

Control Flow

Edward Z. Yang

1. Structured Programming
2. Procedural abstraction — the stack
3. Continuations

Although this slide may not show it, there is a certain progression that today's discussion of control flow will take. We'll start at the beginning of computing history, when things were more hardcore, and programmers did all of their control flow using GOTO. We'll trace the structured programming revolution, which espoused the now obvious idea that instead of using GOTO, more structured control flow operators like IF and FOR should be used.

GOTO/IF/FOR all have the characteristic that their destinations are statically known. However, there is useful control flow which is based on runtime behavior. The two most common forms, procedural abstraction and exceptions, are based off the stack, so we'll spend some time looking more closely at these very familiar control flow concepts. This will lead us back to lower abstraction levels and look at continuations, a generalization of GOTO supporting indirect control flow, and the substrate from which all control flow can be derived.

```

10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
    IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.000001) GO TO 30
    X = X-Y-Y
30  X = X+Y

    . . .
50  CONTINUE
    X = A
    Y = B-A
    GO TO 11

    ...

```

To understand what "structured programming" is, we have to turn the clock back to a time when structured programming **wasn't** the standard methodology for writing programs. At the dawn of computing, your programs might have looked something like this Fortran code. Can you tell what this program does? Me neither.

In 1968, Dijkstra wrote a letter to the editor in the Communications of the ACM, arguing that GOTO was harmful, and advocating that programmers switch to new "higher level" control flow statements.

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

Go To Statement Considered Harmful



What did Dijkstra argue in his essay? You should read it yourself (it has aged well with time), but essentially, Dijkstra describes the "spaghetti" nature of GOTO programs. He argues that one of the primary ways we understand programs is by some semi-linear processing of the code as a sequence. GOTO destroys this sequencing, since any code can jump anywhere in the program.

Structured Programming with **go to** Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

As an aside, Knuth later wrote a response to Dijkstra's essay, stating that GOTO should be used in some circumstances...

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

Keywords and phrases: structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

CR categories: 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)



I argue for the elimination of go to in certain cases, and for their **introduction** in other cases.

In his essay, Knuth described some common control flow patterns which are greatly assisted by the use of GOTO. Many of these control flow patterns have since been reabsorbed into high-level languages ("break" for the search/ashes, exceptions for error exits, switch for merged conditional branches, TCO for binary search, coroutines), but any working C programmer will tell you, when tastefully applied, that the GOTO statement is a great boon for program readability and understandability: it's a way of bootstrapping your own structured control flow operators.

Structured Programming

Dijkstra's original essay coincided with the rise of a new paradigm of programming, structured programming.

if...then ... else...

while... do...

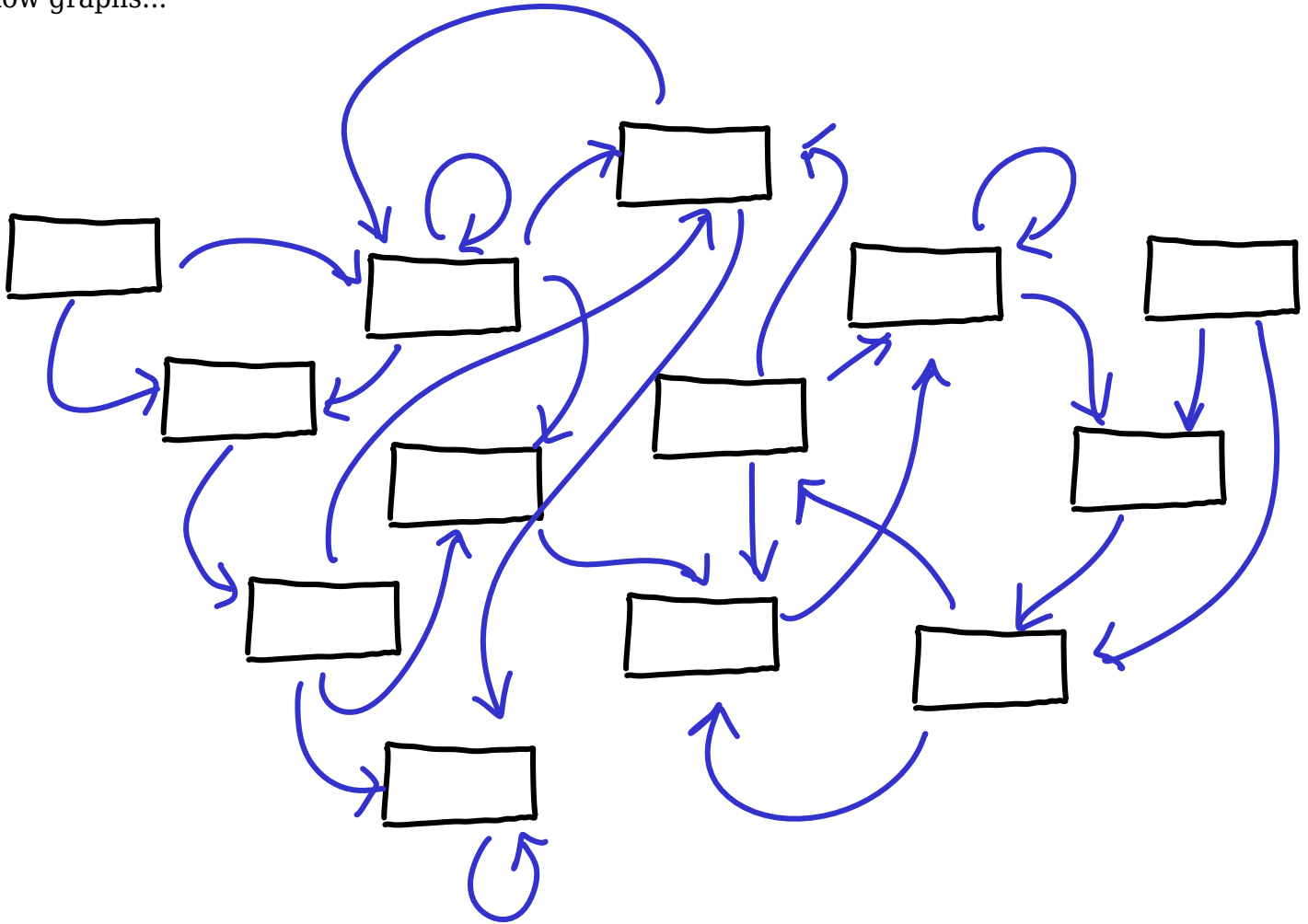
Structured Programming

case...

for... { ... }

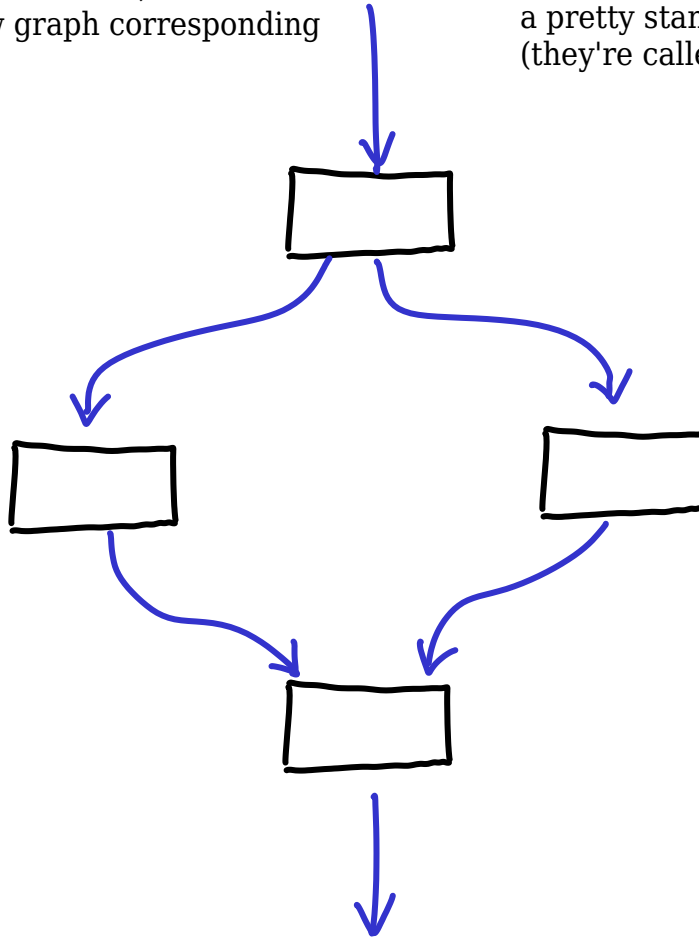
For a modern programmer, this style is so ubiquitous so as to be utterly banal

But it was a real sea change back in the day. It let programmers take otherwise unstructured flow graphs...

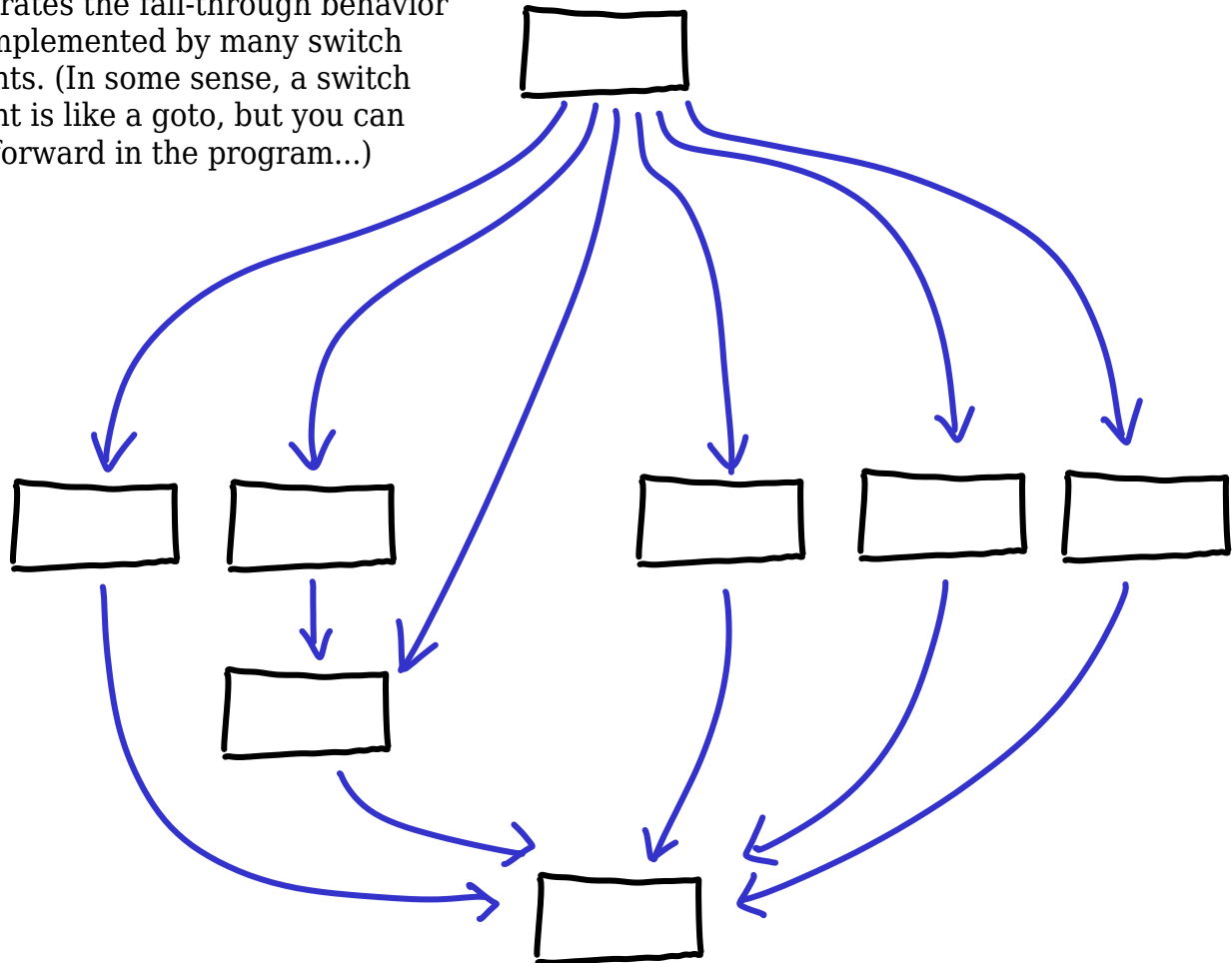


...and impose structure on them. Here, for example, is a control flow graph corresponding to an if statement.

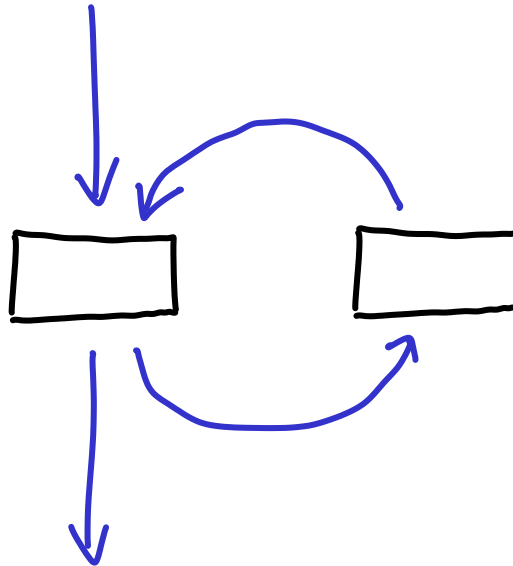
By the way, organizing straight-line code into "blocks" like this is a pretty standard compiler technique (they're called "basic blocks").



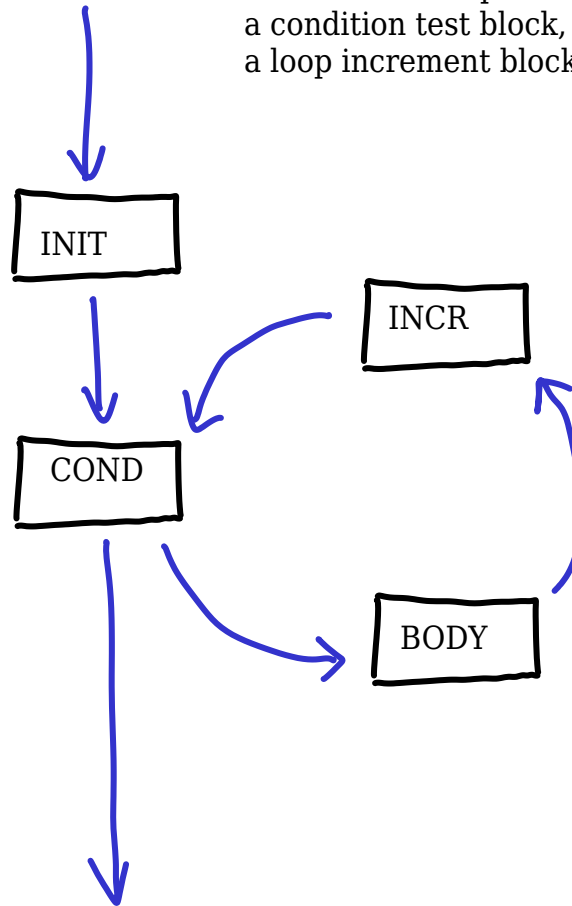
Here's another one, showing a switch statement. The "doubled up" block demonstrates the fall-through behavior that is implemented by many switch statements. (In some sense, a switch statement is like a goto, but you can only go forward in the program...)



Here's a while loop (the left block is the condition, and the right block is the loop body).



Here is a for loop. We have an initialization block, a condition test block, a loop body block, and a loop increment block.



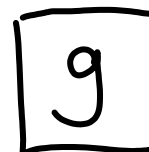
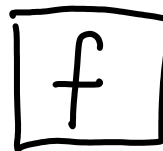
Procedural Abstraction

The Stack

Another essential component of the structured programming was the shift towards procedural abstraction: the idea that programs shouldn't be single blobs of code but be organized into subroutines of code which could be called multiple times. Once again, in modern development practice, this style of programming is also ubiquitous, although procedural abstraction is a very real skill that new programmers have to be taught.

My job is not to teach you about procedural abstraction, but rather, I want to use it as an example to segue into a form of control flow which is different from the block concepts we've seen earlier. We'll see this with a little example.

```
function f(x) {  
    return h(x) + 1;  
}
```



```
function g(x) {  
    return h(x) - 1;  
}
```

```
function h(x) {  
    return x * 2;  
}
```

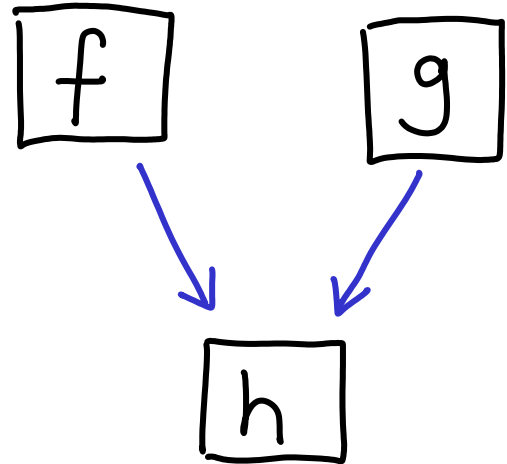


What is the control flow diagram for this code? Function calls are simple enough...

```
function f(x) {  
    return h(x) + 1;  
}
```

```
function g(x) {  
    return h(x) - 1;  
}
```

```
function h(x) {  
    return x * 2;  
}
```

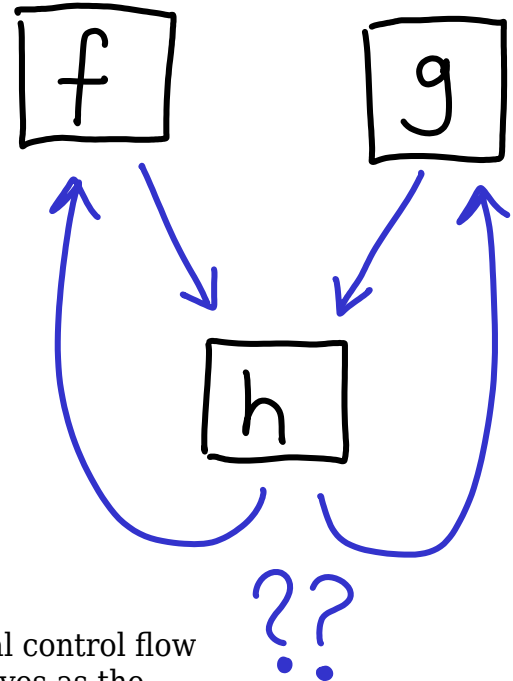


But what happens when we RETURN from the function call? An execution of g could return to f or g or maybe some arbitrary code which called into h. There might be a lot of places we could return to. At runtime, how do we know where to go?

```
function f(x) {  
    return h(x) + 1;  
}
```

```
function g(x) {  
    return h(x) - 1;  
}
```

```
function h(x) {  
    return x * 2;  
}
```



By the way, this problem of determining the interprocedural control flow graph of a program is a quite important one, because it serves as the basis for approaches to control-flow integrity, one of the primary defenses against code execution attacks (e.g. buffer overflows and return-oriented programming.)

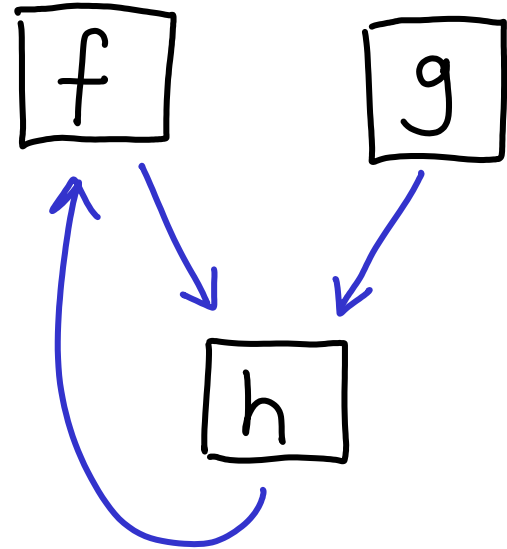
control link
parameters
local variables
return to main

control link
parameters
local variables
return to f

Stack!

Why do stacks grow downwards? It's a strange historical accident, stemming from stacks usually living at the END of the address space (code living in front!)

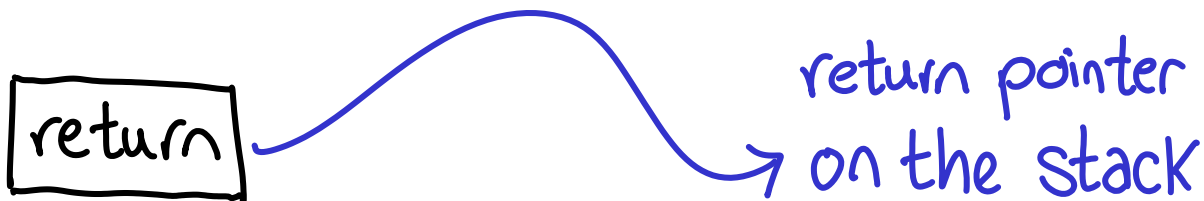
The answer is that the return pointer is stored in the activation frame on the stack.



Nitpickers, this slide is not quite right: it is not that we are returning to f, but we are returning to the "CONTINUATION" of f; the code to be run AFTER the function call! In C, this is just the address of the pointer immediately after the CALL instruction.

The stack dictates where my control goes.

To highlight the difference; if I make a conditional branch, I know exactly where my execution will go if the condition is true or false. But suppose I throw an exception. Where will control end up?



Dynamic control flow

The answer: there is no way to know without consulting the `*stack*`, which lets us know where to return to.

Similarly, shellcode which overwrites the stack can execute arbitrary code (without writing instructions to the stack) precisely because by providing an arbitrary list of return addresses, you can program the control flow of existing "gadgets" in code. This dynamism is how most arbitrary code execution exploits operate!

As it turns out, the stack gets leaned on a lot for this kind of dynamic control flow.

Exceptions

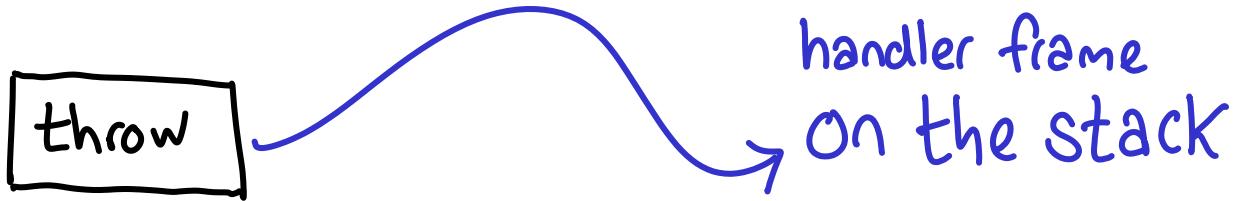
Also The Stack

There are two primary components to exceptions: the ability to throw an exception (where the control flow branches off) and the ability to catch an exception (where the control flow returns to.)

raise/throw

handler/catch

Dynamic control flow



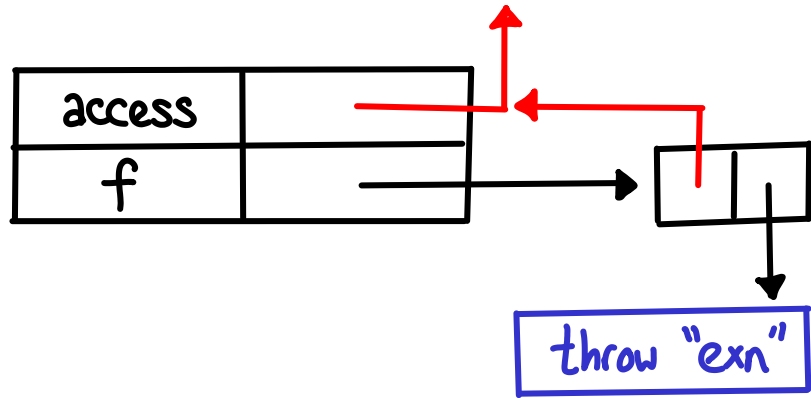
Dynamic control flow

Here's as simple example in JavaScript to demonstrate.

```
function f(y) { throw "exn"; }  
try {  
    f(1);  
} catch(e) { showError(); }
```



```
function f(y) { throw "exn"; }  
try {  
    f(1);  
} catch(e) { showError(); }
```

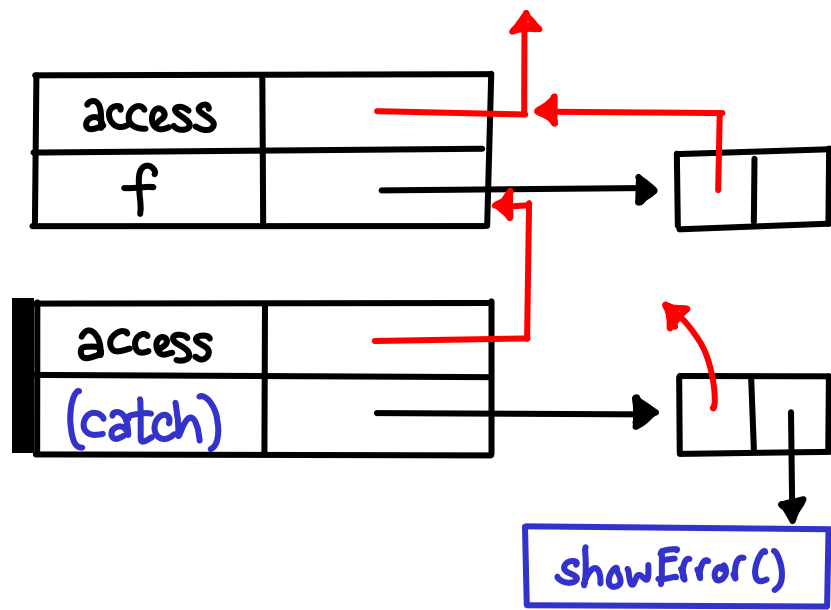


```
function f(y) { throw "exn"; }  
try {  
    f(1);  
} catch (e) { showError(); }
```

When we handle a try-statement, we now push an exception handler onto the stack; denoted using a black stripe on the left of the box.

```
function f(y) { throw "exn"; }  
try {  
    f(1);  
} catch(e) { showError(); }
```

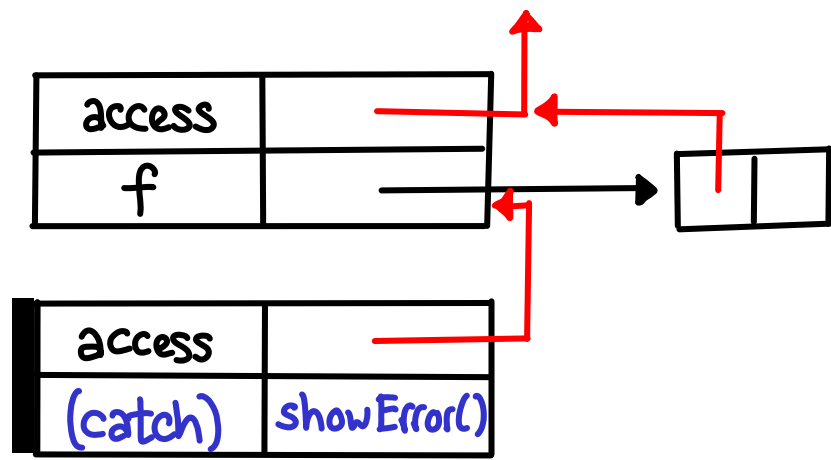
Convention: in these examples, the control link is implicitly pointing to the record immediately above.



In this slide, I've modeled the try block as a new activation record. However, this is a bit inaccurate, because try doesn't actually introduce a new lexical scope. For the purposes of this lecture, it won't really matter. Maybe some day I'll redo the slides to not conflate activation records and the control stack, but this is how the textbook is setup too.

NB: try does not actually introduce scope

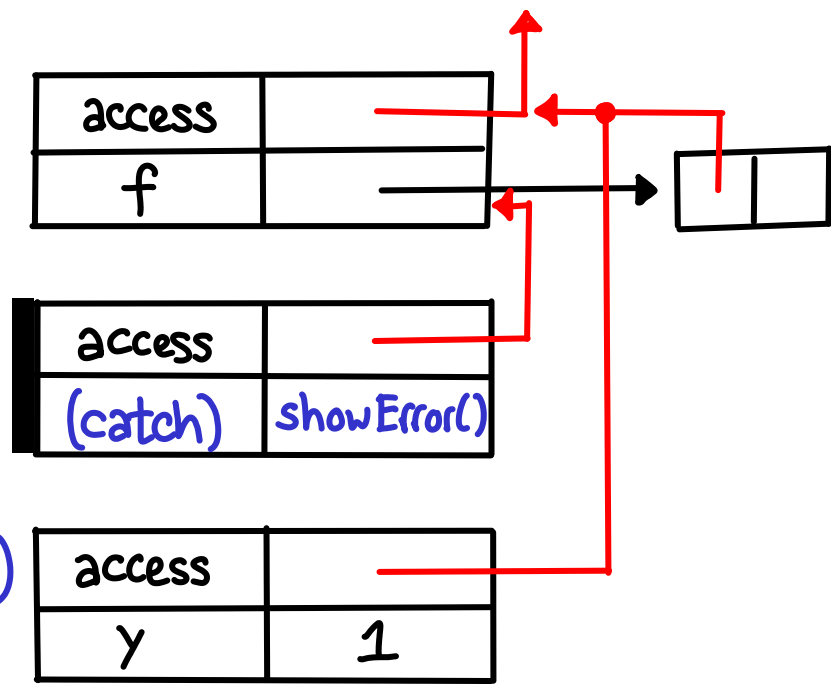
```
function f(y) { throw "exn"; }  
try {  
    f(1);  
} catch (e) { showError(); }
```



```

function f(y) { throw "exn"; }
try {
  f(1);
} catch (e) { showError(); }

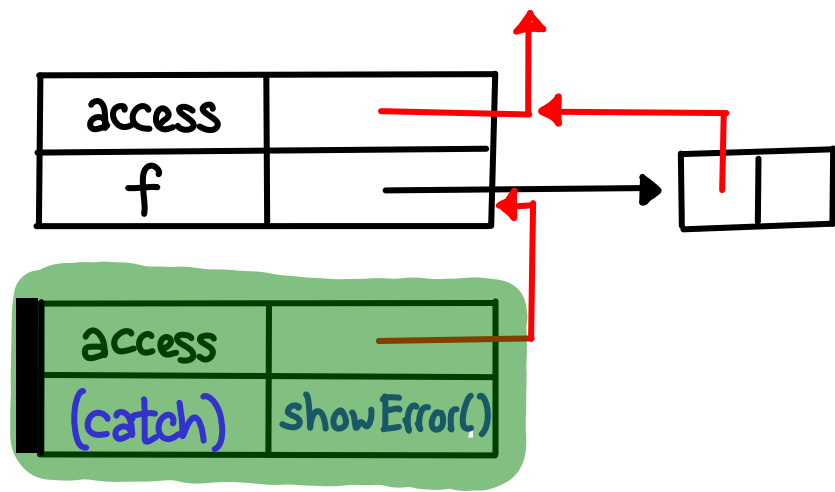
```



```

function f(y) { throw "exn"; }
try {
    f(1);
} catch(e) { showError(); }

```



When an exception is thrown, we walk up the stack looking for the nearest handler frame, and execute the code associated with it (ignoring any stack frames we traversed along the way).

```
try {  
  function f(y) { throw "exn"; }  
  function g(h) { try { h(1); }  
                  catch(e) { ③ } }  
}
```

```
  try {  
    g(f);  
  } catch(e) { ① }  
} catch(e) { ② }
```

Here is a modestly more complex example.

```
try {  
  function f(y) { throw "exn"; }  
  function g(h) { try { h(1); }  
                  catch(e) { ③ } }  
  
  try {  
    g(f);  
  } catch(e) { ① }  
} catch(e) { ② }
```




try {

function f(y) { throw "exn"; }

function g(h) { try { h(1); }

catch(e) { ③ } }

try {

g(f);

} catch(e) { ① }

} catch(e) { ② }

try {

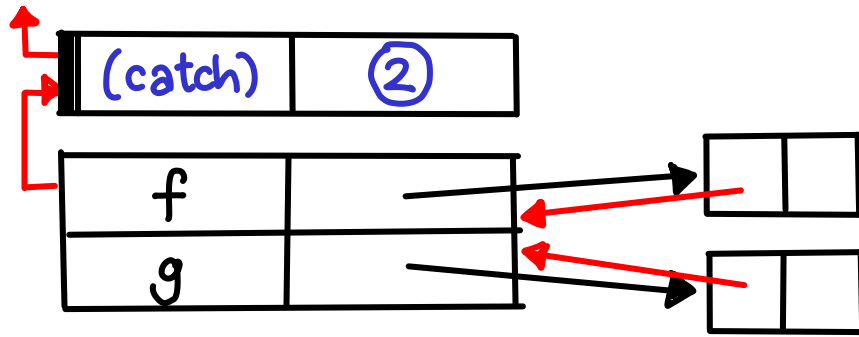
```
function f(y) { throw "exn"; }  
function g(h) { try { h(1); }  
                catch(e) { ③ } }
```

try {

g(f);

} catch(e) { ① }

} catch(e) { ② }



```

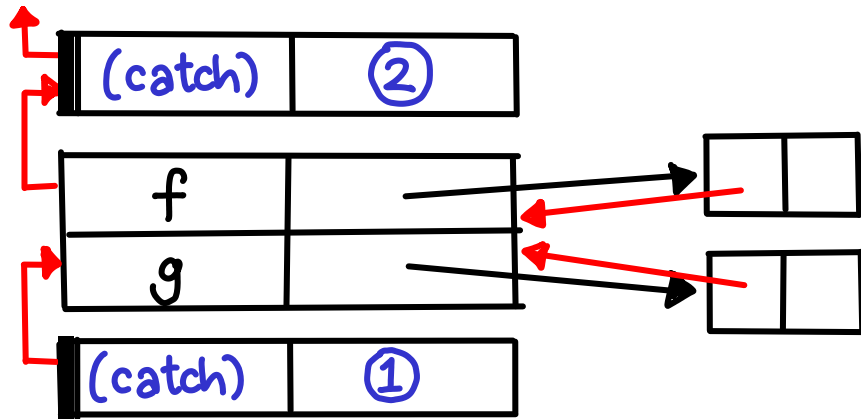
try {
  function f(y) { throw "exn"; }
  function g(h) { try { h(1); }
                  catch(e) { ③ } }

```

```

  try {
    g(f);
    catch(e) { ① }
  } catch(e) { ② }

```



```

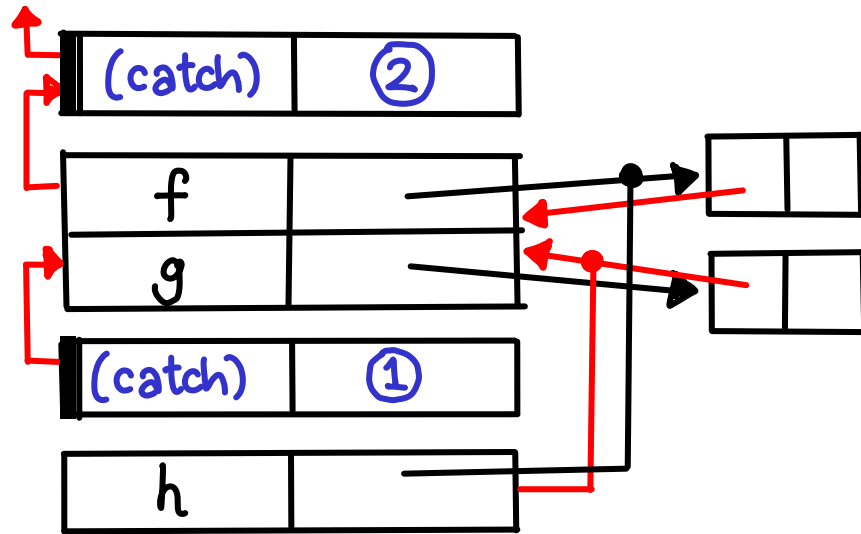
try {
  function f(y) { throw "exn"; }
  function g(h) { try { h(1); }
                  catch(e) { ③ } }
}

```

```

try {
  g(f);
} catch(e) { ① }
} catch(e) { ② }

```

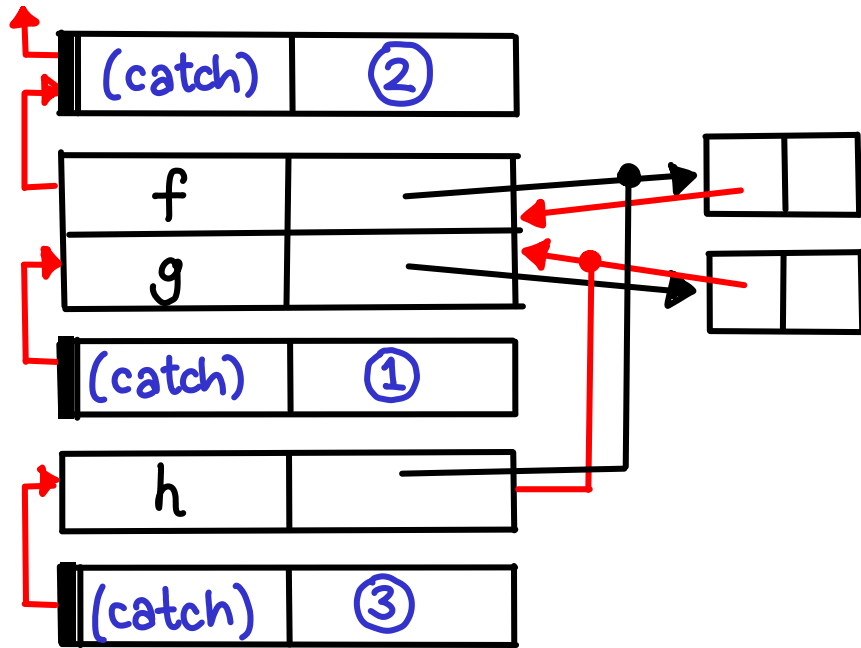


```

try {
  function f(y) { throw "exn"; }
  function g(h) { try { h(1); }
                  catch(e) { ③ } }

  try {
    g(f);
  } catch(e) { ① }
} catch(e) { ② }

```

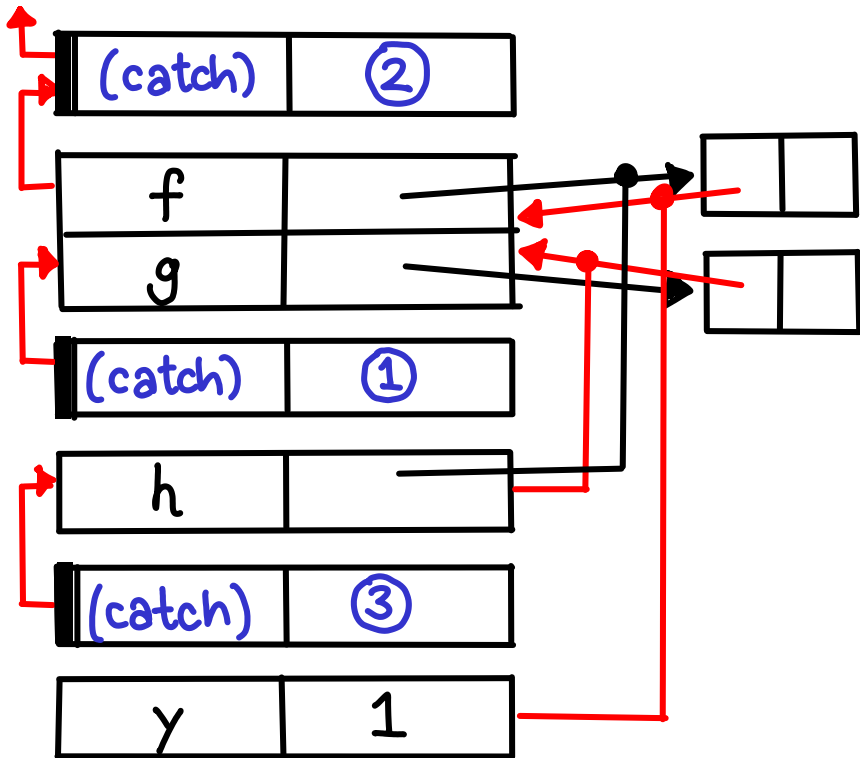


```

try {
  function f(y) { throw "exn"; }
  function g(h) { try { h(1); }
                  catch(e) { ③ } }

  try {
    g(f);
  } catch(e) { ① }
} catch(e) { ② }

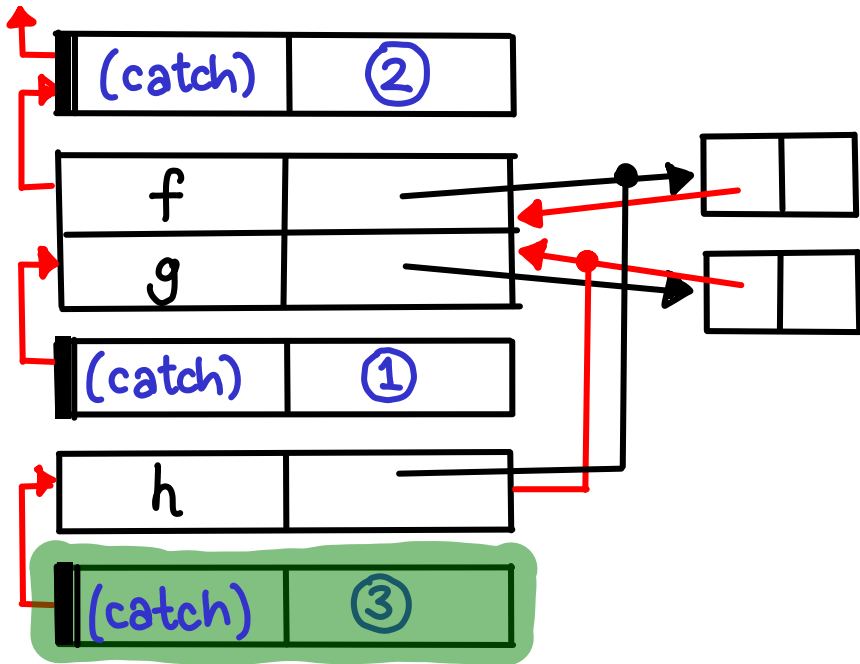
```



```

try {
  function f(y) { throw "exn"; }
  function g(h) { try { h(1); }
    catch(e) { ③ } }
  try {
    g(f);
  } catch(e) { ① }
} catch(e) { ② }

```



```

try {
  function f(y) { throw "exn"; }
  function g(h) { try { h(1); }
                  catch(e) { ③ } }

```

```

try {
  g(f);
} catch(e) { ① }
} catch(e) { ② }

```

One thing that's worth remarking is that which exception handler catches are particular exception is dictated by dynamic scoping rules.

DYNAMIC

try {

function f(y) { throw "exn"; }

function g(h) { try { h(1); }

catch(e) { ③ } }

try {

g(f);

} catch(e) { ① }

} catch(e) { ② }

Exceptions do NOT follow
lexical scoping!

(Is there a control flow
structure which does follow
lexical scoping? Hmmmmm)

LEXICAL

```
{ let e = 2;
```

```
  function f(y) { print(e); }
```

```
  function g(h) { {let e = 3; h(1); }
```

```
{ let e = 1;
```

```
  g(f);
```

```
}
```

```
}
```

(Just to drive the point home;
here's something that is lexically
scoped.)

LEXICAL

```
{ let e = 2;  
  function f(e, y) { print(e); }  
  function g(e, h) { { let e = 3; h(e, 1); }  
}
```

```
{ let e = 1;  
  g(e, f);  
}
```

```
}
```

Here's a cute trick to make the scoping dynamic: just add a parameter for the variable in question to all the functions. This trick will come back!

DYNAMIC

Continuations

(it's more likely than
you think!)

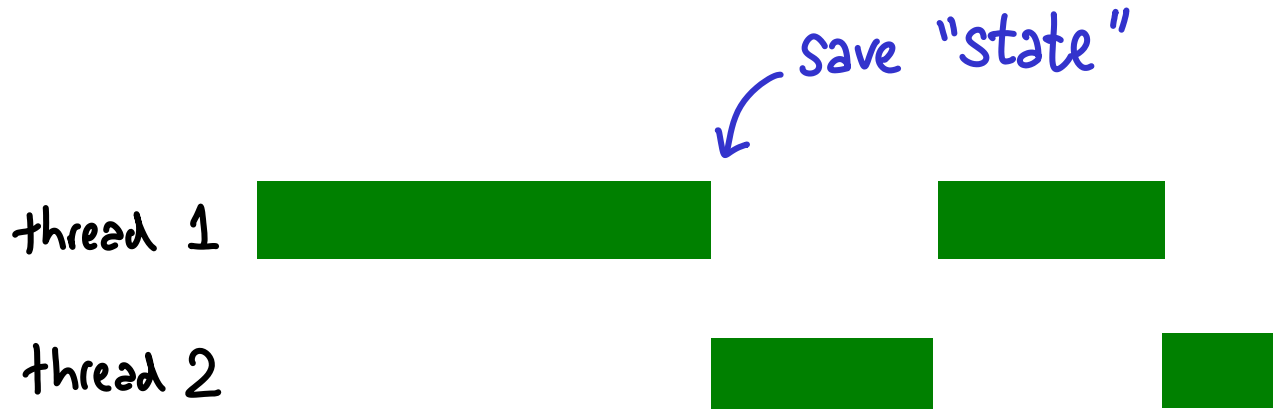
Example: "Async" programming

```
function getPhoto(tag, handlerCallback) {  
    asyncGet(requestTag(tag), function(photoList) {  
        asyncGet(requestOneFrom(photoList),  
            function(photoSizes) {  
                handlerCallback(sizesToPhoto(photoSizes));  
            });  
    });  
}
```

A few examples are in order. Perhaps the most practical example is an interesting historical accident, where in order to perform "asynchronous" computation (non-blocking computation), many programming languages have found themselves FORCING their users to write their code in continuation passing style. If you've ever written code with callbacks, the CALLBACK is representing a CONTINUATION.

<http://elm-lang.org/learn/Escape-from-Callback-Hell.elm>

Example: Cooperative Multithreading



What is really being achieved when you write an explicit callback is you are implementing cooperative multithreading in a program. The callback is a way for a thread of execution to "save its current state" and let other code run while it is waiting for its result. This is in contrast to blocking code, which doesn't save its state and just hogs the entire CPU while waiting. Languages with proper concurrency internally implement this "state saving", either by leaning on the operating system's multiplexing capabilities (here the state is the registers) or by tracking the relevant state in the runtime (green threads.)

Example: GUI/Web Programming

```
waitForButton();  
firstOperation();  
sleep(1000);  
secondOperation();  
  
button.onclick = function(e) {  
    firstOperation();  
    setTimeout(function() {  
        secondOperation();  
    }, 1000);  
}
```

Asynchronous computation abounds in user interface programming, especially on the web. Once again, the callback is the primary mechanism by which you can say what should happen IF some even happens. (The left hand piece of code is the blocking version of something nearly equivalent.)

Example: Debugger

```
...  
■ var x = y + 3  
...
```

Hit Breakpoint

x		2
y		4

The ability to handle continuations is quite powerful. Your debugger is one tool which builds off the power of this capability. When you set a breakpoint in a debugger and execution pauses, you have available the continuation for the program, if you so wish to continue its execution.

What is a continuation?

Continuation-passing Style

Implementing control flow with continuations

There are a few ways to come to understand continuations. The strategy I'm going to take today is talk about continuations in abstract during the execution of an otherwise unremarkable programming language: to get you thinking about how the code you write **IMPLICITLY** manages the future of its computation. From there, we'll talk about a mechanical technique that you can use to reify the continuation of your program as a function, which coincidentally is the same procedure performed when you write code with callback. Then we'll give some examples of how code written in this style can do a variety of interesting control flow patterns.

$$(2 * x + 1 / y) * 2$$

$$(2 * x + 1 / y) * 2$$

1. Multiply 2 and x
2. Divide 1 by y
3. Add ① and ②
4. Multiply ③ and 2

$$(2 * x + 1 / y) * 2$$

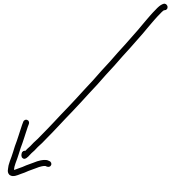
- current computation →
1. Multiply 2 and x
 2. Divide 1 by y
 3. Add ① and ②
 4. Multiply ③ and 2
- } continuation

$$(2 * x + 1 / y) * 2$$

var before = 2 * x;

just a function {
function cont(r) {
 return (before + r) * 2;
}
cont(1 / y)

Continuations
as an **implicit** notion



first-class
Continuations
(**call/cc**)



continuation passing
style to **explicitly**
encode continuations

node.js example

current computation


```
var data = fs.readFileSync("foo.txt")  
console.log(data);  
processData(data);
```



continuation

node.js example

```
fs.readFile("foo.txt", callback)
```

```
function callback(err, data) {
```

```
    var data =
```

```
    console.log(data);
```

```
    processData(data);
```

```
}
```


Continuation Passing Style

2 MAXIMO
NEVER USE RETURN

Some languages have native support for continuations (they tend to be Schemes), but in most languages, you're not actually given a way to get your grubby hands on the "continuation" of a program. The way we can reify program control so it's available is the CPS transformation.

CPS is founded upon INVERSION OF CONTROL. A value is not something to be passed along to a program continuation; rather, the continuation is passed TO THE VALUE, which is responsible for invoking the continuation with the value.

Don't call me: I'll call you!

original

CPS

```
function zero() {  
  return Ø;  
}
```

original

```
function zero() {  
  return 0;  
}
```

CPS

current
continuation ↓

```
function zero(cc) {  
}
```

original

```
function zero() {  
  return ∅;  
}
```

CPS

current
continuation ↓

```
function zero(cc) {  
  cc(∅);  
}
```

↑
call the continuation
with the return value

original

CPS

```
function fact(n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}
```

original

```
function fact(n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}
```

CPS

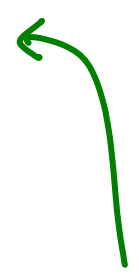
```
function fact(n, cc) {  
  if (n == 0) {  
    } else {  
    }  
}
```

original

```
function fact(n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}
```

CPS

```
function fact(n, cc) {  
  if (n == 0) {  
    cc(1);  
  } else {  
    }  
}
```




as before

original

```
function fact(n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}
```

CPS

```
function fact(n, cc) {  
  if (n == 0) {  
    cc(1);  
  } else {  
    cc(... fact ...)   
  }  
}
```

fact doesn't
return

original

```
function fact(n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}
```

CPS

```
function fact(n, cc) {  
  if (n == 0) {  
    cc(1);  
  } else {  
    fact(n-1, ...cc...);  
  }  
}
```

original

```
function fact(n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}
```

CPS

```
function fact(n, cc) {  
  if (n == 0) {  
    cc(1);  
  } else {  
    fact(n-1, function(r) {  
      cc(r * n);  
    });  
  }  
}
```

original

```
function twice(f, x) {  
  return f(f(x));  
}
```

CPS

```
function twice(f, x, cc) {  
  f(x, function(r) {  
    f(r, cc);  
  });  
}
```

Triangle of
DOOM

original

```
function twice(f, x) {  
  var r = f(x);  
  return f(r);  
}
```

CPS

```
function twice(f, x, cc) {  
  f(x, function(r) {  
    f(r, cc);  
  });  
}
```

The rules

function (x) { \Rightarrow function (x, cc) {

return x \Rightarrow cc(x)

var r = g(x); \Rightarrow g(x, function(r) {
stmts translated stmts
});

Do-notation CPSES for you!

$$\text{do } \{ x \leftarrow e; s \} \equiv e \gg \backslash x \rightarrow \text{do } \{ s \}$$

$$\text{do } \{ e; s \} \equiv e \gg \text{do } \{ s \}$$

$$\text{do } \{ e \} \equiv e$$

$$\begin{aligned} \text{let } m \gg f &= \backslash cc \rightarrow m (\backslash r \rightarrow f r cc) \\ \text{return } x &= \backslash cc \rightarrow cc x \end{aligned}$$

Do-notation CPSes for you!

$$\text{do } \{ x \leftarrow e; s \} \equiv \lambda cc. e (\lambda x. \text{do } \{ s \} cc)$$

$$\text{do } \{ e; s \} \equiv \lambda cc. e (\lambda _ . \text{do } \{ s \} cc)$$

$$\text{do } \{ e \} \equiv \lambda cc. e cc$$

$$\begin{aligned} \text{let } m \gg f &= \lambda cc \rightarrow m (\lambda r \rightarrow f r cc) \\ \text{return } x &= \lambda cc \rightarrow cc x \end{aligned}$$

Tail call optimization

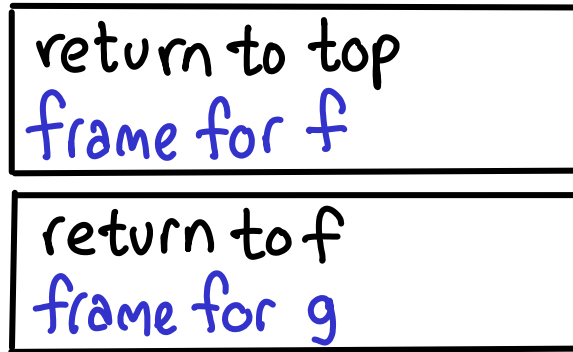
$$f\ x = \text{if } p\ x \text{ then } g\ x$$

$$\text{else } g\ x + 2$$

tail position (points to $g\ x$)
 not tail position (points to $g\ x + 2$)

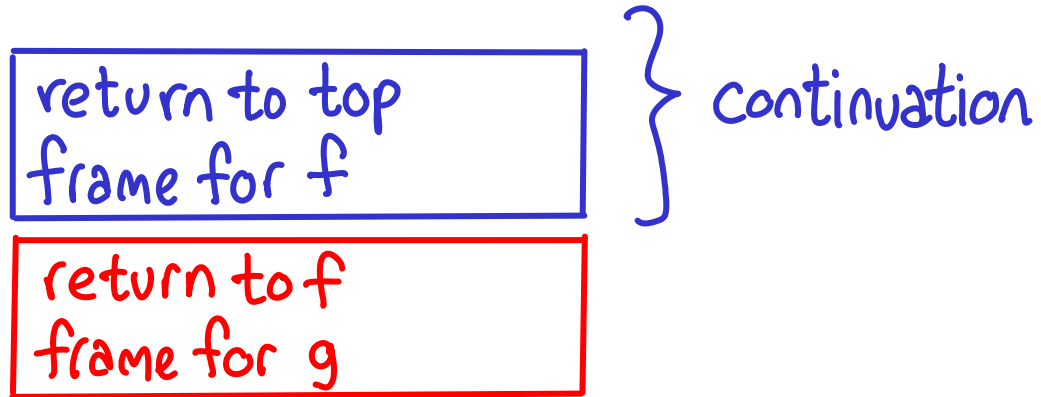
Continuations help us talk compactly about many of the control flow ideas. Consider for example TCO.

Tail call optimization



Stack

Tail call optimization



Stack

Tail call optimization



return to f
frame for g

Stack

TCO removes
stack frame



Tail call optimization

$$f\ x\ cc = \text{if } p\ x \text{ then } g\ x\ (\lambda r. cc\ r) \\ \text{else } g\ x\ (\lambda r. cc\ (r+2))$$

When we CPS it...

Tail call optimization

tail call optimization


$$f\ x\ cc = \text{if } p\ x \text{ then } g\ x\ cc \\ \text{else } g\ x\ (\lambda r. \underline{cc\ (r+2)})$$

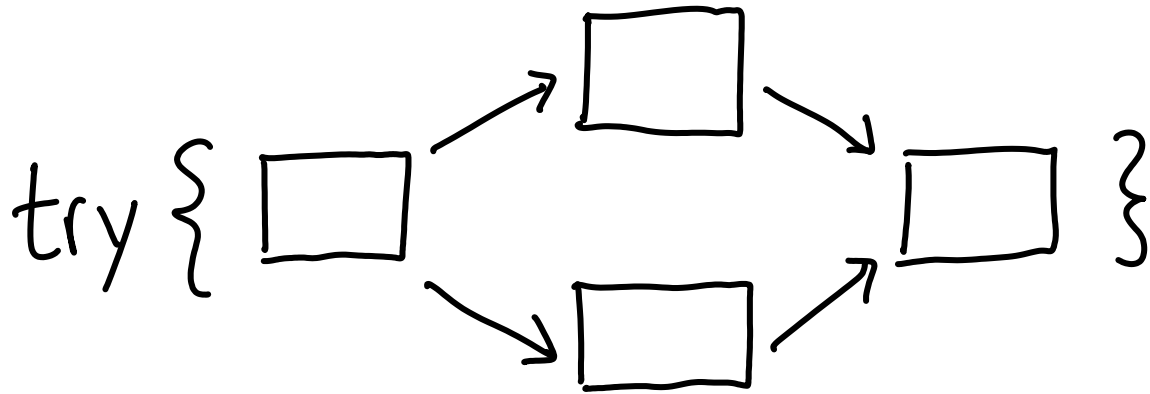
We notice that the TCO corresponds precisely to the eta-reduction of the continuation (which is otherwise just the identity function applied to k !) Cases where we can't eliminate it are when the continuation is a nontrivial extension of k .

By the way, this suggests very interesting optimizations possible when the representation of continuations is intensional. Example:
<http://blog.sigfpe.com/2011/05/fast-forwarding-lrand48.html>

Can't eliminate new
continuation / stack
frame

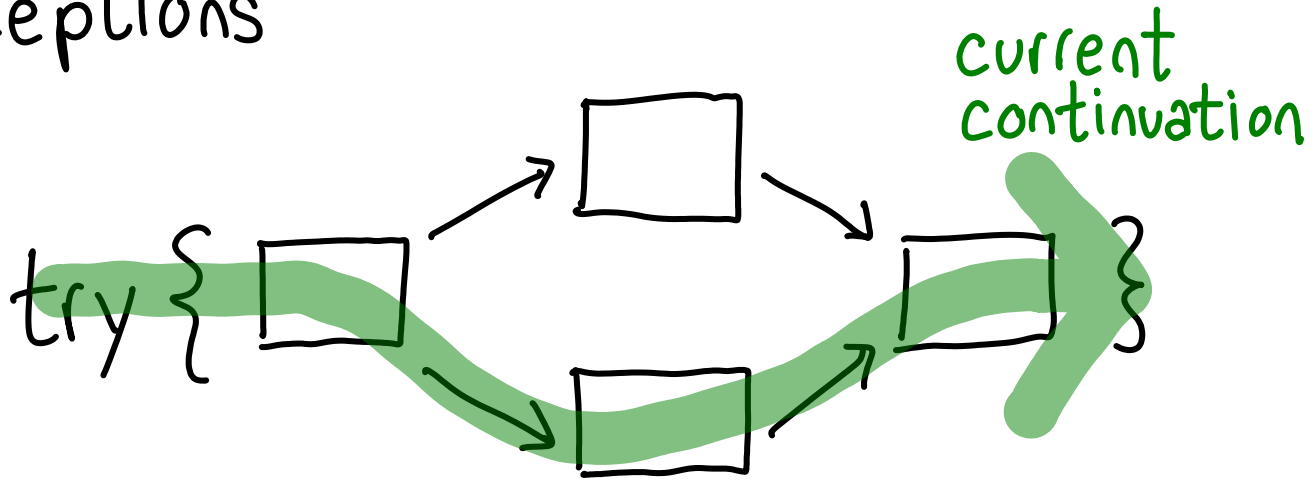


Exceptions



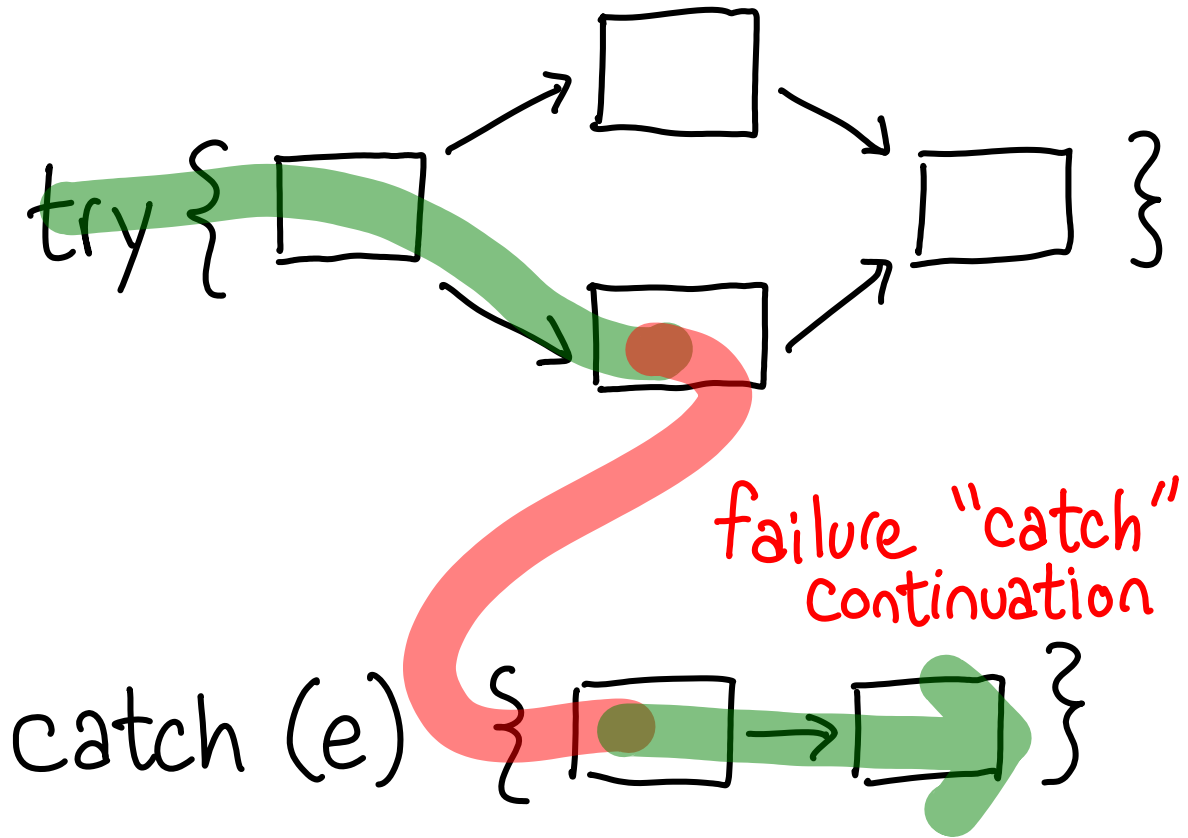
catch (e) { [] → [] }

Exceptions



catch (e) { [] → [] }

Exceptions




```
function f(y) { throw "exn"; }  
try {  
    f(1); console.log("NO");  
} catch(e) { showError(); }  
console.log("YES");
```

Try Statement

```
function f(y) { throw "exn"; }
```

```
try {
```

```
    f(1); console.log("NO");
```

```
} catch(e) { showError(); }
```

```
console.log("YES");
```

← current
continuation

Apply Expression

```
function f(y) { throw "exn"; }  
try {  
    f(1); console.log("NO");  
} catch(e) { showError(); }  
console.log("YES");
```

Apply Expression

composed w/ the
previous current
continuation!

```
function f(y) { throw "exn"; }  
try {
```

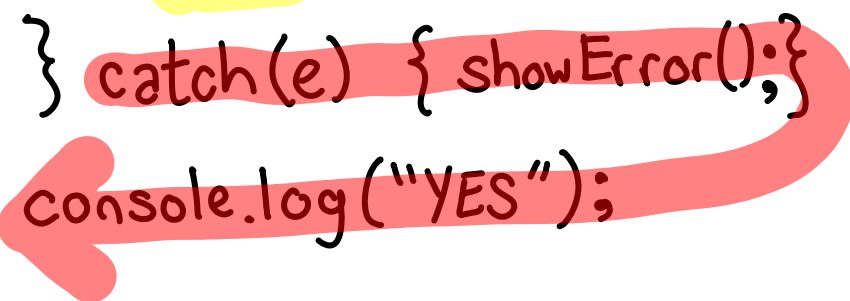
```
  f(1); console.log("NO");  
} catch(e) { showError(); }
```

```
console.log("YES");
```

current continuation

Apply Expression

```
function f(y) { throw "exn"; }  
try {  
  f(1); console.log("NO");  
} catch (e) { showError(); }  
console.log("YES");
```

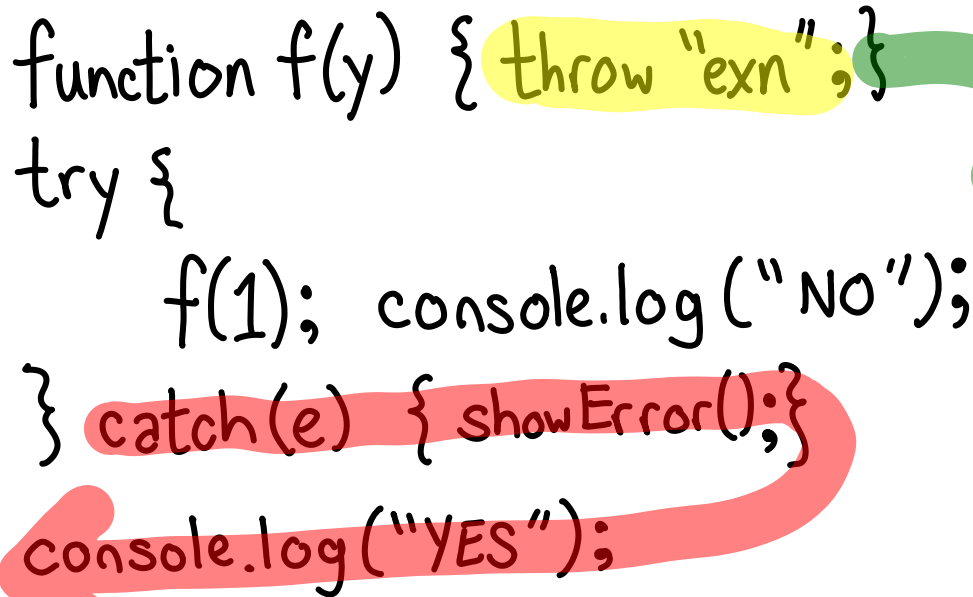


failure ("catch") continuation

Throw Statement

(abbr)
current
continuation

```
function f(y) { throw "exn"; }  
try {  
    f(1); console.log("NO");  
} catch(e) { showError(); }  
console.log("YES");
```



failure ("catch") continuation

CatchClause

```
function f(y) { throw "exn"; }  
try {  
    f(1); console.log("NO");  
} catch(e) { showError(); }
```

```
← console.log("YES");
```

current continuation

CatchClause

failure continuation
restored



```
function f(y) { throw "exn"; }  
try {  
    f(1); console.log("NO");  
} catch(e) { showError(); }
```

```
console.log("YES");
```



current continuation


Finally!

```
try {  
    throw Ø;  
} catch (e) {  
    throw 1;  
} finally {  
    console.log("b");  
}
```

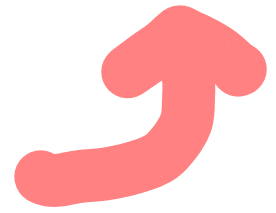
Finally!

```
try {  
    throw 0;  
} catch (e) {  
    throw 1;  
} finally {  
    console.log("b");  
}
```

always runs, no
matter how
we exit scope



Finally!



failure continuation

```
try {  
    throw ∅;  
} catch (e) {  
    throw 1;  
} finally {  
    console.log("b");  
}
```

}



current continuation

Finally!



```
try {  
    throw ∅;
```

```
} catch (e) {  
    throw 1;
```

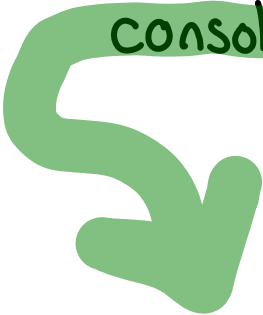
failure continuation

```
} finally {
```

```
    console.log("b");
```

```
}
```

current continuation



Finally!

```
try {  
    throw ∅;  
} catch (e) {  
    throw 1;  
}  
finally {  
    console.log("b");  
}
```

failure
continuation



current continuation



Finally!

```
try {  
    throw 0;  
} catch (e) {  
    throw 1;  
} finally {  
    console.log("b");  
}
```



current
continuation

Finally!

```
try {  
  throw Ø;  
} catch (e) {  
  throw 1;  
} finally {  
  console.log("b");  
}
```

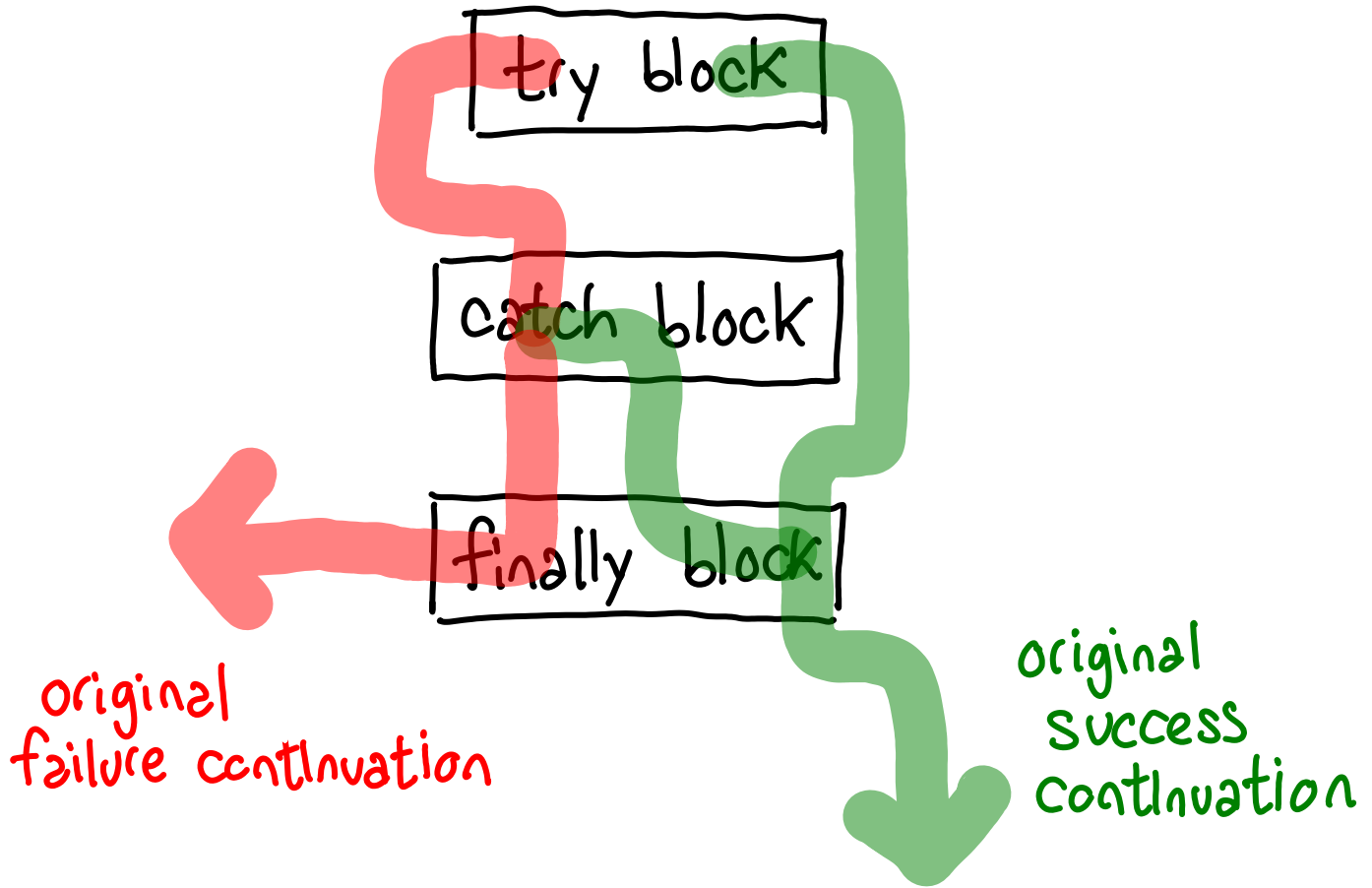
The diagram illustrates the execution flow of the provided code. A red arrow starts at the `throw Ø;` line, loops around to the `throw 1;` line, and then points down to the `finally` block. From the `finally` block, a red arrow points left to the label 'original failure continuation'. A green arrow starts at the `throw 1;` line, loops around to the `finally` block, and then points down to the label 'original success continuation'.

original
failure continuation

original
success
continuation

Finally!

continuations are
generalized goto!



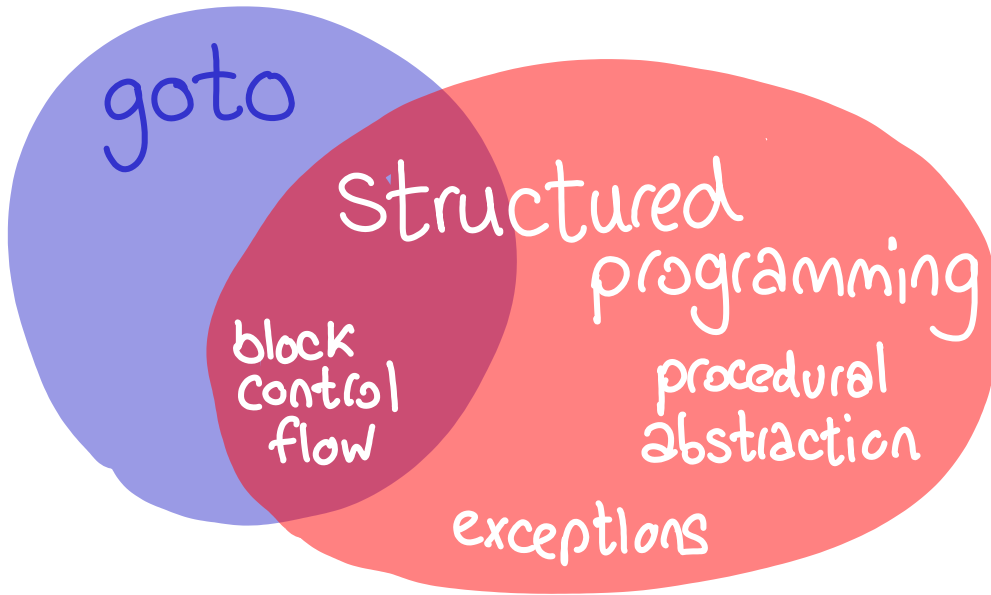
Other examples

- Call with current continuation
- Nondeterministic choice
- Generators
- Coroutines

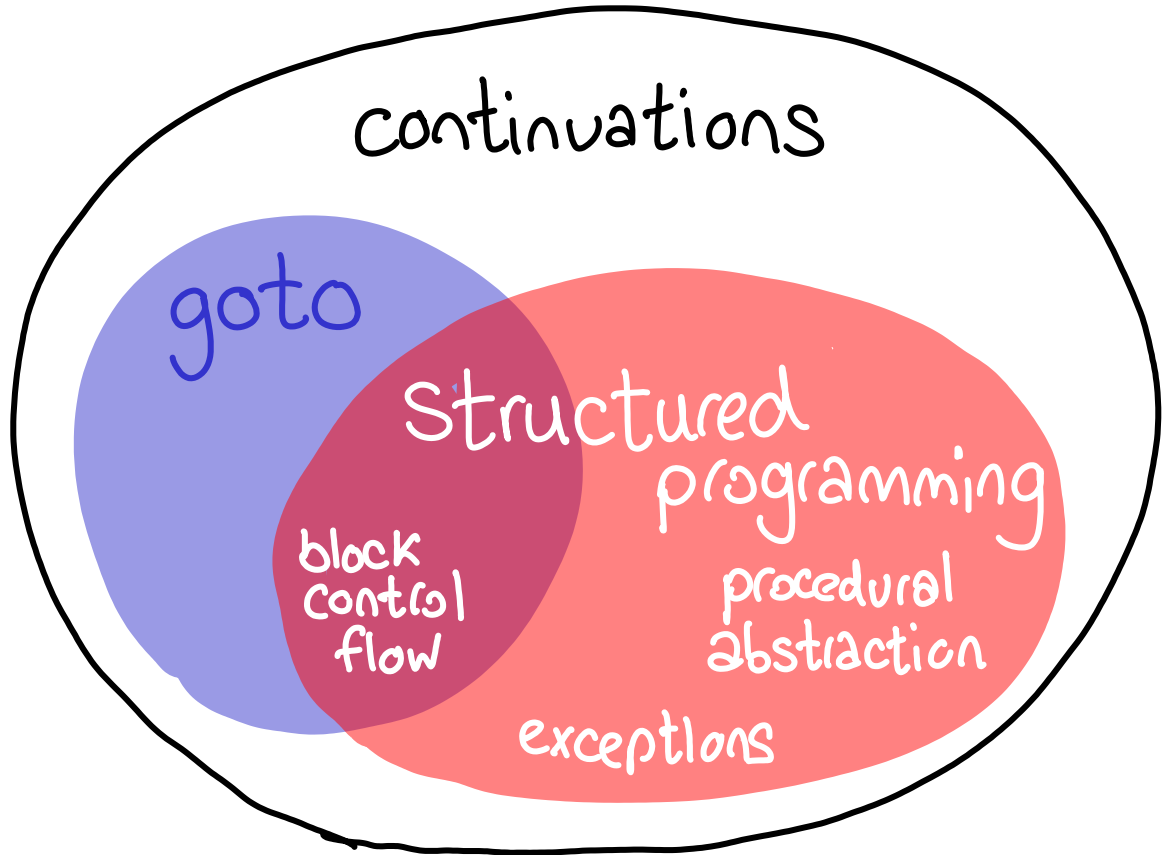
Conclusion

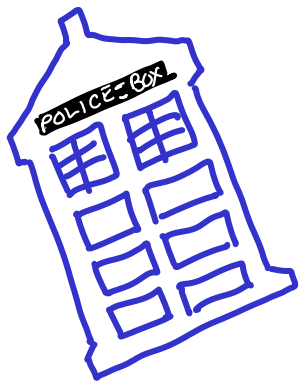


Conclusion



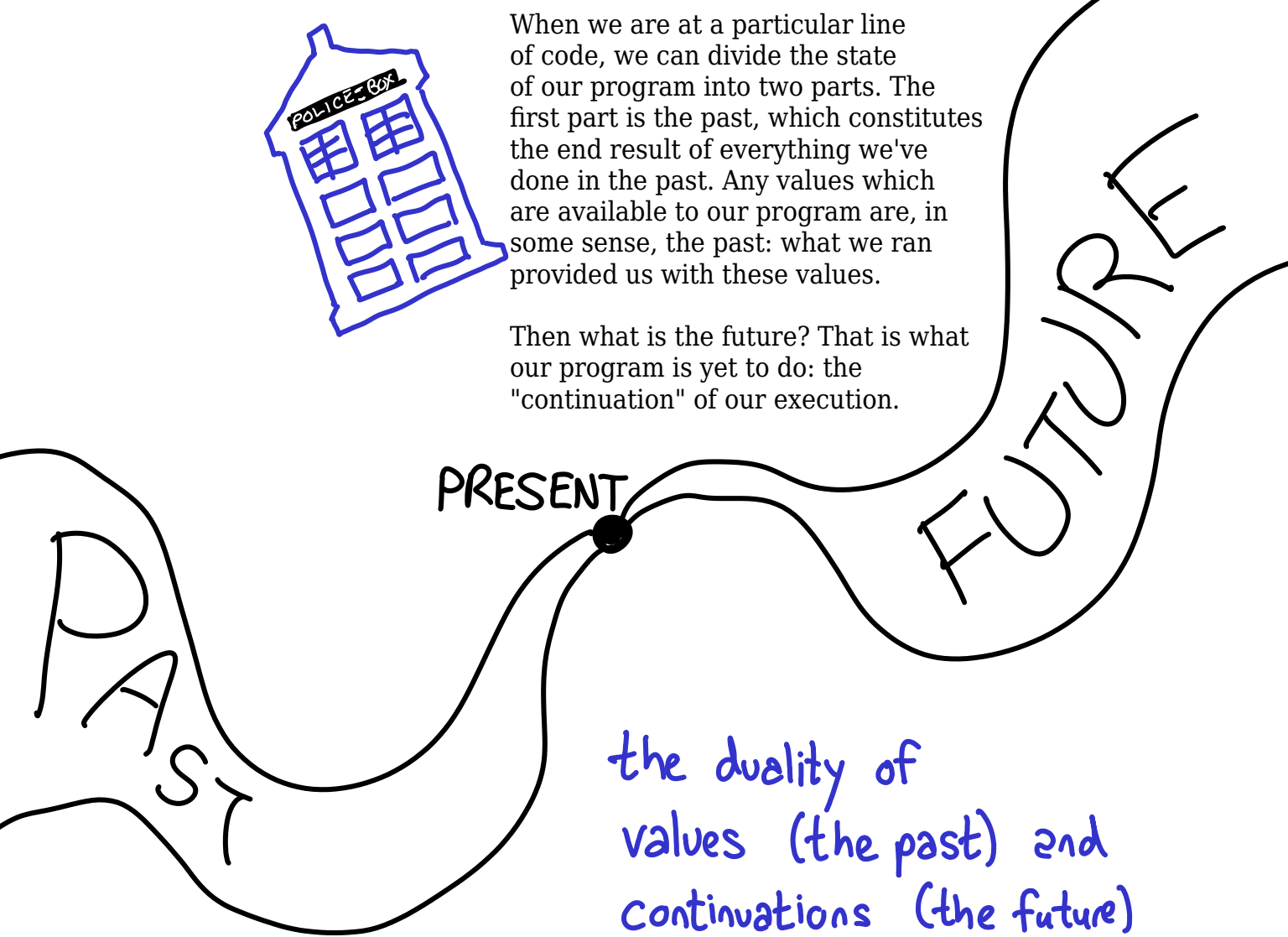
Conclusion





When we are at a particular line of code, we can divide the state of our program into two parts. The first part is the past, which constitutes the end result of everything we've done in the past. Any values which are available to our program are, in some sense, the past: what we ran provided us with these values.

Then what is the future? That is what our program is yet to do: the "continuation" of our execution.



the duality of
values (the past) and
continuations (the future)