# **Control Flow in Common Lisp**

aka Why Lisp Doesn't Need To Throw Exceptions

Michał "phoe" Herda



phoe@disroot.org

https://phoe.github.io

We need means of unwinding and protecting against unwinds

We need means of unwinding and protecting against unwinds

- We need means of unwinding and protecting against unwinds
  - We can implement all of CL-specific control flow with this!

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

- We need means of unwinding and protecting against unwinds
  - 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)

- We need means of unwinding and protecting against unwinds
  - 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

- We need means of unwinding and protecting against unwinds
  - 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

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

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

• Common Lisp enthusiast (#lisp, Lisp Discord, Reddit)

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

- Common Lisp enthusiast (#lisp, Lisp Discord, Reddit)
  - Author of *The Common Lisp Condition System* (Apress, 2020)

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

- 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

• I kinda like the topic of control flow

### Let's talk about control flow

• I kinda like the topic of control flow

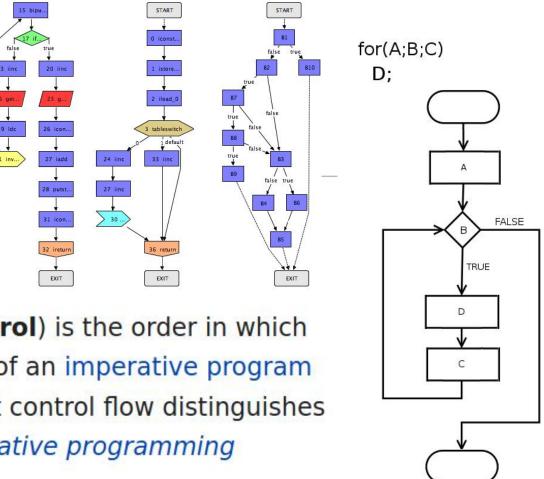
#### Let's talk about control flow

### 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.



#### **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.

#### **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.

baz()

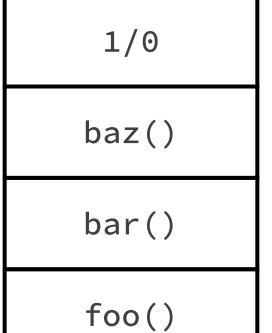
bar()

foo()

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



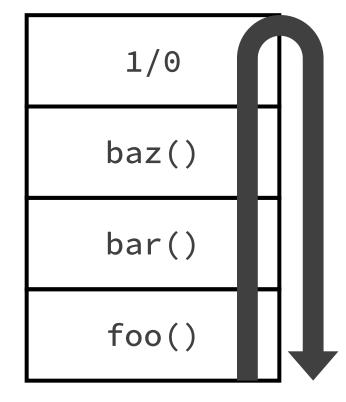
#### ArithmeticException







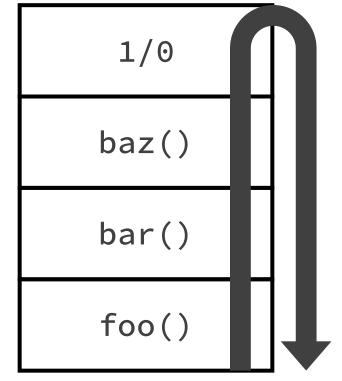
ArithmeticException







ArithmeticException







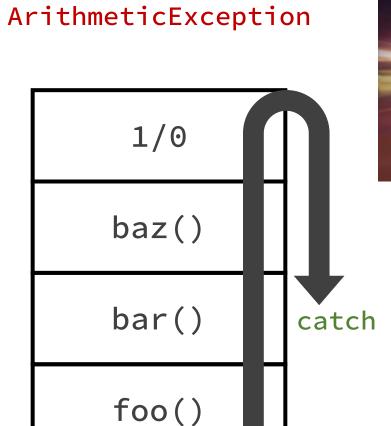


#### ArithmeticException

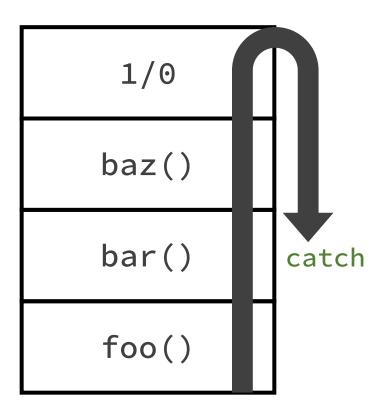






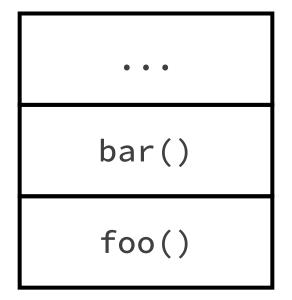






bar()

foo()



// execution continues

#### Let's talk about conditions

#### Condition systems [edit]

Common Lisp, Dylan and Smalltalk have a condition system<sup>[53]</sup> (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.<sup>[54]</sup>

#### Let's talk about conditions

#### Condition systems [edit]

Common Lisp, Dylan and Smalltalk have a condition system<sup>[53]</sup> (see Common Lisp Condition System) that encompasses the ementioned exception handling systems. In those languages or environments the advent of a configuration of an error according to Kent Pitman) implies a function call, and only late in the experior 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<sup>[53]</sup> (see Common Lisp Condition System) that encompasses the ementioned exception handling systems. In those languages or environments the advent of a configuration of an error according to Kent Pitman) implies a function call, and only late in the experior 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]

(baz)

(bar)

(foo)

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

division-by-zero (/ 1 0)
(baz)
(bar)
(foo)



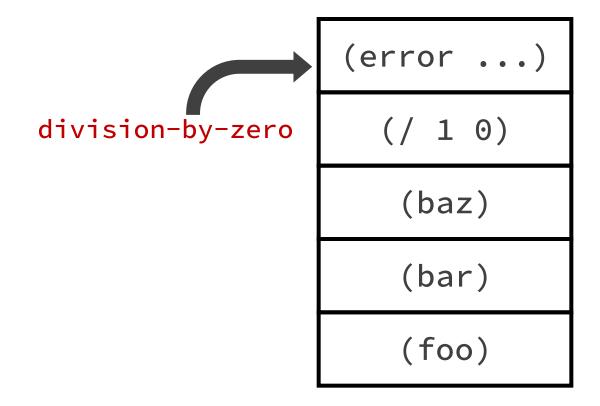


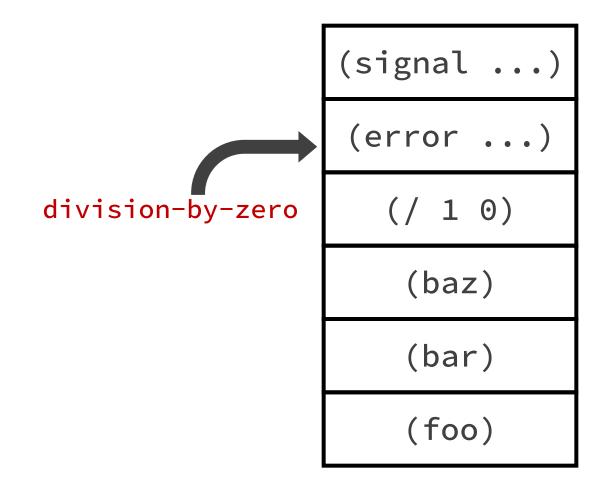


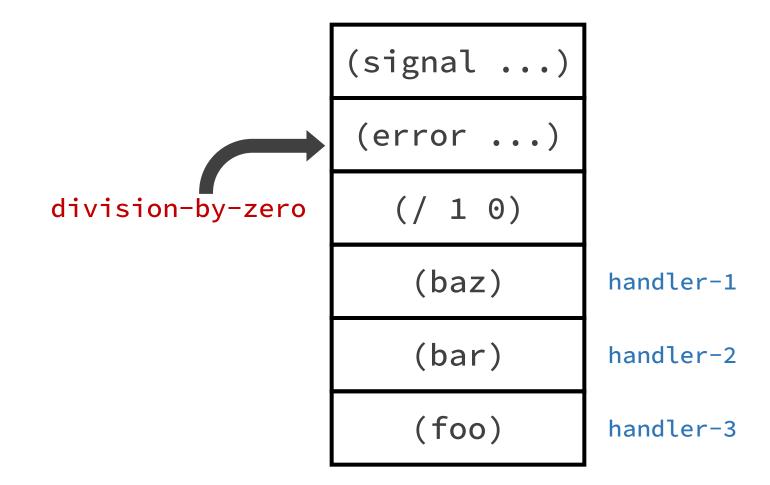


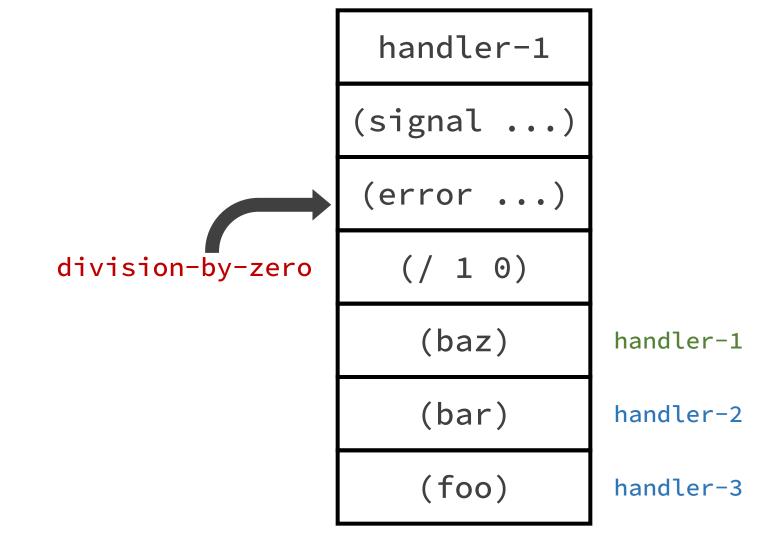


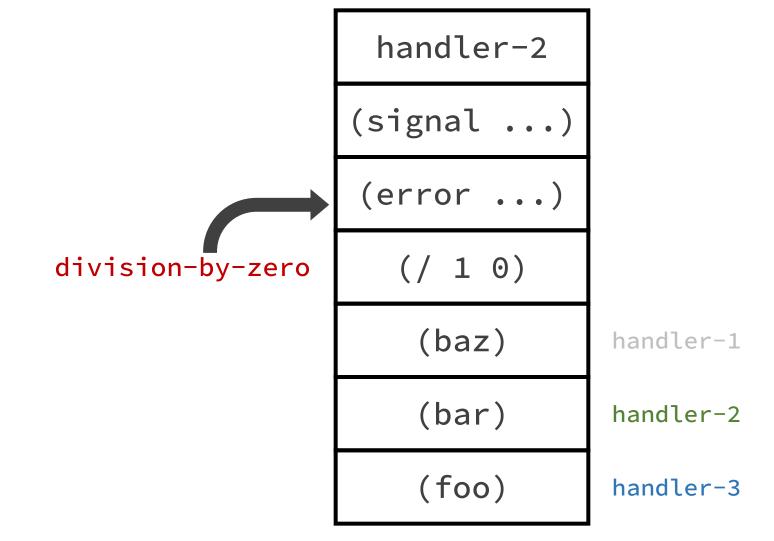


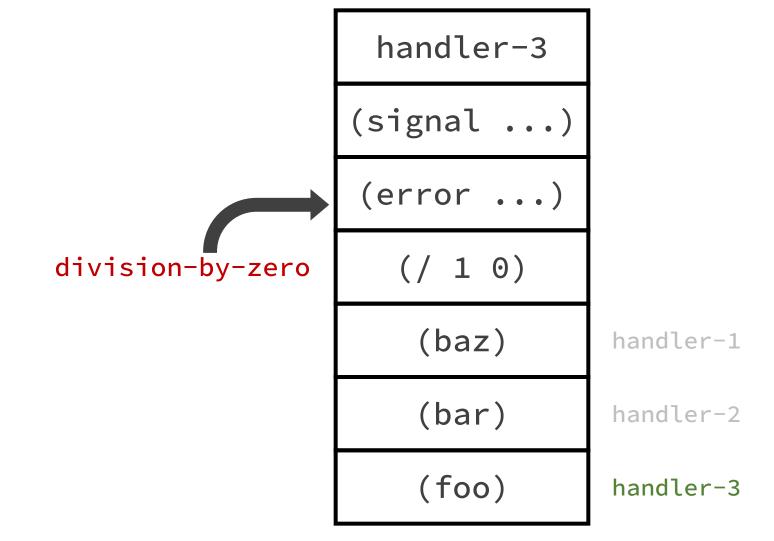


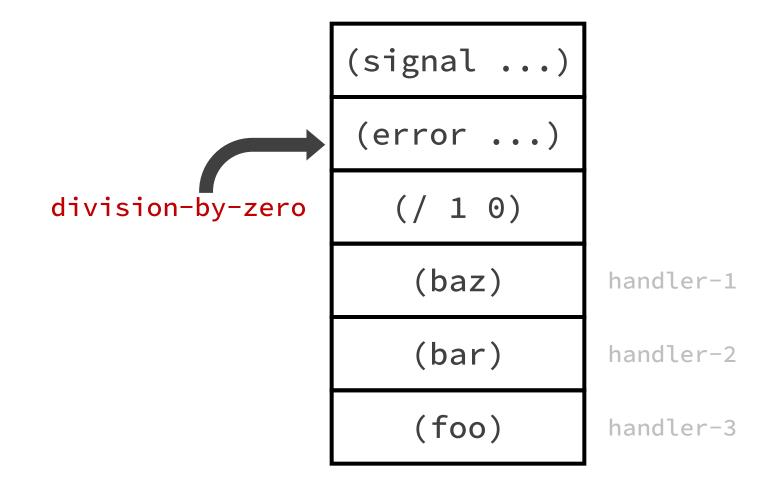


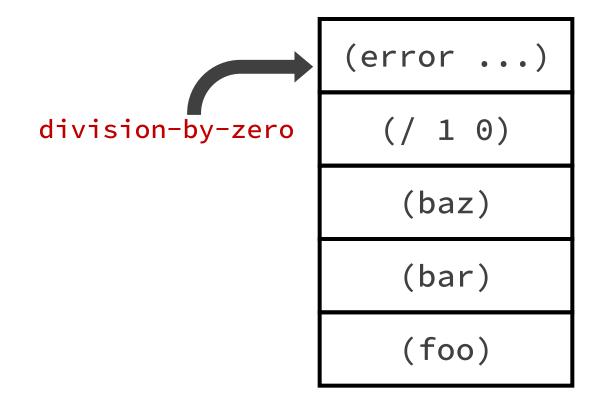


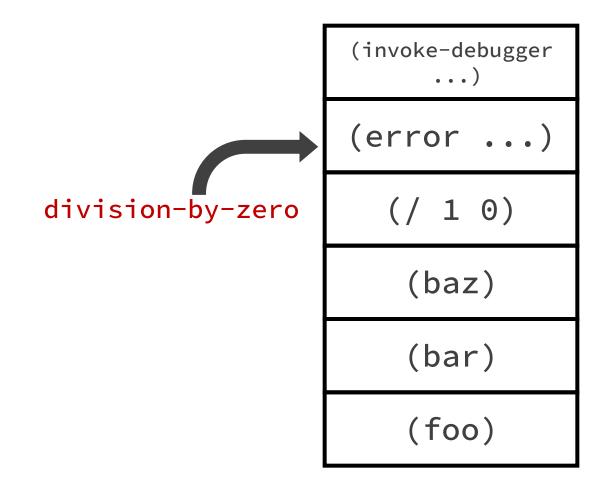


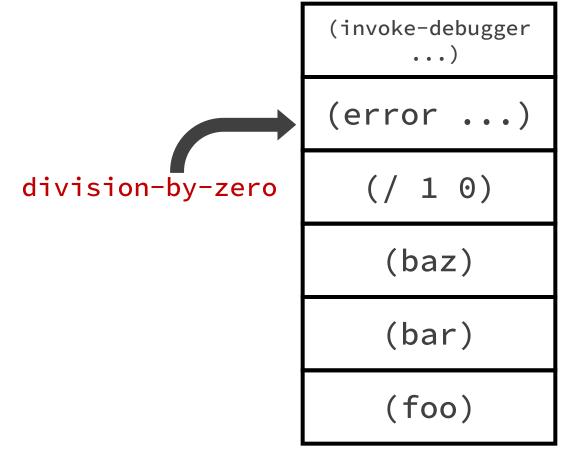




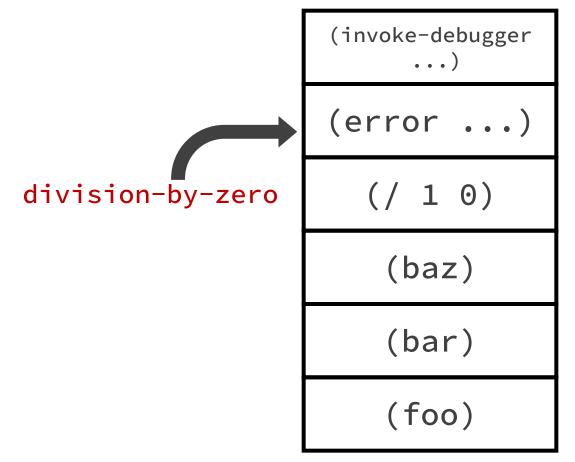




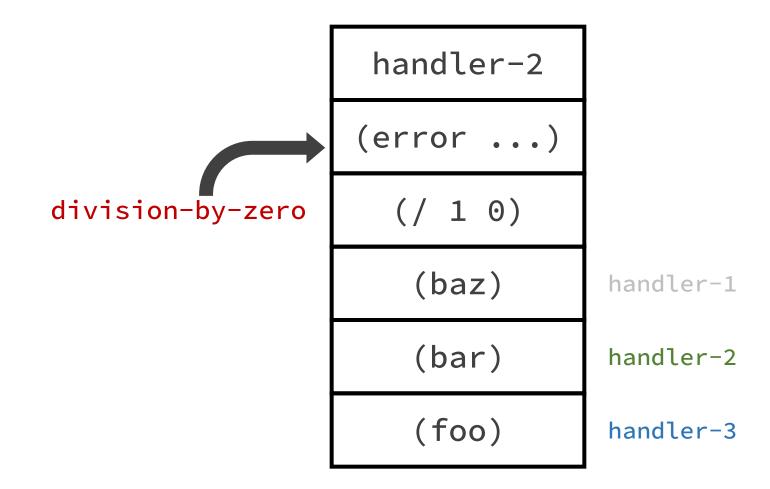


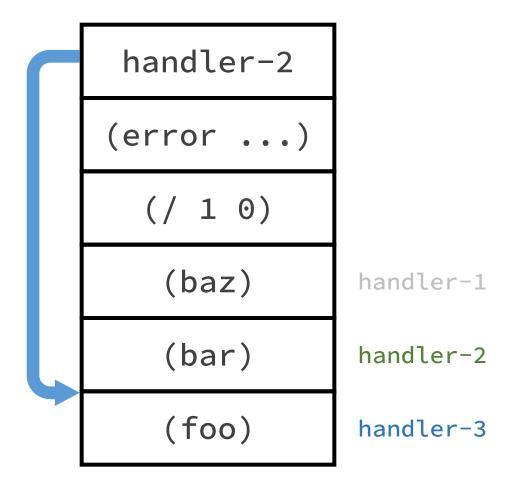


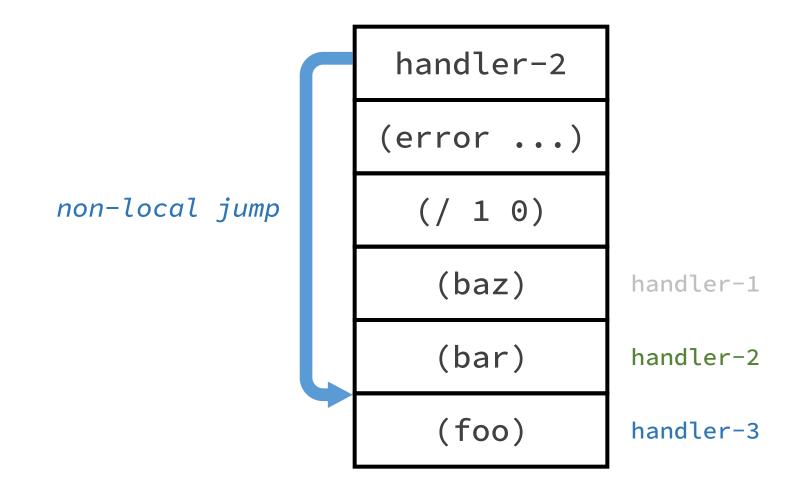




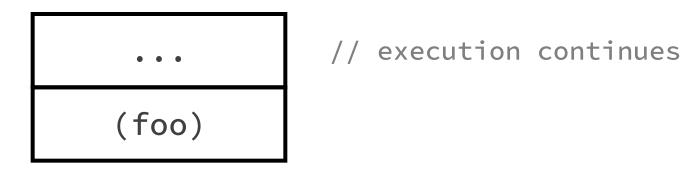


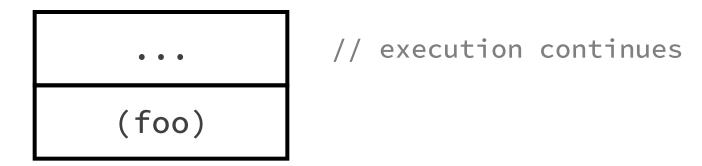


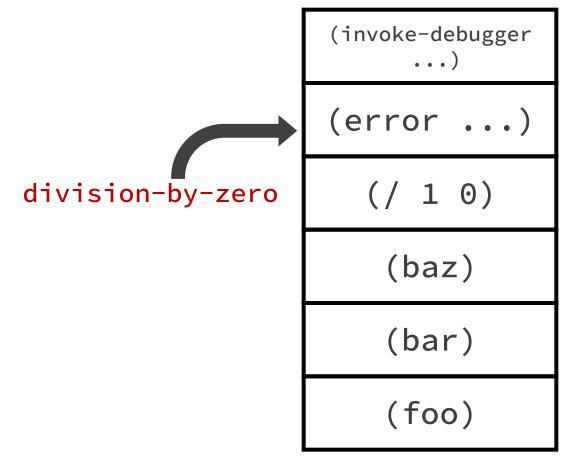




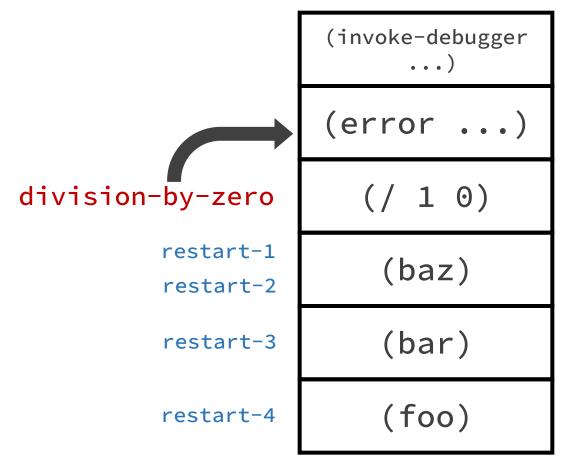
(foo)













division-by-zero "Return 42 instead." "Query the user for new numbers." "Try opening another file." "Abort and return to toplevel."

(invoke-debugger ...) (error ...) (/10)(baz) (bar) (foo)





"Return 42 instead."

"Query the user for new numbers."

"Try opening another file."

"Abort and return to toplevel."



(invoke-debugger
...)

(error ...)

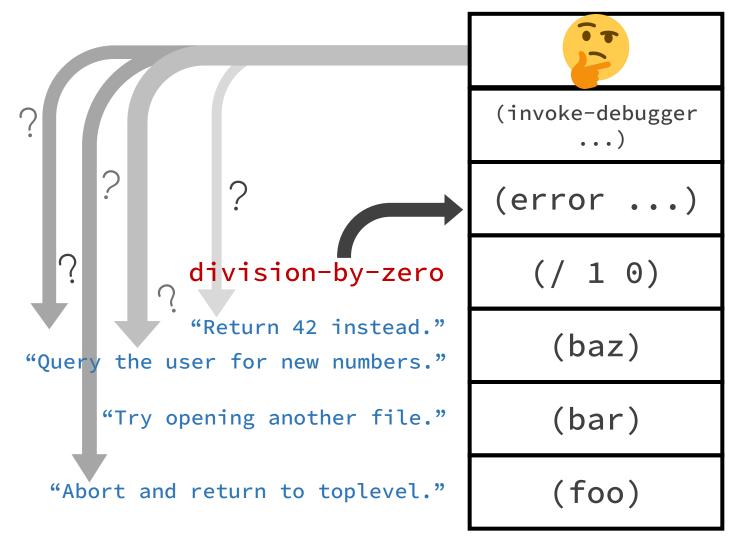
(/ 1 0)

(baz)

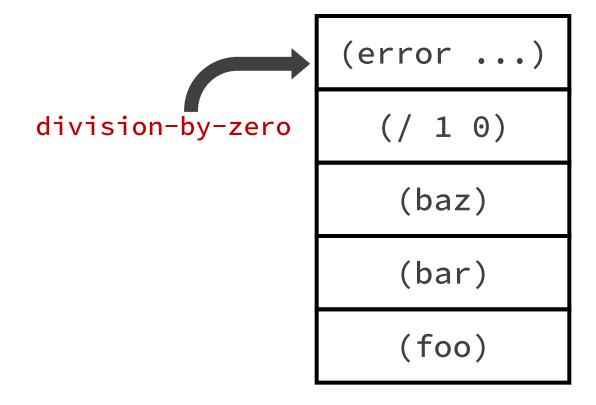
(bar)

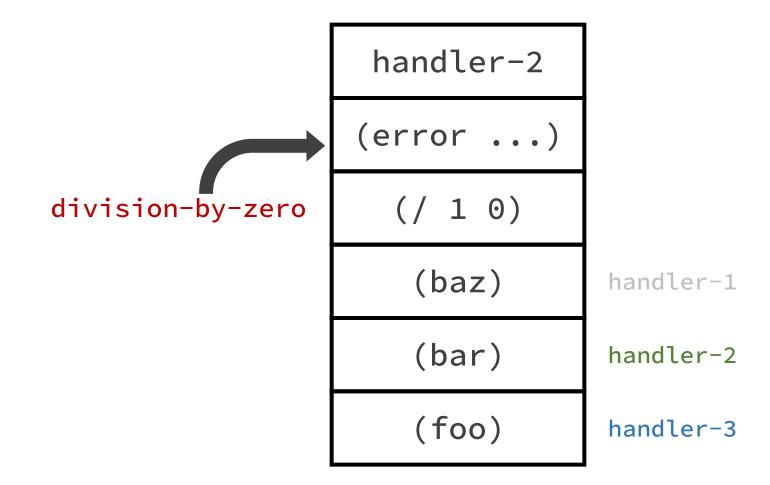
(foo)

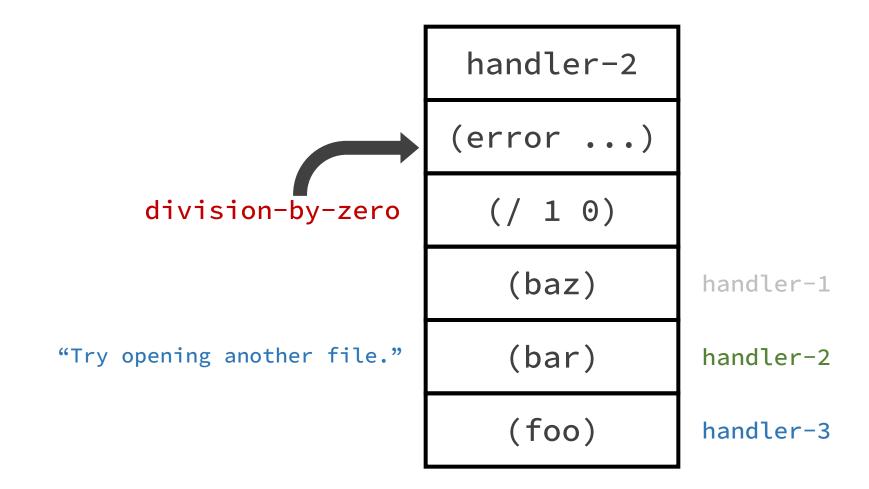


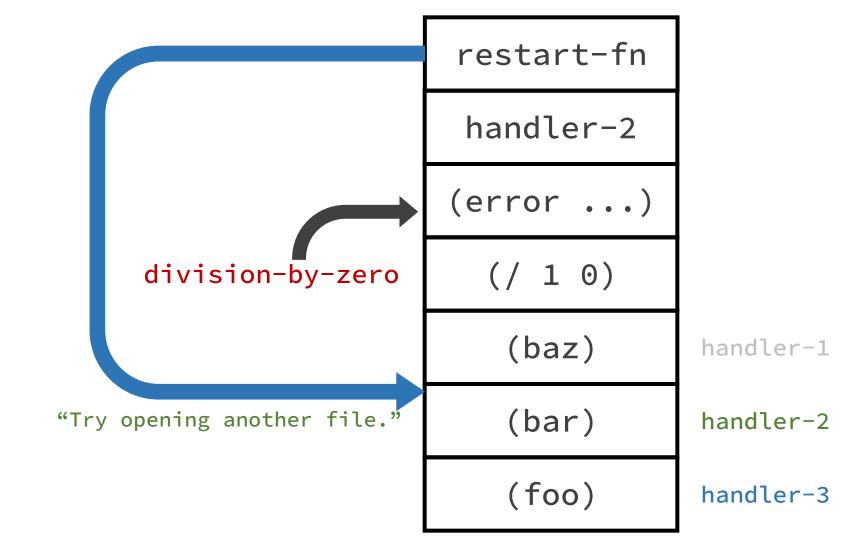


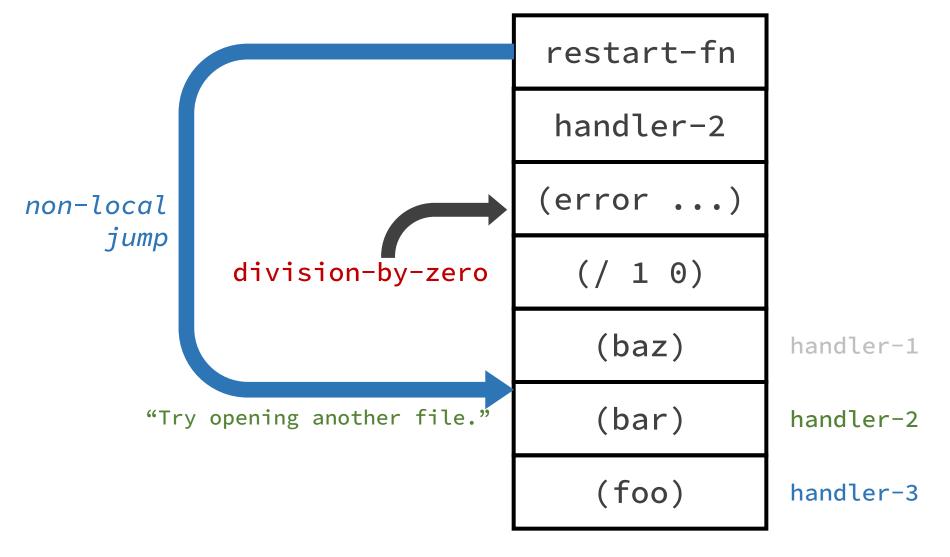






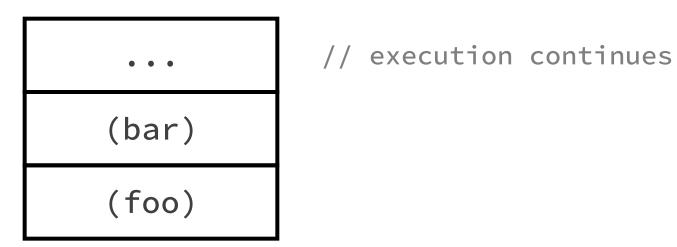






(bar)

(foo)



(quux)
(baz)
(bar)
(foo)

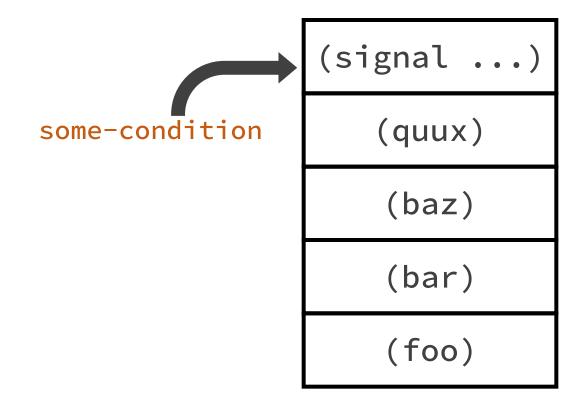
some-condition

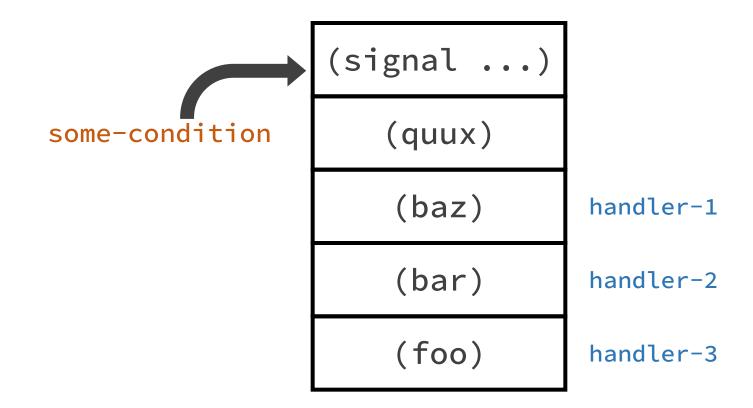
(quux)

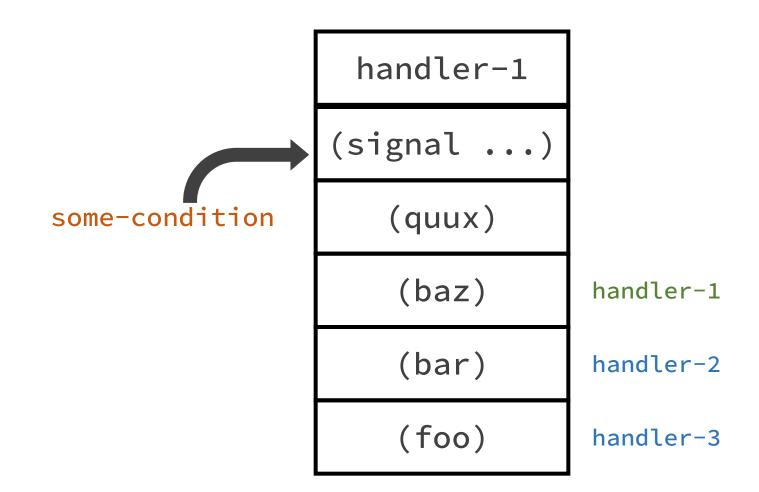
(baz)

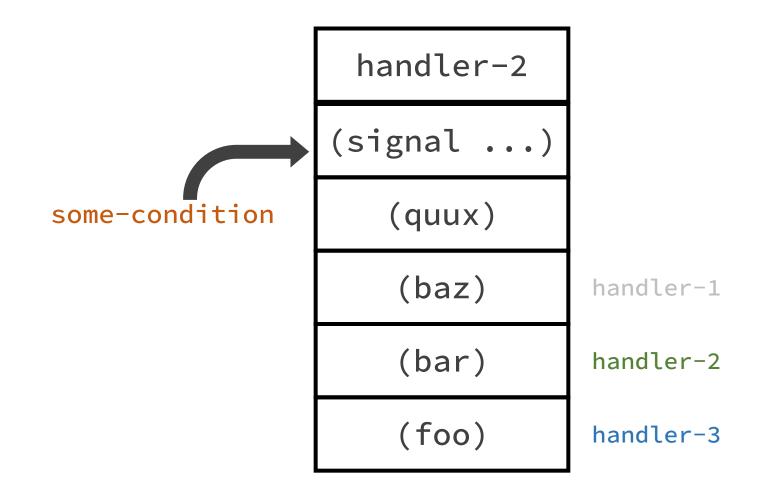
(bar)

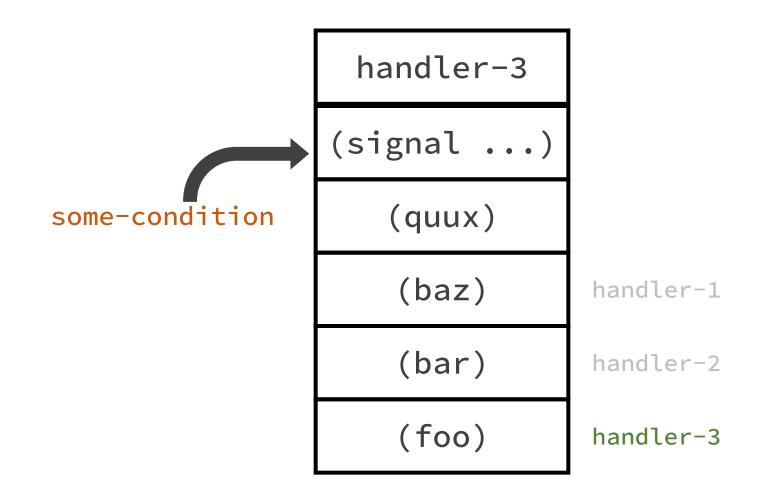
(foo)

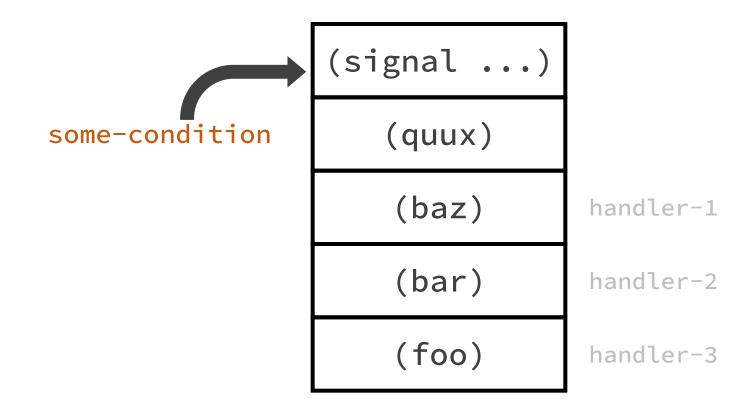


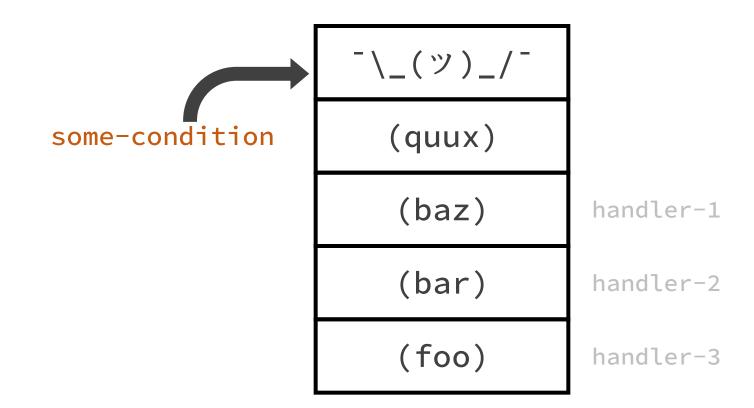












(quux)
(baz)
(bar)
(foo)



// execution continues

Construct the exception object

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

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

Construct the condition object

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

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

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



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



- 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



- 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



- 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



- 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



- 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™



- 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



- 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



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

- 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™
- 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

#### 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

- 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..

S

77

# 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

#### 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



- 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..

- 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..

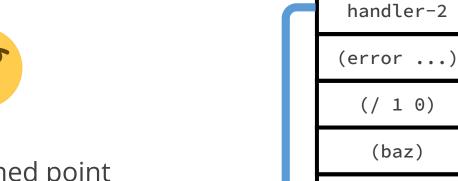
#### Let's talk about non-local control flow

- 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

- 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 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

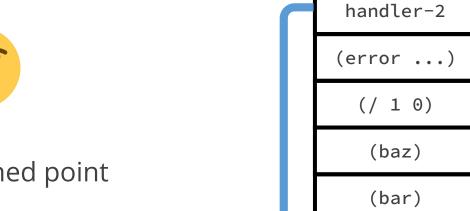


(bar)

(foo)

#### Let's talk about control flow

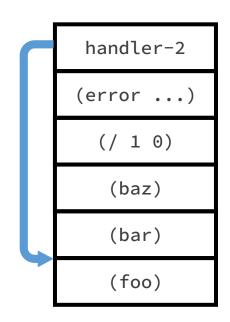
- Construct the condition object
- Call handlers from the stack in order
  - What do the handlers do?
    - Maybe execute some code
    - Maybe invoke a restart
    - Maybe do nothing and return
    - Maybe unwind the stack to a predefined point
    - Maybe there are no handlers
    - Maybe she's born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



(foo)

- 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





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

(baz))

• if

```
iftagbody/go
```

```
(tagbody
10 (print "hello")
20 (go 10))
```

```
• if

    tagbody/go

• block/return-from
(block my-block
  (\ldots)
  (... (return-from my-block 42))
  (\ldots)
```

```
• if
tagbody/go
block/return-from
catch/throw
(catch 'quux
  (\ldots)
  (... (foo))
                              (defun foo ()
                                (throw 'quux 42))
  (\ldots))
```

```
• if
tagbody/go
block/return-from
catch/throw
• unwind-protect
(let ((thing (make-thing)))
  (unwind-protect (frob thing)
    (cleanup thing)))
```

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

    lambda/apply

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

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

    lambda/apply; and funcall

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

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

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

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

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

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

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

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

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

```
(let ((x 42))
  (lambda () x))
;; #<FUNCTION (LAMBDA ())>
(funcall *); (funcall #<FUNCTION (LAMBDA ())>)
;; => 42
;; but we can close over more
;; than just lexical variables!
```

```
(let ((x 42))
  (lambda () x)
;; #<FUNCTION (LAMBDA ())>
(funcall *); (funcall #<FUNCTION (LAMBDA ())>)
;; => 42
;; but we can close over more
;; than just lexical variables!
```

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

```
(defun foo (x) (funcall x))
(defun bar (); block bar
...)
```

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

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

(bar)

```
(foo)
(bar)
```

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

```
(lambda () ...)

(foo)

(bar)
```

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

```
(lambda () ...)

(foo)

(bar)
```

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

```
(lambda () ...)
(foo)
(bar)
```

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

```
(defun foo (x) (funcall x))
(defun bar ()
  (let ((fn (lambda ()
               (return-from
                bar 42))))
    (foo fn)))
(bar)
;; => 42
```

- 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
```

- 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

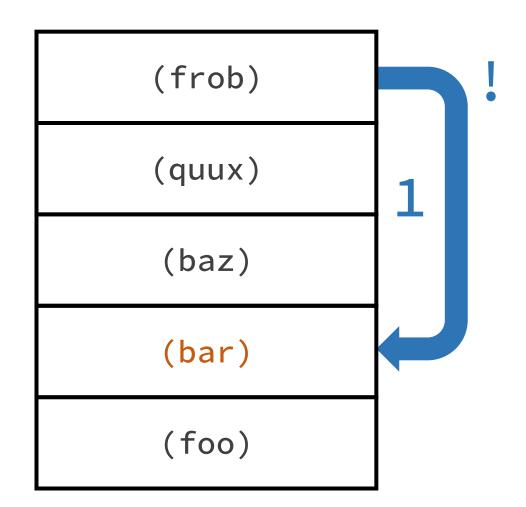
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



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



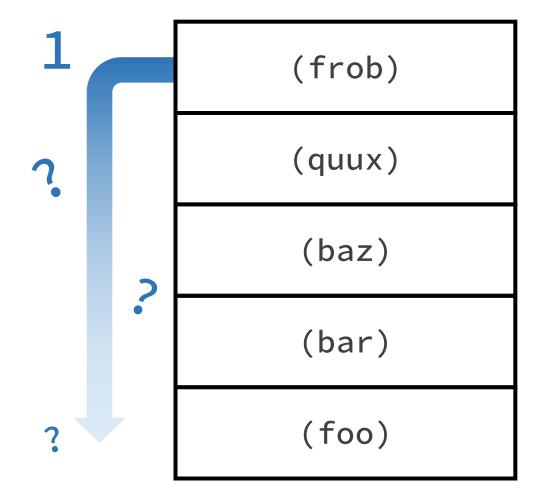
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



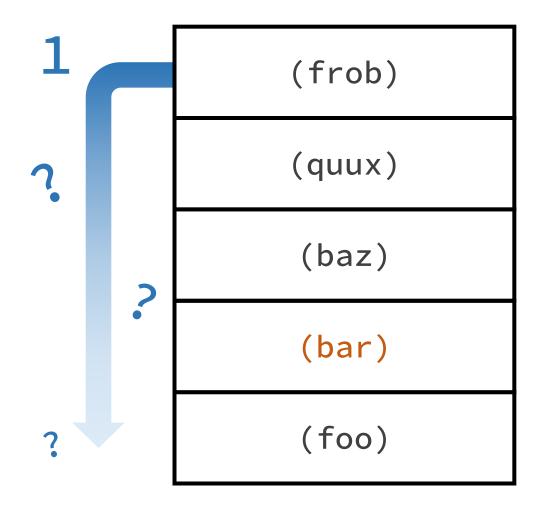
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



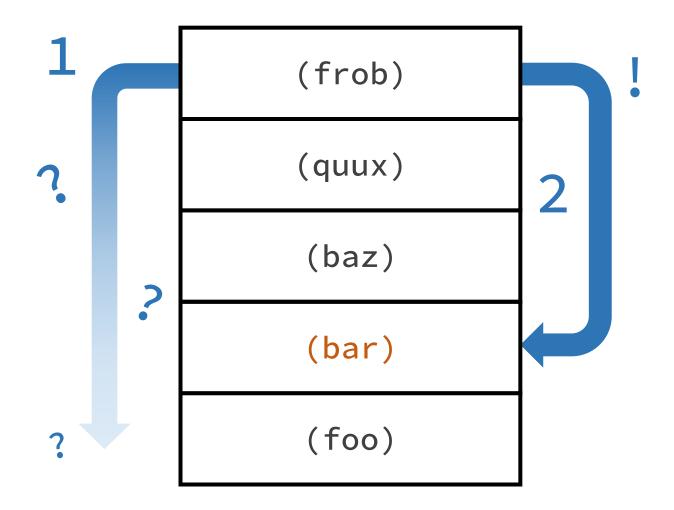
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



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



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



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

- 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)
```

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

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

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

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

- 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, ...)

- 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, ...)

- 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, ...)

- 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, ...)

- 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

- 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

- 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

Do not conflate unwinding with throwing exceptions

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

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

- 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

- 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





but wait hold on for just one moment

# **Control Flow in Common Lisp**

aka Why Lisp Doesn't Need To Throw Exceptions

**Appendix A** 

**Proving one-phase unwind in TAGBODY and BLOCK** 

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

- if
- tagbody/go
- catch/throw
- unwind-protect
- lambda/apply

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

    block/return-from ; 1-phase unwind (no search)

                     ; 2-phase unwind (search)
```

- if
- tagbody/go
- catch/throw
- unwind-protect
- lambda/apply

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

    block/return-from ; 1-phase unwind (no search)

                     ; 2-phase unwind (search)
```

```
• if
tagbody/go
                     ; 1-phase unwind (no search)

    block/return-from ; 1-phase unwind (no search)

                     ; 2-phase unwind (search)
catch/throw
• unwind-protect
                 (block foo
• lambda/apply
                   (lambda ()
                      (return-from foo)))
```

```
• if
tagbody/go
                    ; 1-phase unwind (no search)
block/return-from ; 1-phase unwind (no search)
                ; 2-phase unwind (search)
catch/throw
unwind-protect
                (let ((fn (block foo
• lambda/apply
                            (lambda ()
                              (return-from foo)))))
                  (funcall fn))
                ; ...?
```

```
• if
tagbody/go
                     ; 1-phase unwind (no search)

    block/return-from ; 1-phase unwind (no search)

                     ; 2-phase unwind (search)
catch/throw
unwind-protect
                 (let ((fn (block foo
• lambda/apply
                             (lambda ()
                               (return-from foo)))))
                   (funcall fn))
                 ; ERROR: Condition CONTROL-ERROR
                 ; was signaled.
                                                   177
```

```
• if
tagbody/go
                     ; 1-phase unwind (no search)

    block/return-from ; 1-phase unwind (no search)

                     ; 2-phase unwind (search)
catch/throw
unwind-protect
                 (let ((fn (block foo
• lambda/apply
                             (lambda ()
                               (return-from foo)))))
                   (funcall fn))
                 ; ERROR: Condition CONTROL-ERROR
                 ; was signaled.
                                                   178
```

```
• if
                     ; 1-phase unwind (no search)
tagbody/go

    block/return-from ; 1-phase unwind (no search)

                     ; 2-phase unwind (search)
catch/throw
• unwind-protect
• lambda/apply
                                curn-from foo)))))
                   (funcall fn))
                  ERROR: Condition CONTROL-ERROR
                  was signaled.
```

179

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

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

```
(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 ((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 B** 

**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

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

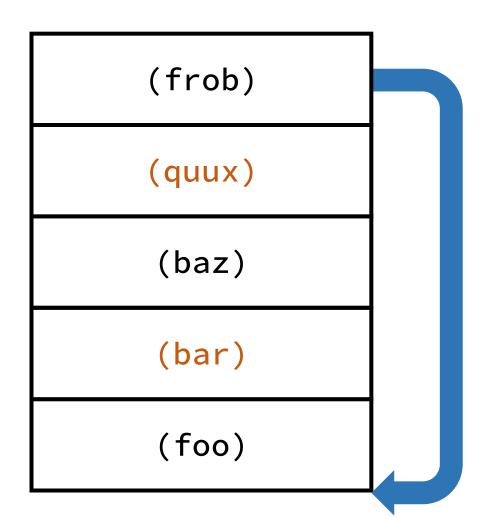
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



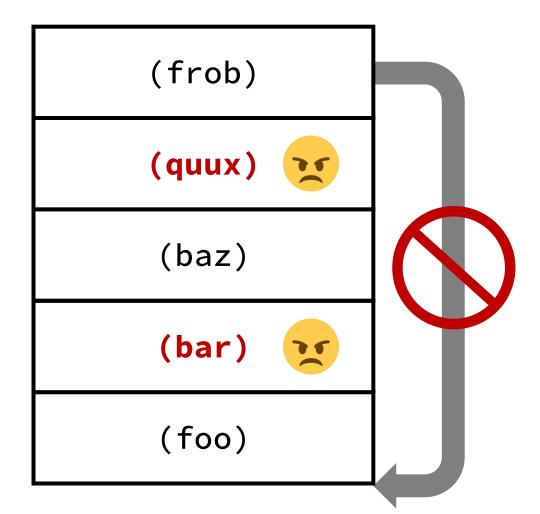
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



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



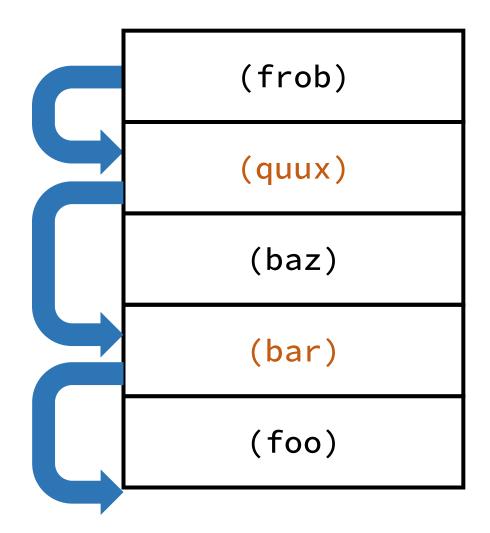
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



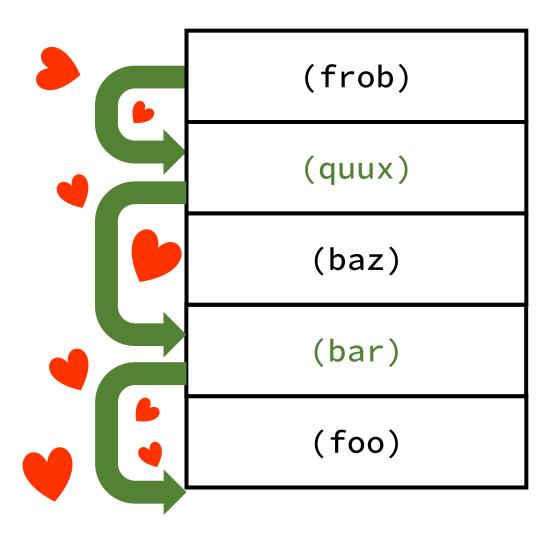
- if
- tagbody/go
- block/return-from
- catch/throw
- unwind-protect
- lambda/apply



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



- 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 C** 

**Common Lisp condition system without Common Lisp** 

(this is the last one I promise)

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

- 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

### 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)
Copyright (c) 1992 by Nimble Computer Corporation

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.

### 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)
Copyright (c) 1992 by Nimble Computer Corporation



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

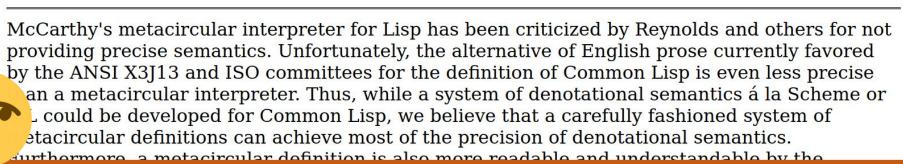
Finally, a

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.

### 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)
Copyright (c) 1992 by Nimble Computer Corporation



metacircular definition for Common Lisp special forms enables us

to enable

Finally, a

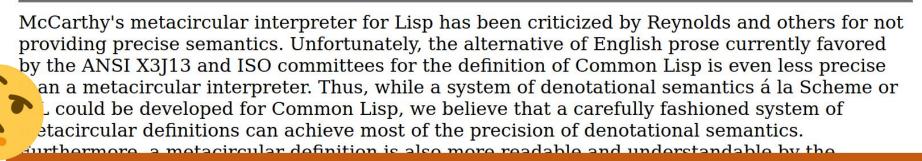
sophisticated systems like "Portable Common Loops" to become truly portable.



### 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)
Copyright (c) 1992 by Nimble Computer Corporation



metacircular definition for Common Lisp special forms enables us

to enable

Finally, a

sophisticated systems like "Portable Common Loops" to become truly portable.

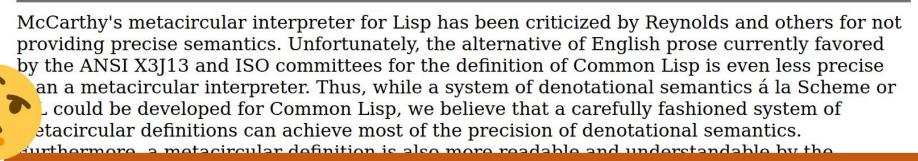
Can we port the condition system to Java?



### 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)
Copyright (c) 1992 by Nimble Computer Corporation



metacircular definition for Common Lisp special forms enables us

to enable

Finally, a

sophisticated systems like "Portable Common Loops" to become truly portable.

Can we port CL control flow to Java?



### Metacircular Ser Special Forms

#### Henry G. Baker

Nimble Computer Corporation (818) 986-1436 (818) 986-136 Copyright (c) 1992 by Nimble

McCarthy's metacircular interplant providing precise semantics. Unflate the ANSI X3J13 and ISO committee and a metacircular interpreter. The could be developed for Commo etacircular definitions can achieve turthermore, a metacircular definition.

**Common Lisp** 

CA 91436

by Reynolds and others for not a nglish prose currently favored ommon Lisp is even less precise notational semantics á la Scheme or that a carefully fashioned system of precision of denotational semantics.

Finally, a

metacircular definition for Common Lisp special forms enables us

to enable

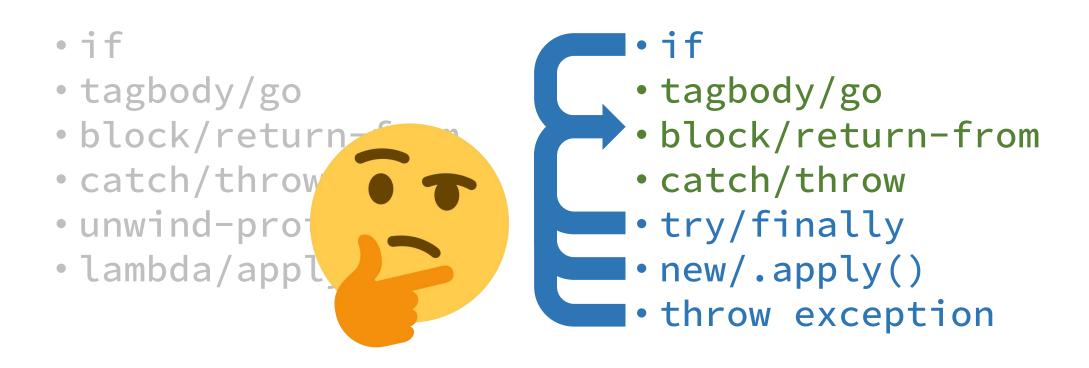
sophisticated systems like "Portable Common Loops" to become truly portable.

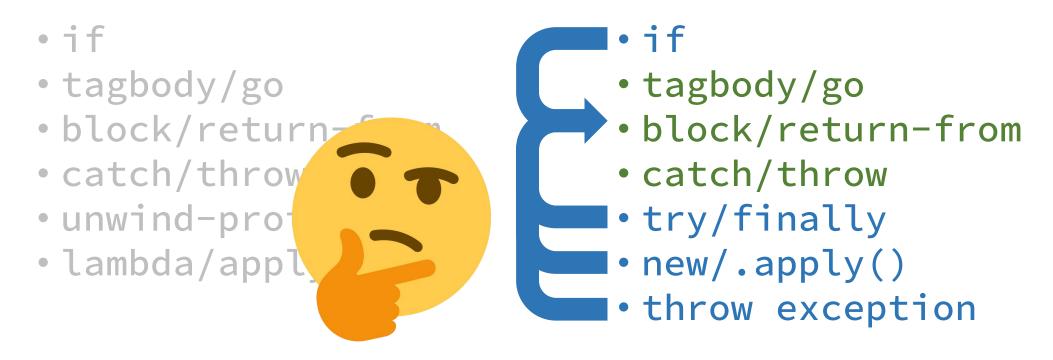
Can we port CL control flow to Java?



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

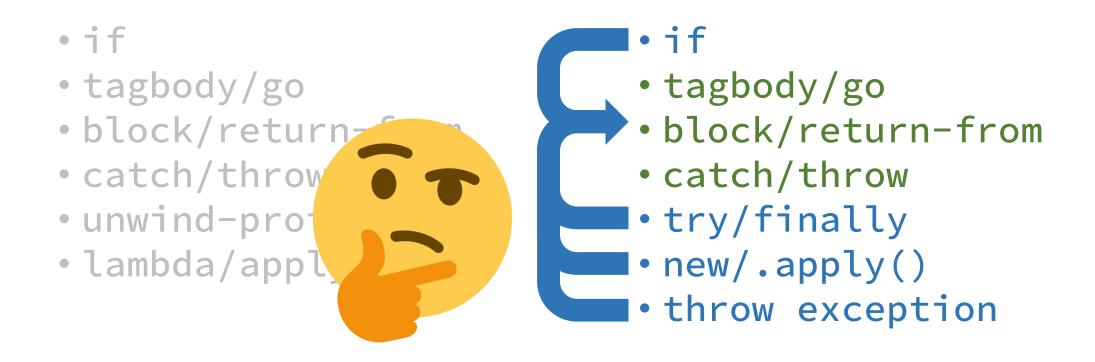
- if
- tagbody/go
- block/return-from
- catch/throw
- try/finally
- new/.apply()
- throw exception





"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



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





yes, it's seriously the end this time