpreter for Lisp has been criticized by Reynolds and others for not providing precise semantics. Unfortunately, the alternative of English prose currently favored by the ANSI X3J13 and ISO committees for the definition of Common Lisp is even less precise than a metacircular interpreter. Thus, while a system of denotational semantics á la Scheme or ML could be developed for Common Lisp, we believe that a carefully fashioned system of metacircular definitions can achieve most of the precision of denotational semantics. Furthermore, a metacircular definition is also more readable and understandable by the average Common Lisp programmer, since it is written in terms he mostly understands. Finally, a metacircular definition for Common Lisp special forms enables us to transparently customize the representation of certain "built-in" mechanisms such as `function` closures, to enable sophisticated systems like "Portable Common Loops" to become truly portable.

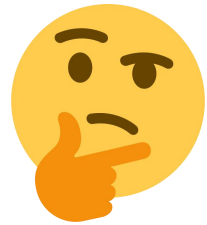
Let’s talk about control flow in Common Lisp and Java

Metacircular Semantics for Common Lisp Special Forms

Henry G. Baker
Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

McCarthy's metacircular interpreter for Lisp has been criticized by Reynolds and others for not providing precise semantics. Unfortunately, the alternative of English prose currently favored by the ANSI X3J13 and ISO committees for the definition of Common Lisp is even less precise than a metacircular interpreter. Thus, while a system of denotational semantics á la Scheme or ML could be developed for Common Lisp, we believe that a carefully fashioned system of metacircular definitions can achieve most of the precision of denotational semantics. Furthermore, a metacircular definition is also more readable and understandable by the

Finally, a metacircular definition for Common Lisp special forms enables us to transparently customize the representation of certain "built-in" mechanisms such as `function` closures, to enable sophisticated systems like "Portable Common Loops" to become truly portable.



Let’s talk about control flow in Common Lisp and Java

Metacircular Semantics for Common Lisp Special Forms

Henry G. Baker
Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

McCarthy's metacircular interpreter for Lisp has been criticized by Reynolds and others for not providing precise semantics. Unfortunately, the alternative of English prose currently favored by the ANSI X3J13 and ISO committees for the definition of Common Lisp is even less precise than a metacircular interpreter. Thus, while a system of denotational semantics á la Scheme or VDL could be developed for Common Lisp, we believe that a carefully fashioned system of metacircular definitions can achieve most of the precision of denotational semantics. Furthermore, a metacircular definition is also more readable and understandable by the

Finally, a metacircular definition for Common Lisp special forms enables us to enable sophisticated systems like "Portable Common Loops" to become truly portable.

Let’s talk about control flow in Common Lisp and Java

Metacircular Semantics for Common Lisp Special Forms

Henry G. Baker
Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

McCarthy's metacircular interpreter for Lisp has been criticized by Reynolds and others for not providing precise semantics. Unfortunately, the alternative of English prose currently favored by the ANSI X3J13 and ISO committees for the definition of Common Lisp is even less precise than a metacircular interpreter. Thus, while a system of denotational semantics á la Scheme or ISWIM could be developed for Common Lisp, we believe that a carefully fashioned system of metacircular definitions can achieve most of the precision of denotational semantics. Furthermore, a metacircular definition is also more readable and understandable by the

Finally, a metacircular definition for Common Lisp special forms enables us to enable sophisticated systems like "Portable Common Loops" to become truly portable.

- Can we port the condition system to Java?

Let’s talk about control flow in Common Lisp and Java

Metacircular Semantics for Common Lisp Special Forms

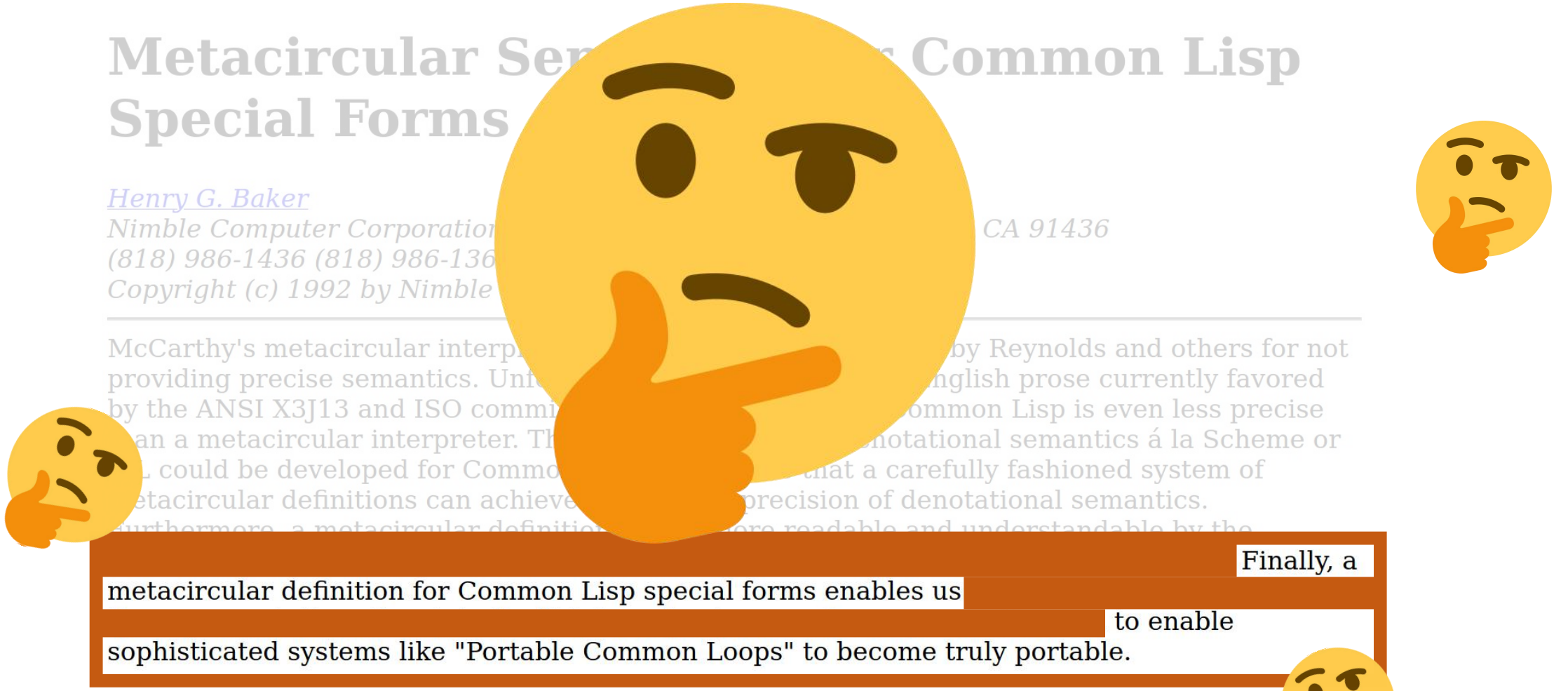
Henry G. Baker
Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

McCarthy's metacircular interpreter for Lisp has been criticized by Reynolds and others for not providing precise semantics. Unfortunately, the alternative of English prose currently favored by the ANSI X3J13 and ISO committees for the definition of Common Lisp is even less precise an a metacircular interpreter. Thus, while a system of denotational semantics á la Scheme or L could be developed for Common Lisp, we believe that a carefully fashioned system of etacircular definitions can achieve most of the precision of denotational semantics. Furthermore, a metacircular definition is also more readable and understandable by the

Finally, a metacircular definition for Common Lisp special forms enables us to enable sophisticated systems like "Portable Common Loops" to become truly portable.

- Can we port CL control flow to Java?

Let’s talk about Common Lisp control flow in Java



- Can we port CL control flow to Java?

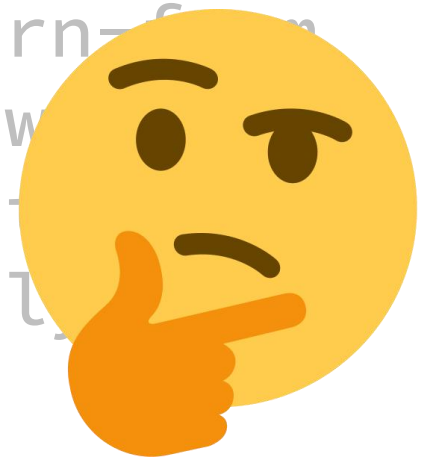
Let’s talk about Common Lisp control flow in Java

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-protect`
- `lambda/apply`

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `try/finally`
- `new/.apply()`
- `throw exception`

Let's talk about Common Lisp control flow in Java

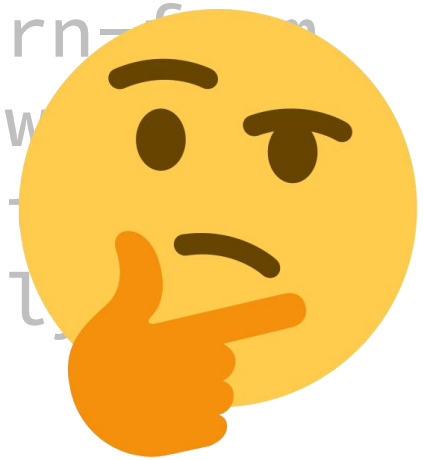
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-prot
- lambda/appl



- if
- tagbody/go
- block/return-from
- catch/throw
- try/finally
- new/.apply()
- throw exception

Let's talk about Common Lisp control flow in Java

- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `unwind-prot`
- `lambda/appl`



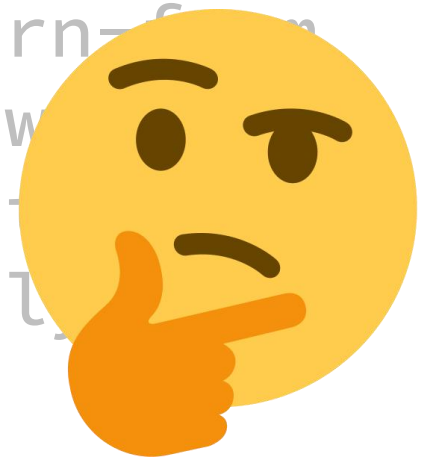
- `if`
- `tagbody/go`
- `block/return-from`
- `catch/throw`
- `try/finally`
- `new/.apply()`
- `throw exception`

"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

--- Abraham H. Maslow

Let's talk about Common Lisp control flow in Java

- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-prot
- lambda/appl



- if
- tagbody/go
- block/return-from
- catch/throw
- try/finally
- new/.apply()
- throw exception

https://github.com/phoe/cafe-latte

<3

<3

yes, it’s seriously the end this time