Exceptions and Continuations

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - **SRIs Today!!!**
  - Assignment 9 is due July 30
  - **Final exam, 1:00pm, Friday, August 2 in CHEB 170**
- Readings: Read Chapter 6.6, 9
- Lecture Contents
  - Exceptions
  - Motivation for Continuations
  - Continuations
  - Continuations in Scheme
  - Uses and Abuses of Continuations
  - Implementation
  - Co-routines

How are the Student Ratings of Instruction (SRI) used?

- ✓ Course and program ***(re) design.***
- ✓ ***Evaluation*** of teaching effectiveness.
- ✓ ***Promotion and tenure applications*** for instructors, and other personnel decisions.
- ✓ Preparation of supporting evidence for ***teaching awards and grants***.
- ✓ ***Quality assurance*** processes in the review and restructure of institutional, faculty, department and program goals.

How to complete the SRI

- Find the email in your Dal email account
  - Subject heading (depending on the system) is:
    - *Student Ratings of Instruction; or*
    - *Course Name and Number*
- Open the email and click on the link
  - Your course list should be visible
- Select the course for which you want to complete the evaluation
- Be sure to hit the SUBMIT button when you FINISH completing the form
- You may also SAVE and return to your work later

Also available via Brightspace

Return Values

- In most languages functions typically return r-values
  - A value that can be assigned to a variable or used in an expression
- Some languages, such as C++, allow functions to return l-values (locations of the value)
  - Seen in a previous lecture
- Return of l-values can be simulated in most languages
  - Using pointers in C
  - Returning references in Java
  - Etc.
- But … Sometimes it’s hard to know what to return!

Exception Handling

- Things go wrong (bleep happens), we need to handle it gracefully
- *Exception* are unexpected or abnormal conditions during execution
  - Generated automatically in response to runtime errors
  - Raised explicitly by the program
- Exception handling is needed to
  - Perform operations necessary to recover from the exception
  - Terminate the program gracefully
  - Clean up resources allocated in the protected block
- Exception handling allows the programmer to
  - Specify what to do when an error occurs during program run-time
  - Separate the common path code from the error handling code

Exception Handling Syntax

- Syntax for catching and handling exceptions tends to be similar
- A **protected block** comprises 3 parts:
  - **try** : the common path code to be executed
  - **catch** : exception handlers for each exception to be caught
  - **finally** : an optional "clean-up" handler that always runs after the "try" regardless of whether an exception occurs
- Exception are **raised** (or thrown) by a raise (or throw) statement

```
raise Exception_1(…)
```

```
try {
  // common path
} catch ( Exception_1 e ) {
  // Exception 1 handler
} catch ( Exception_2 e ) {
  // Exception 2 handler
} ...
} else { // optional
  // default handler
} finally { // optional
  // clean up code
}
```

Exception Handling Semantics

- An exception handler is lexically bound to a block of code
- When an exception is raised in the block, search for a handler in present scope
- If there is no matching handler in present scope,
  - The scope is exited (may include block or subroutine)
  - A handler is searched for in the next scope

```
def main():
  try:
    parse()
  except MemoryError as p:
    print “Program too big!”

def parse():
  try:
    l = parseS()
    if lookahead() != None:
      raise ParseError( 'S' )
    eval_result( top_ref, l )
  except ParseError as p:
    print "Syntax Error”
  except EvalError as p:
    print "Evaluation Error”
```

Out of Memory

Language Support

- How are exceptions represented?
  - Built-in exception type (Python)
  - Object derived from an exception class (Java)
  - Any kind of data can be passed as part of an exception
- When are exceptions raised?
  - Automatically by the run-time system as a result of an abnormal condition

    e.g., division by zero, null dereference, out-of-bounds, etc
  - `throw` or `raise` statement to raise exceptions manually
- Where can exceptions be handled?
  - Most languages allow exceptions to be handled locally
  - Propagate unhandled exceptions up the dynamic chain.
    - Clu does not allow exceptions to be handled locally
- Some languages require exceptions that are thrown but not handled inside a subroutine be declared (Java)

Language Non-support

- Some languages do not support exceptions
  e.g., C
- Solution 1:
  - Reserve a return value to indicate an exception
- Solution 2:
  - Caller passes a closure (exception handler) to call
- Solution 3:
  - In C, signals and setjmp / longjmp can be used to simulate exceptions

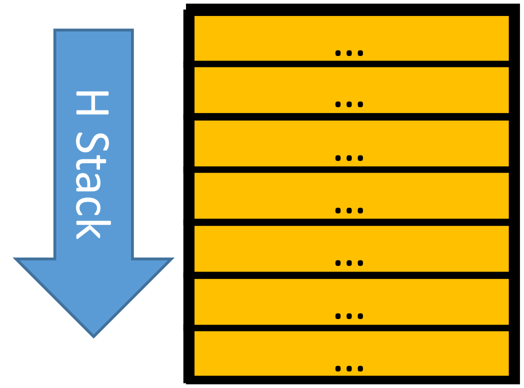
```
#include <setjmp.h>

int func(…) {
  static jmp_buf env;
  int i = setjmp(env);
  if( i == 0 ) {
    /* common path */
    ...
    /* exception 42 */
    longjmp(env, 42);
    ...
  } else if( i == 42 ) {
    /* handle exception 42 */
  } else if( i == ? ) {
    ...
```

Exception Implementations

- Options:
  - Simple (Pay as you go)
  - Location to Exception map (Pay on Exception)
  - Hybrid

Simple, Pay-as-You-Go Exception Handling



- Idea:
  - The program uses a second stack, called a Handler Stack (HS)
  - When a protected block is entered, a handler is pushed on the (HS)
    - Pointer to the handler code
    - Current stack frame (Program Stack)

      I.e., referencing environment

    Sound familiar?
    - An optional exit (finally) handler may also be pushed
  - If there are multiple exception handlers, these are implemented using an if/elseif/… construct in a single handler
  - When a protect block is exited, the handler is popped of the stack
- Simple implementation is costly because handler stack is manipulated on entry/exit of each protected block

Simple, Pay-as-You-Go Exception Handling

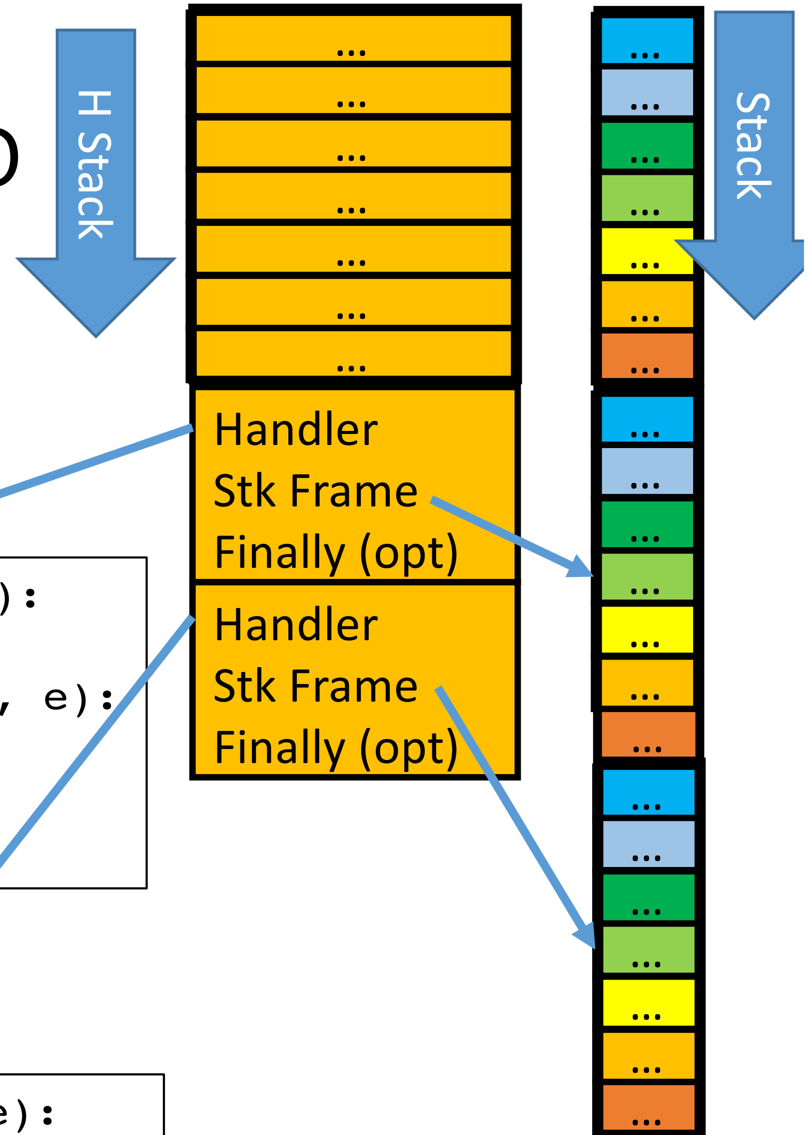
```
def foo():
  try:
    bar()
  except E1 as e:
    # E1 handler
  except E2 as e:
    # E2 handler
```

```
if isinstance(E1, e):
  # E1 handler
elsif isinstance(E2, e):
  # E2 handler
else:
  raise e
```

```
def bar():
  try:
    ...
  except E3 as e:
    # E3 handler
  except E4 as e:
    # E4 handler:
```

E1

```
if isinstance(E3, e):
  # E3 handler
elsif isinstance(E4, e):
  # E4 handler
else:
  raise e
```



Location to Exception Mapping

- A faster implementation (Pay on exception)
- Store a global map of code blocks (memory addresses) to handlers
  - Generated by compiler/linker
- On exception, index map with program counter to get handler
- Still need to keep track of stack frames
  - Each stack frame stores a pointer to most recent protected block

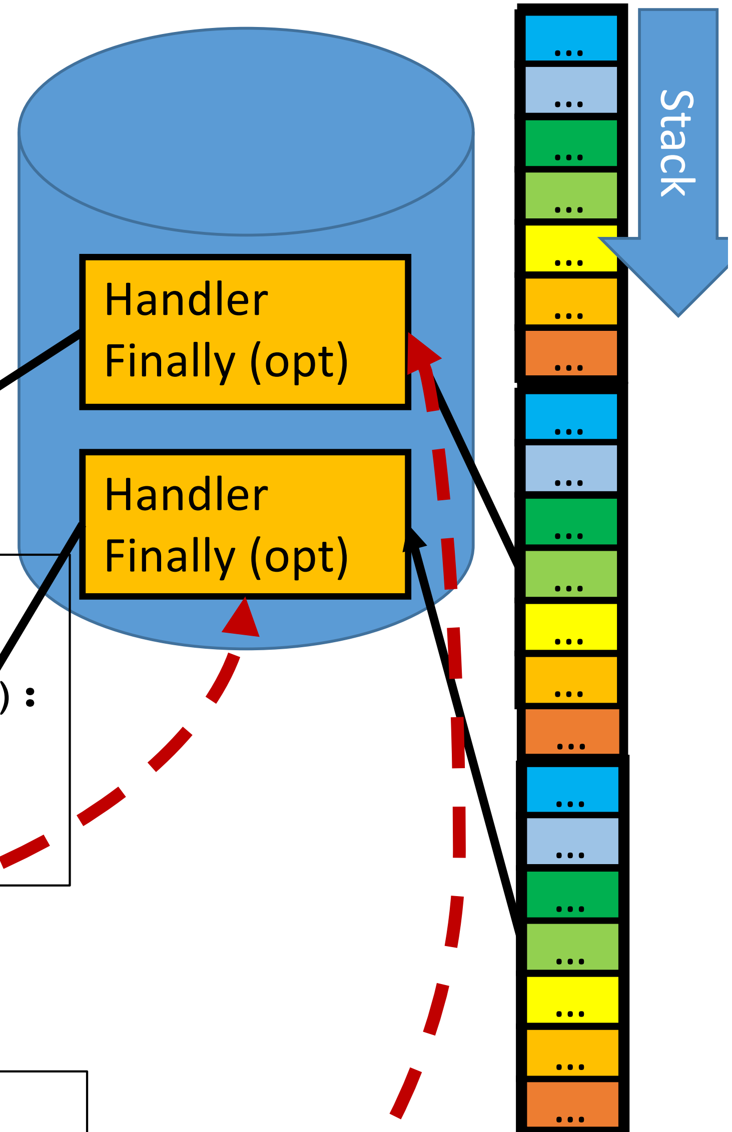
Pay on Exception
Exception Handling

```
def foo():
  try:
    bar()
  except E1 as e:
    # E1 handler
  except E2 as e:
    # E2 handler
```

```
if isinstance(E1, e):
  # E1 handler
elsif isinstance(E2, e):
  # E2 handler
else:
  raise e
```

```
def bar():
  try:
    ...
  except E3 as e:
    # E3 handler
  except E4 as e:
    # E4 handler:
```

```
if isinstance(E3, e):
  # E3 handler
elsif isinstance(E4, e):
  # E4 handler
else:
  raise e
```



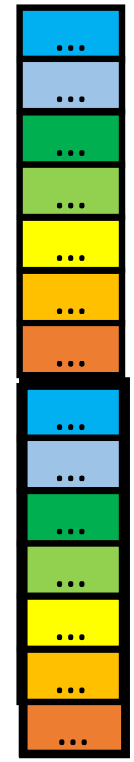
Comparison of the 2 Approaches

- Location-based Exception handling
  - Handling an exception is more costly (search), but exceptions rare
  - No cost if no exceptions
  - Cannot be used if the program consists of separately compiled units and the linker is not aware of this exception handling mechanism
- Hybrid Approach:
  - Use a local map for each subroutine
  - Store a pointer to a local map in subroutines stack frame

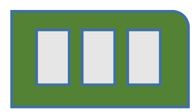
Motivation

- Its useful to have a general way to implement a variety of mechanisms such as
  - Exceptions
  - Gotos
  - Coroutines
  - Subroutines
  - Closures
  - Transform recursion into tail recursion
  - etc
- Continuations are a general mechanism for doing this.

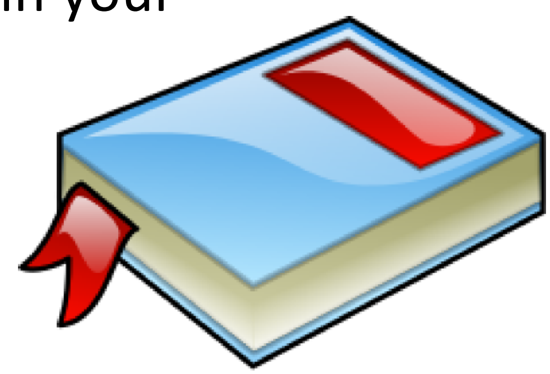
Continuation



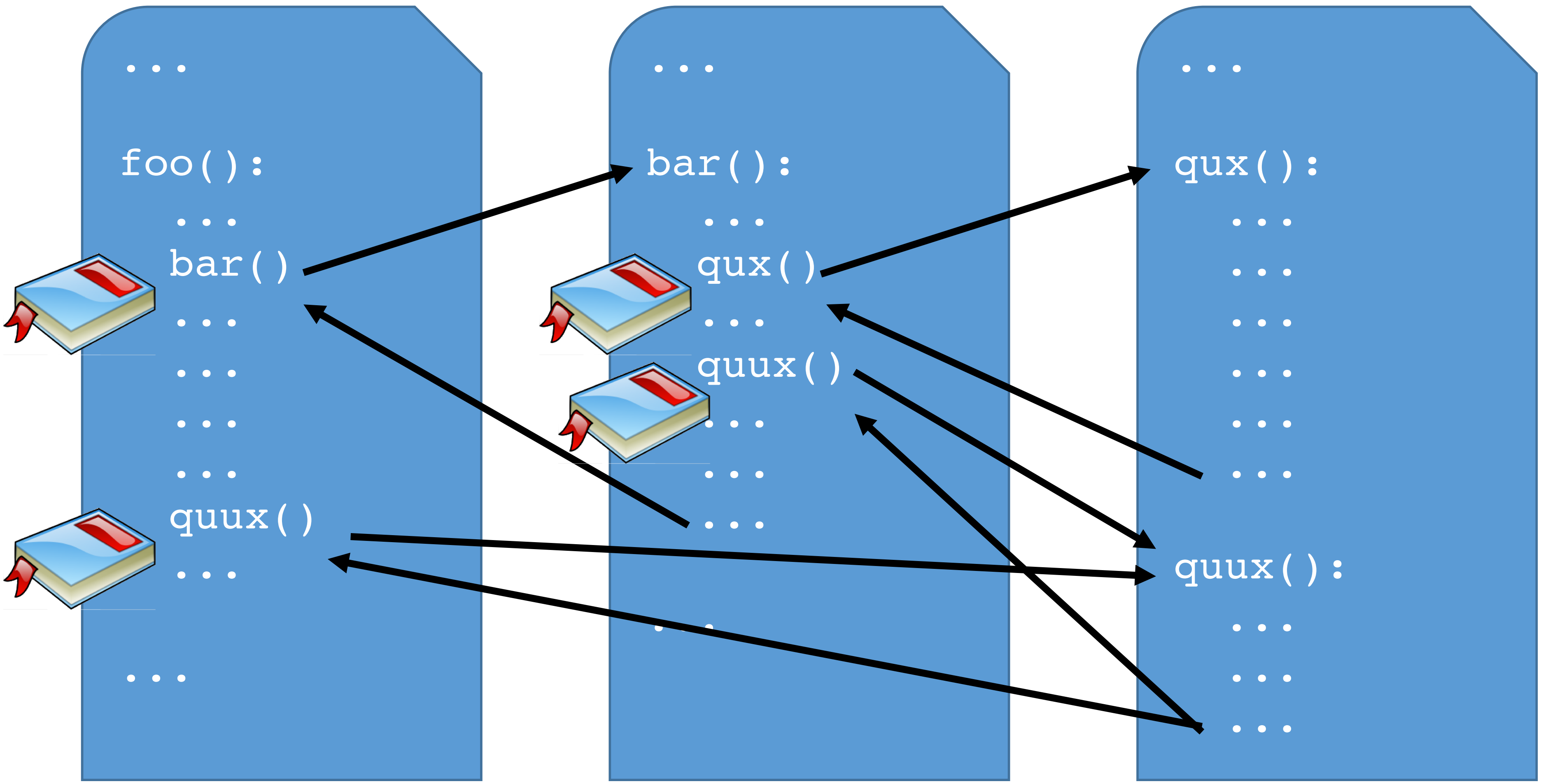
- A continuation is the “future” of the current computation
- Represented as the current
  - Stack contents (sequence of stack frames)
  - Referencing environment
  - Current program state
    - Program counter (current location)
    - Registers
    - Etc.



- Analogy: A bookmark in the computation
  - A way for a program to return to the location in your program, as if nothing has happened
- Analogy: A “back” button for a program



We Already Use Bookmarks!

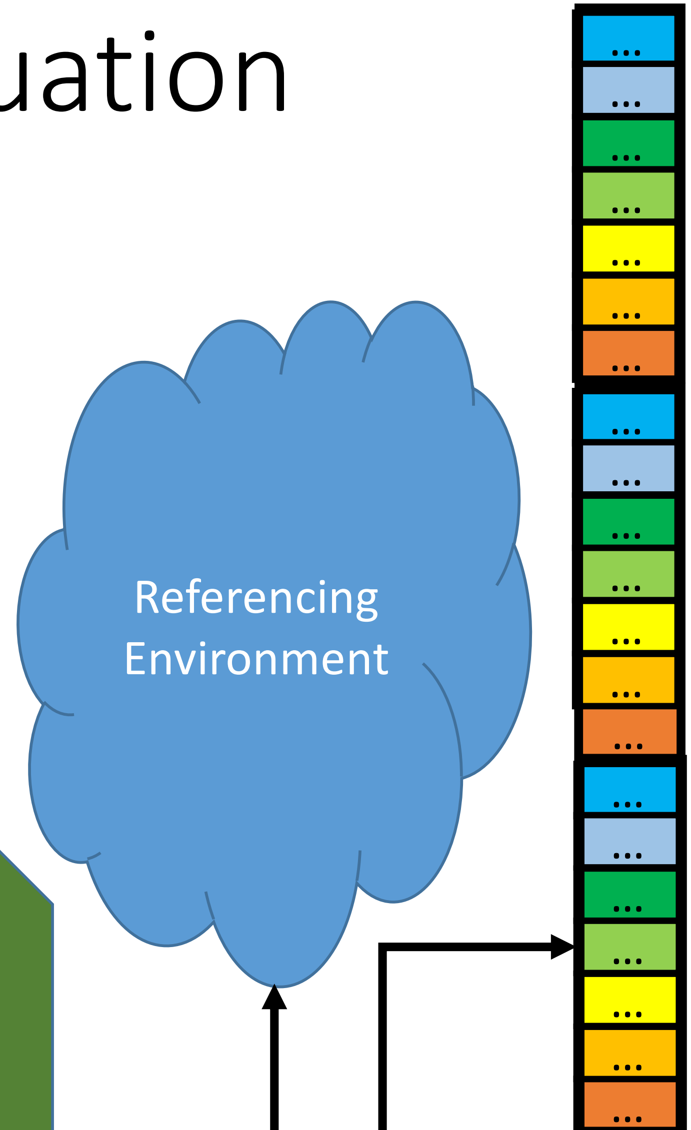


A Picture of a Continuation

```
( define find_neg
  ( lambda ( L )
    ( define finder ( lambda ( exit )
        ( define do_check ( x )
                   ( if ( negative? x)
                         ( exit x) ) )
        ( for-each do_check L )
         #t )
    ( call/cc finder ) ) )
```

Continuation:

- Program State
  - Location
  - Registers
- Referencing Environment
- Stack Frames

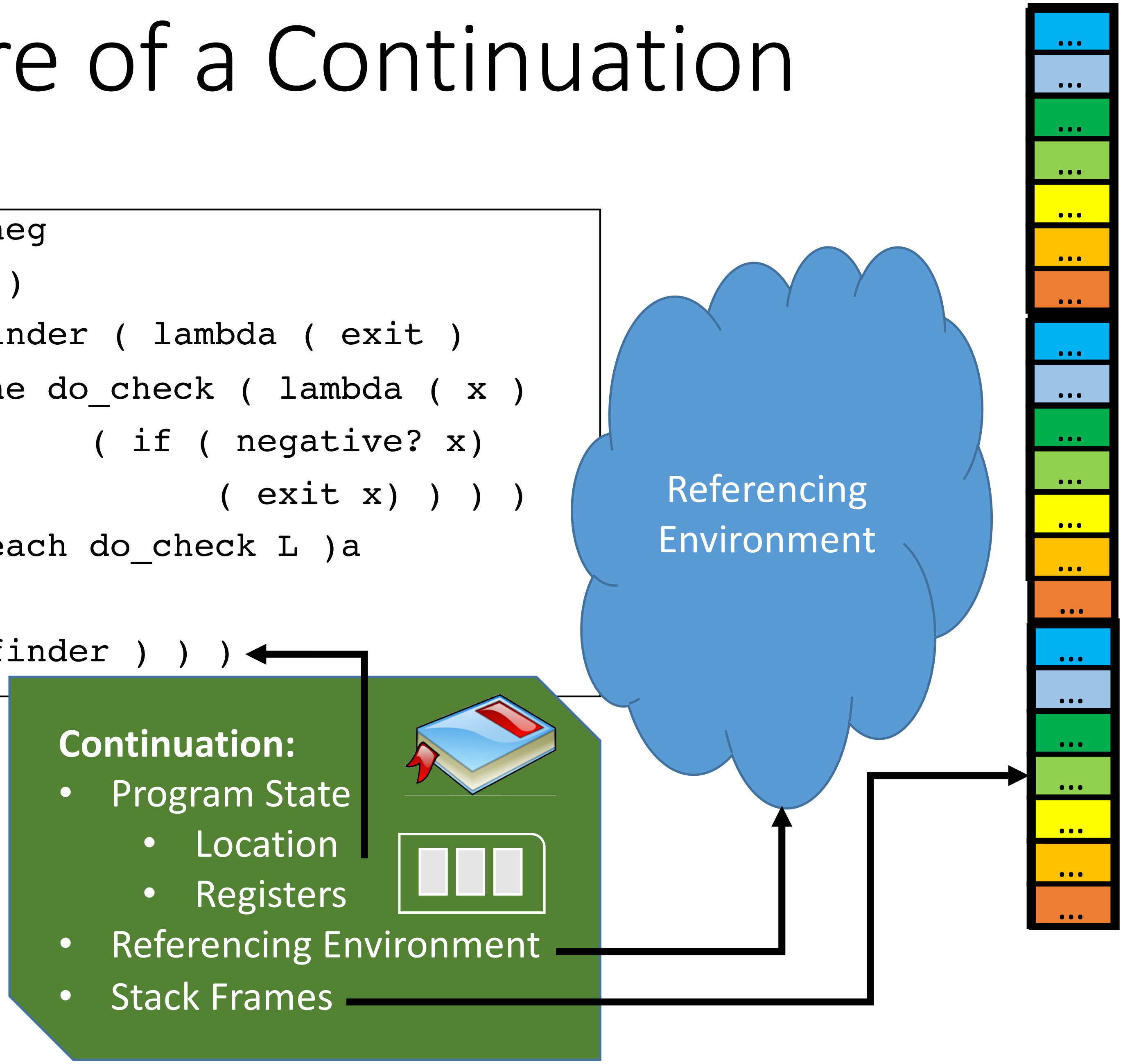


Aside: Current Program State

- What does the immediate behaviour of a program depend on?
  I.e., what affects
  - Which instruction is executed next?
  - The result of the instruction?
- Answer: the CPU State
  - Program counter (location of next instruction)
  - General purpose registers (current values being manipulated by the program
  - Stack pointer register (current stack frame)
- Idea: Current program state corresponds to the current state of the CPU, including current location in the program

A Picture of a Continuation

```
( define find_neg
  ( lambda ( L )
    ( define finder ( lambda ( exit )
        ( define do_check ( lambda ( x )
                  ( if ( negative? x)
                        ( exit x) ) ) )
        ( for-each do_check L )a
         #t )
    ( call/cc finder ) ) )
```



Continuation as First-Class Objects

- A *first-class* object is a something that can be passed to a function and returned by a function
- Continuations are first-class objects in Scheme:
  - Passed as function arguments
  - Returned as function results
  - Stored in variables and data structures
- Note: A continuation can be “resumed” from *anywhere* in the program
  - Just like flipping to a bookmark can be done from anywhere in a book!

Continuations in Scheme

- Continuations are created by
  - Taking a snapshot of the current state of the program

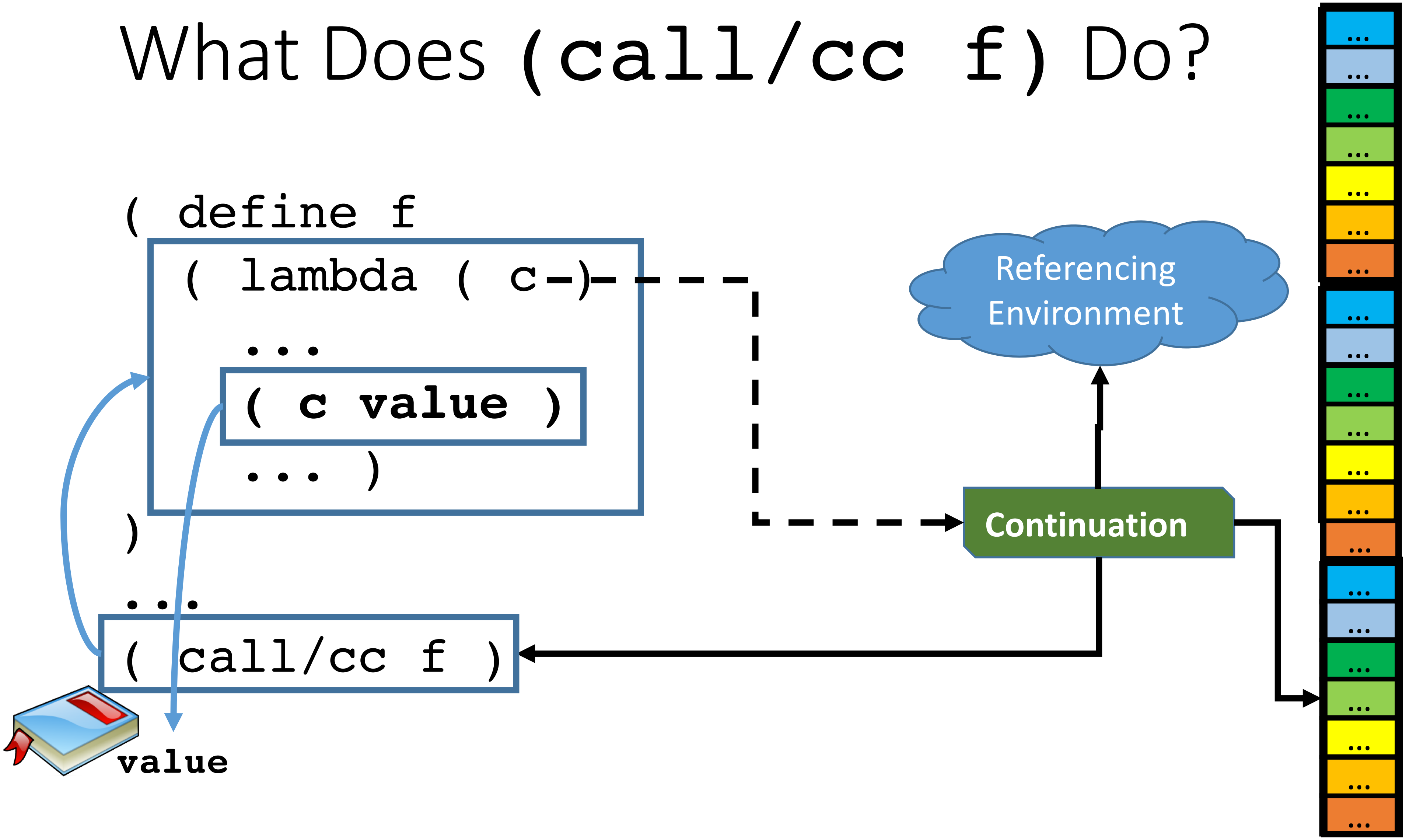
    I.e., creating the continuation
  - Calling a function and passing it the snapshot
- In Scheme this is done with the special function:

```
( call-with-current-continuation f )
```

  - Calls function *f*
  - Passes the current continuation to f as an argument

  Short form: `( call/cc f )`

What Does `(call/cc f)` Do?



Uses of Continuations

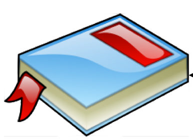
- Continuations can be used to support a variety of special purpose constructs in other languages
  - Escape procedures
  - Exception handling
  - Co-routines
  - Etc

Escape Procedure with Continuation

- Simplest possible use: Escape procedure
  - If function *f* never makes use of the continuation, everything works as if `( f )` was performed
  - If function *f* invokes the continuation, then program state is restored as if f was never called
- Example: Look for the first negative number in a list

```
( define find_neg ( lambda ( L )
  ( call/cc ( lambda ( exit )
    ( for-each ( lambda ( x )
                  ( if ( negative? x )
                    ( exit x ) ) )
                L )
    #t ) ) ) )
```

Continuation



x

- What happens when

```
( find_neg ‘ ( 54 0 37 -3 245 19 ) )
=> -3
```

Exception Handling with Continuations

- Suppose you want to sum a list of integers…

```
( define sum
   ( lambda ( lst )
      ( call/cc ( lambda ( exception )
         ( define r ( lambda ( L )
            ( cond
               ( (null? L ) 0 )
               ( ( integer? ( car L ) )
                  ( + (car L ) ( r ( cdr L ) ) )
               ( #t ( exception #f ) ) ) ) )
         ( r lst ) ) ) ) ) )

( sum ( 1 2 3 4 ) )
=> 10
( sum ( 1 b 3 4 ) )
=> #f
```

Return to this continuation if an exception occurs

Empty list sum is 0

Else if next item is an integer

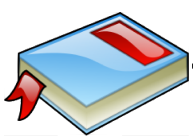
Else, next item is not an integer

Call r with lst

Gotos with Continuations

- But wait there is more!
- The continuation can be invoked anywhere!
- Even after we leave the scope where it was created
- What does this do?

```
( define wormhole #f )
( define rabbit-hole
  ( let ( ( x 42) )
    ( call/cc ( lambda ( hole )
         ( set! wormhole hole ) ) )
    ???
    ... ) ) )
( wormhole 7 )
```



- This is almost like a goto!
- Use at own risk!

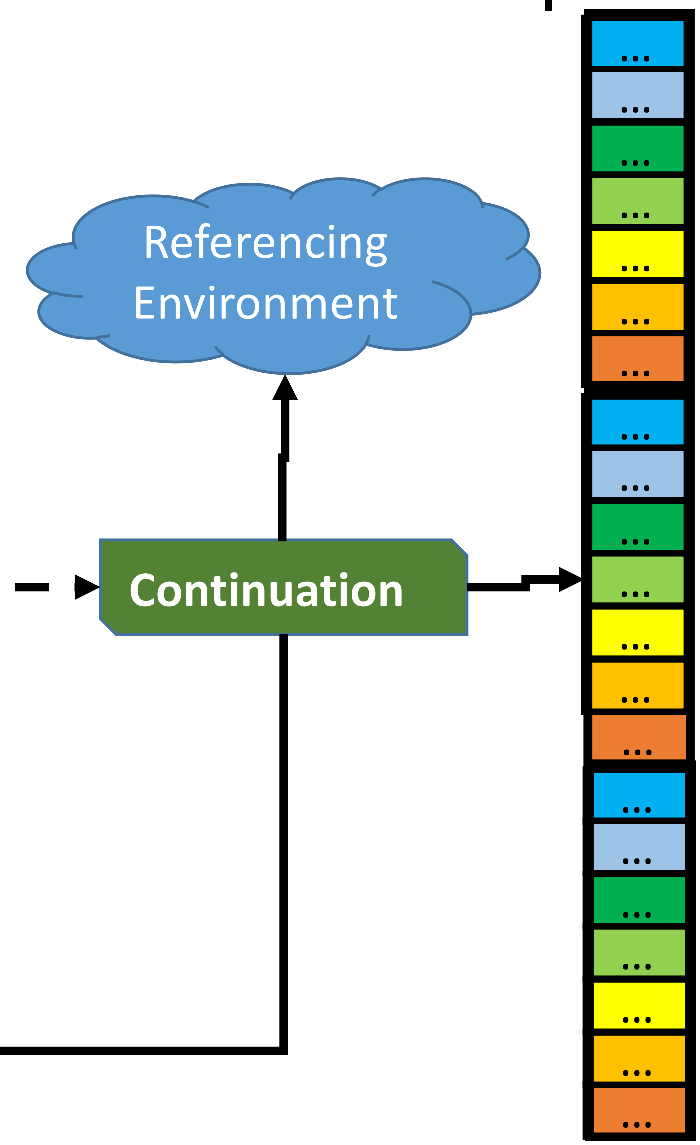
Implementing Continuations

- To implement continuations, need to preserve:
  - Referencing environment : easy
    - Same as closures
  - Current program state: easy
    - Store current CPU state inside a continuation record
  - Stack content : Not so easy (it depends)
    - If continuations are only used within scope of creation, then pointer to stack frame is sufficient
    - If continuations used anywhere, need to make a copy of the entire stack! Why?

Continuations Called within Scope

- If a continuation is called within the scope it was created, the stack frame present at its creation has not been destroyed

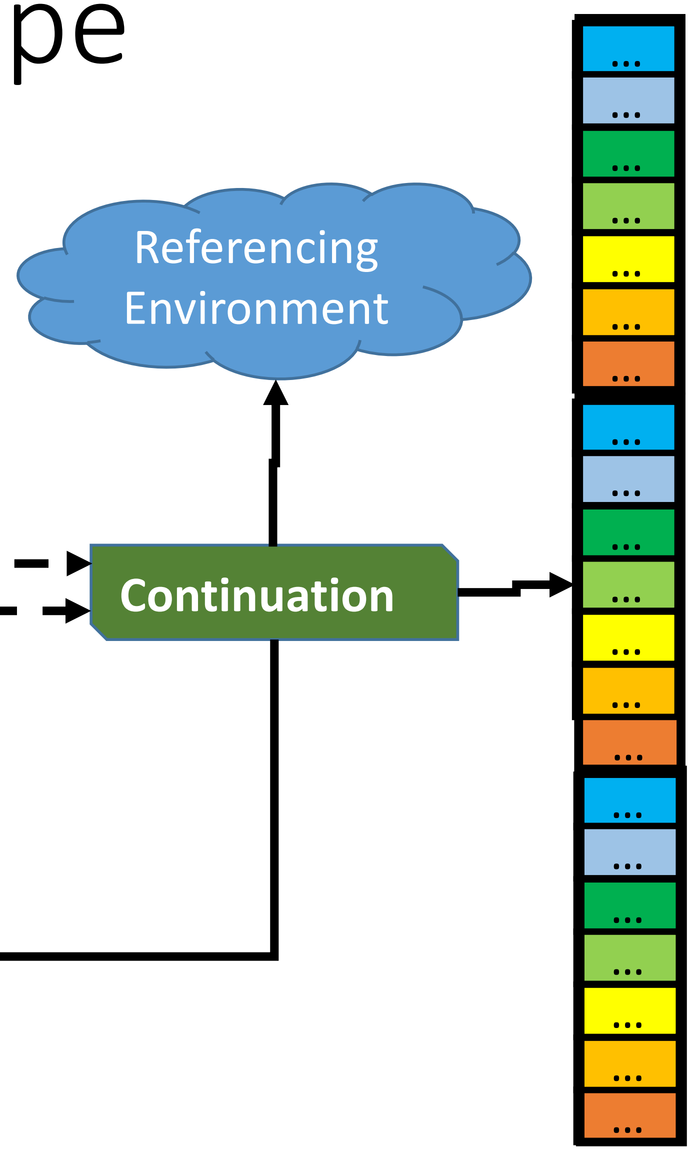
```
( define f
   ( lambda ( c )
      ...
      ( c value )
      ... )
)
...
( call/cc f )
```



The Challenge with Continuations Called Outside of Scope

- If a continuation is called outside the scope it was created, the stack frame present at its creation may be destroyed ☹

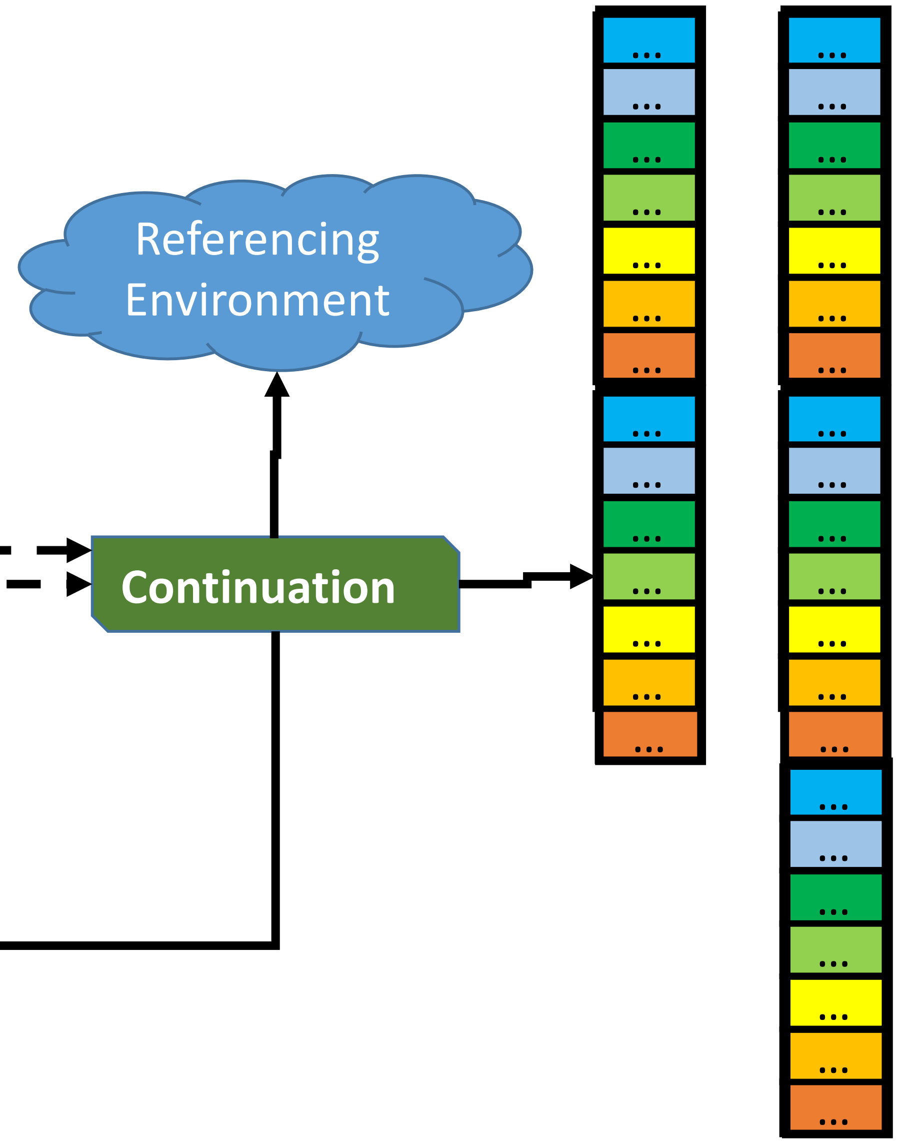
```
( define ext 42 )
( let ( ( y 7 ) )
  ( define f
    ( lambda ( c )
      ( set! ext c )
    )
  )
  ( call/cc f )
   ...
)
( ext 13 )
```



Solution to Continuations Called Outside of Scope

- If a continuation is called outside the scope it was created, the stack frame present at its creation has to be duplicated

```
( define ext 42 )
( let ( ( y 7 ) )
   ( define f
      ( lambda ( c )
         ( set! ext c )
      )
   )
   ( call/cc f )
)
( ext 13 )
```



Recall setjmp()/longjmp()

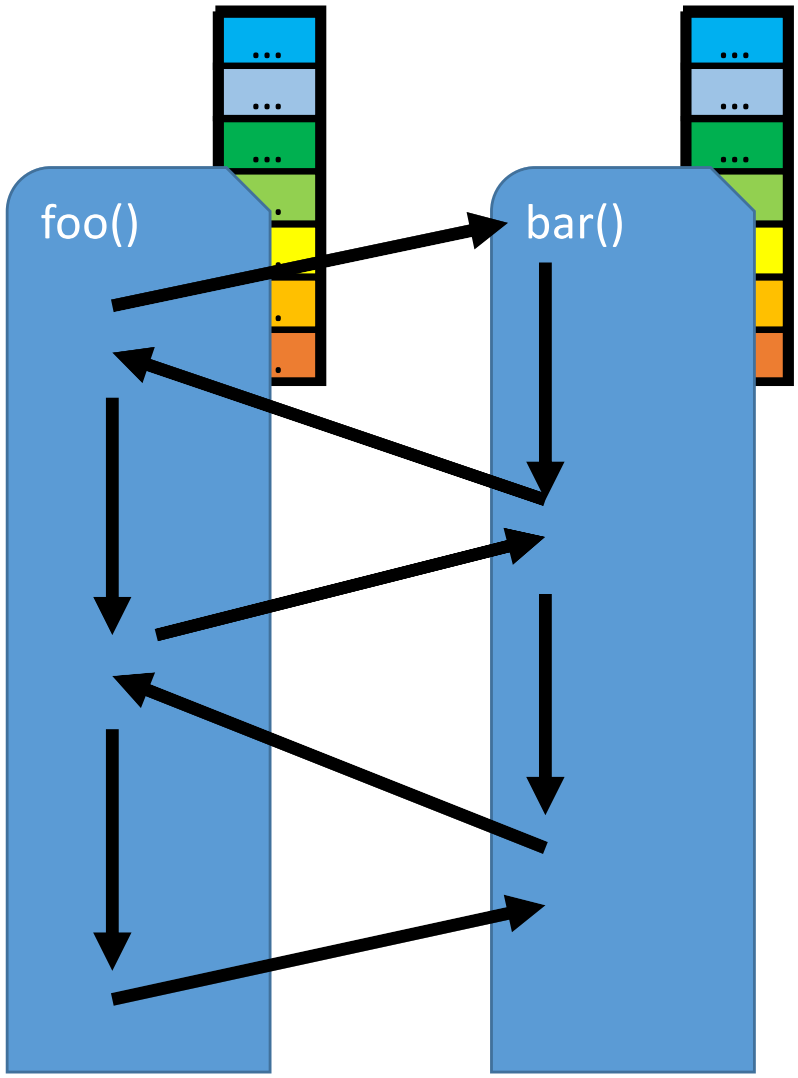
- First invocation of `setjmp()`
  - stores the current context in `env`
  - returns 0
- If no `longjmp(env,…)` occurs
  - Branch terminates as usual
- If `longjmp(env,val)` occurs
  - It jumps directly into `setjmp()`
- The `setjmp()` returns again
  - And returns `val`
  - The else-branch is executed
- Recall, this is how we would implement exceptions in C

```
static jmp_buf env;
...
val = setjmp( env );
if( val == 0 ) {
  /* protected code */
  ...
  longjmp(env,val);
  ...
} else {
  /* handler */
}
...
```

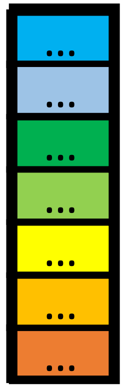
From the man page: *"The longjmp() routines may not be called after the routine which called the setjmp() routines returns."*

Coroutines

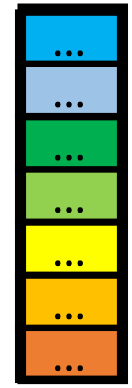
- Coroutines are separate threads of execution that yield control to each other
  - In contrast, real threads do not yield control
- Coroutines were commonly used to structure concurrent operations
- Useful for implementing
  - iterators
  - Generators
- Example:
  - Jumping back and forth from a generator
- Challenge: Need a separate stack for each coroutine.



Coroutines using Continuations



```
(define foo (lambda (…)
  ...
  (let ((next 0))
    ...
    (set! next (call/cc bar))
    ...
    ...
    ...
    (set! next (call/cc next))
    ...
    ...
    ...
    (set! next (call/cc next))
    ...
)))
```



```
(define bar (lambda (next)
  ...
  ...
  ...
  (set! next (call/cc next))
  ...
  ...
  ...
  (set! next (call/cc next))
  ...
  ...
  ...
  (set! next (call/cc next))
)))
```

Discussion:

- Everything old is new again

  - Stephen King, and many others
- Closures and continuations are not “new” concepts
- The have been around for quite some time
- They are now being rediscovered, implemented and used in modern languages because they are a useful way of specifying computation
- These are not esoteric concepts that you will never use
- Languages such as
  - Scala
  - Ruby
  - Actionscript
  - Python
  - Java

  Have these features or will have them!

Coming Up Next

- Type Systems
- Data Types
- Arrays
- Garbage Collection