Control Flow

Edward Z. Yang

1. Structured Programming

2. Procedural abstraction — the stack

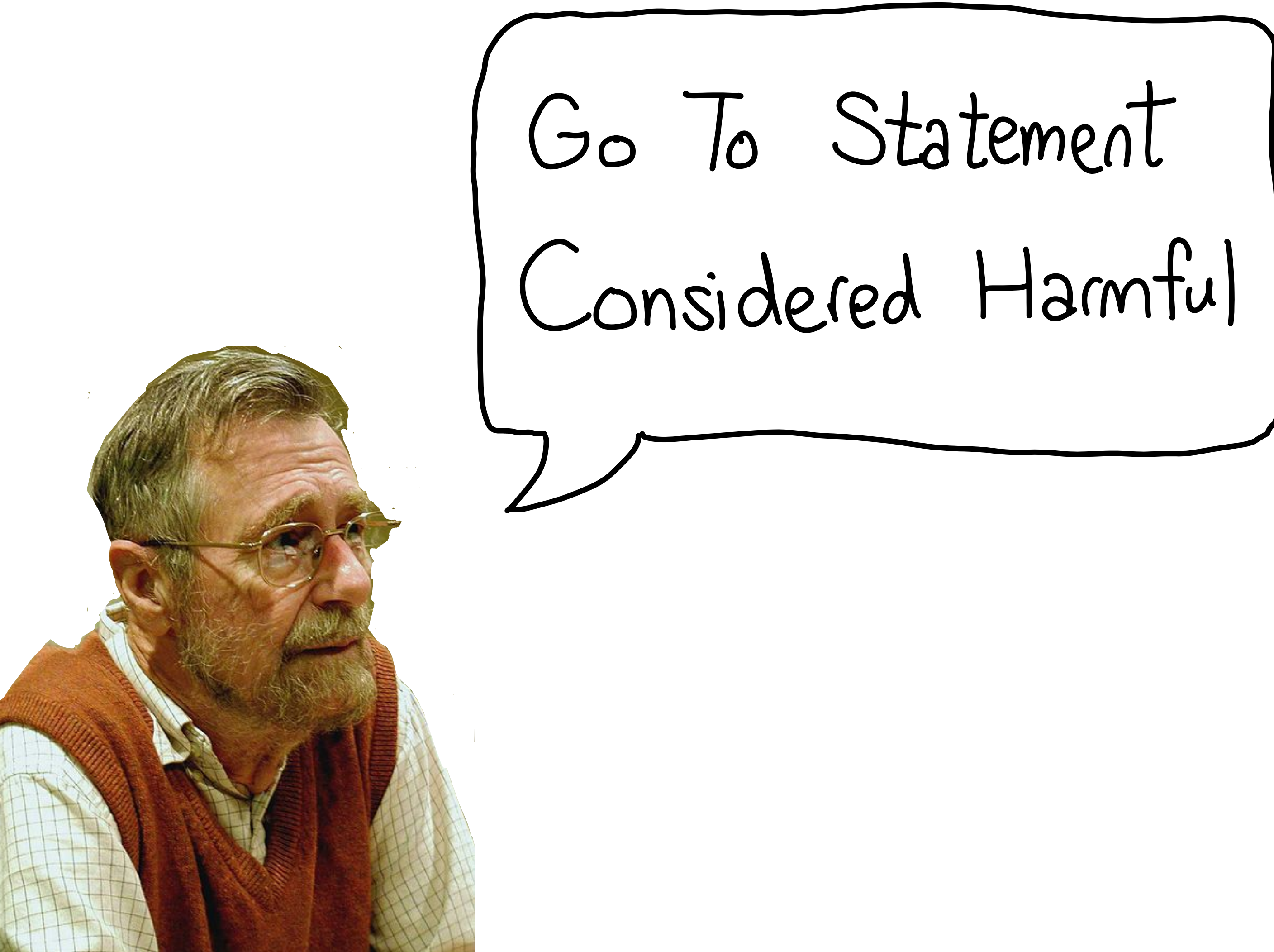
3. Continuations

```
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. 0.00001) GO TO 30
     X = X-Y-Y
30  X = X+Y
     ...
50 CONTINUE
     X = A
     Y = B-A
     GO TO 11
     …
```

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

Structured Programming with go to Statements

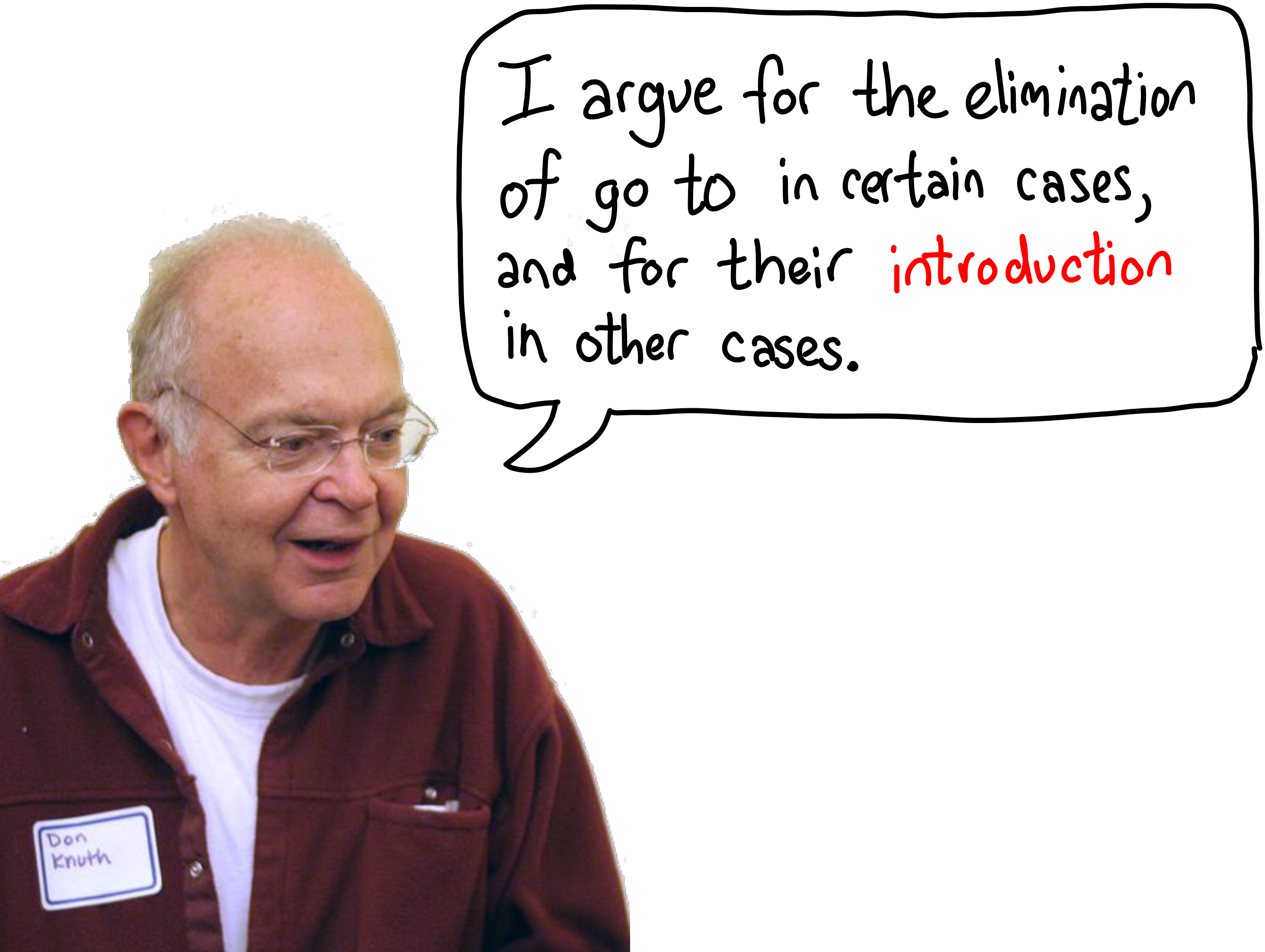
DONALD E. KNUTH

Stanford University, Stanford, California 94305

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.
Don Knuth

Structured Programming

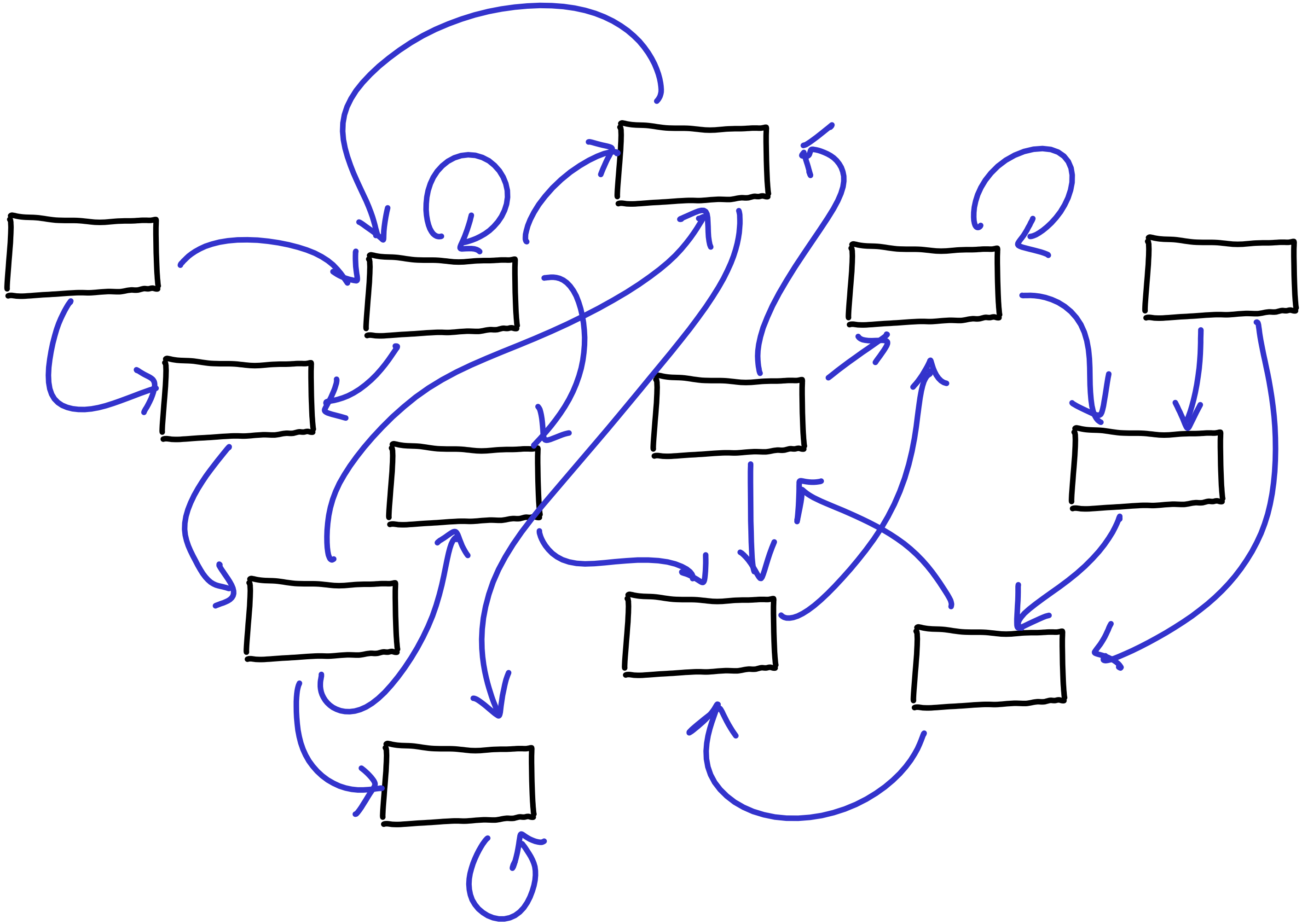
if... then ... else...

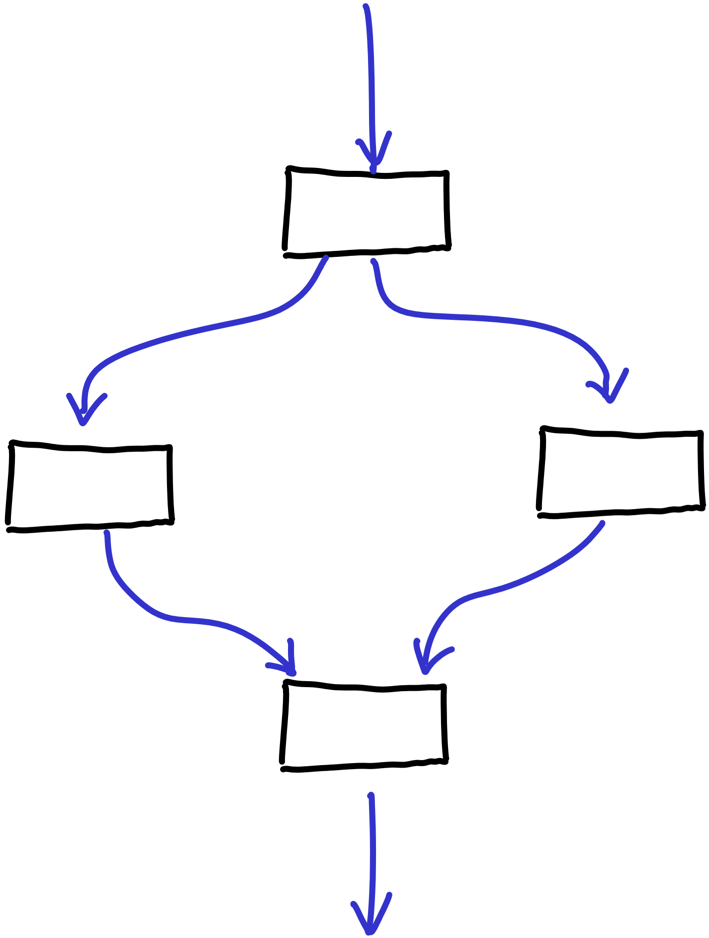
while... do...

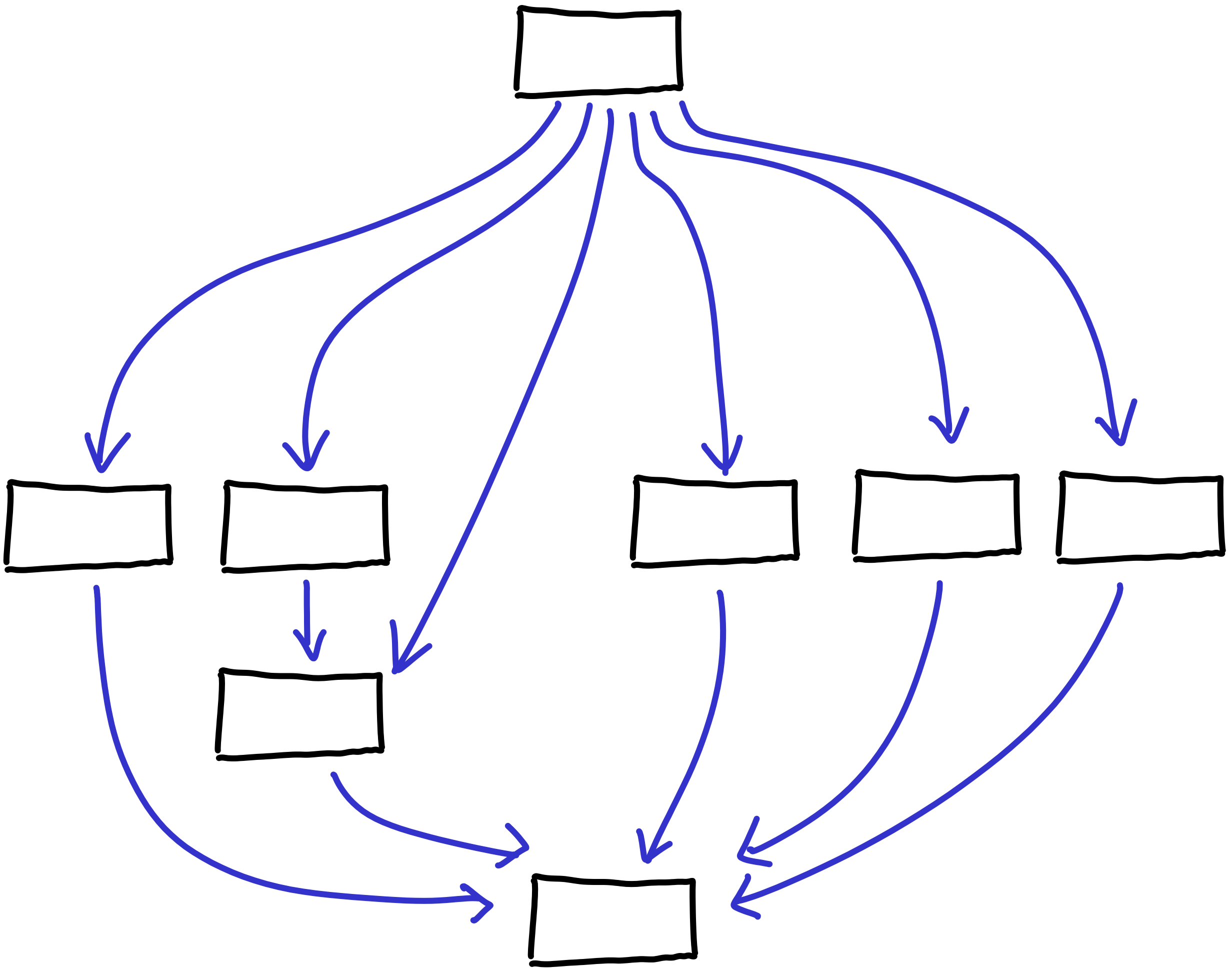
Structured Programming

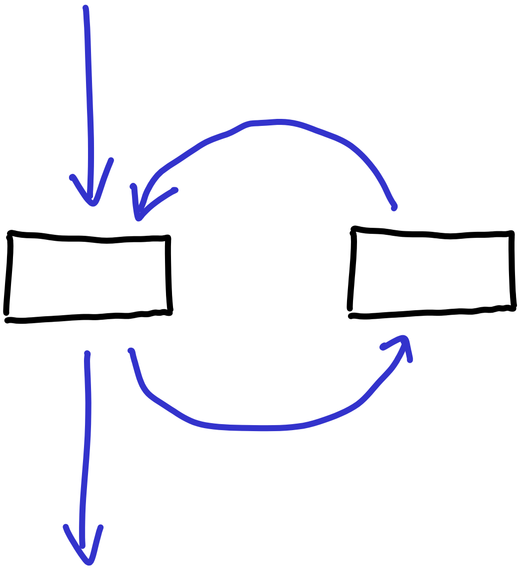
case...

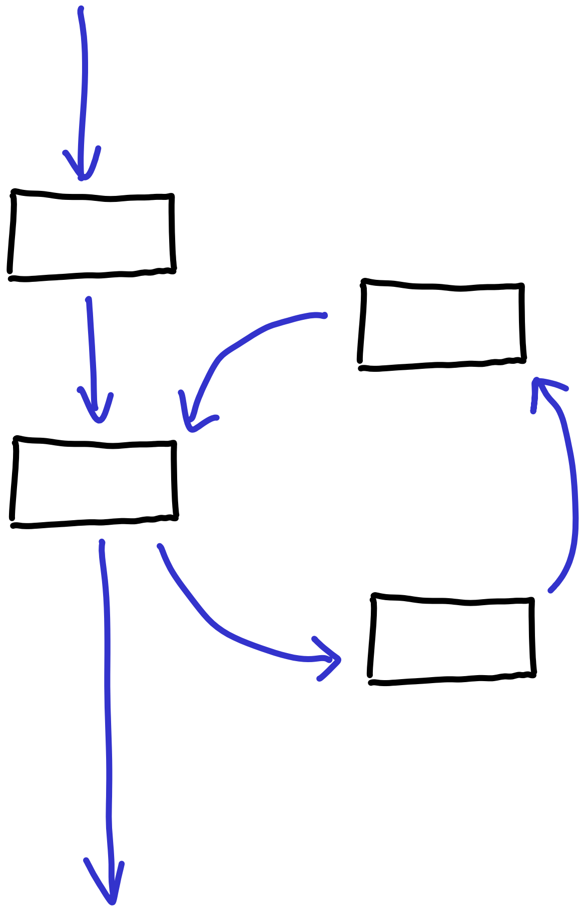
for... { ... }







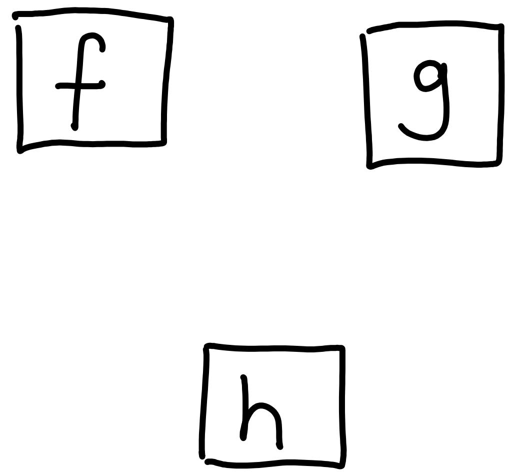




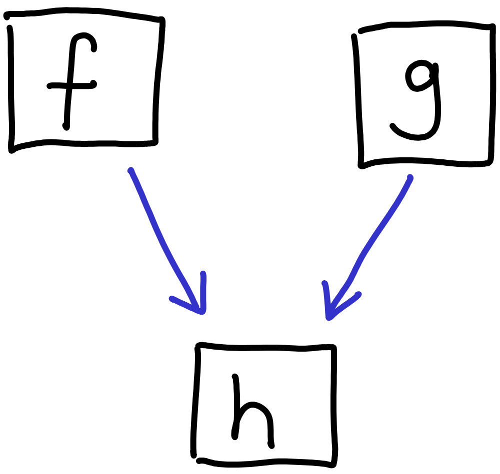
Procedural Abstraction

The Stack

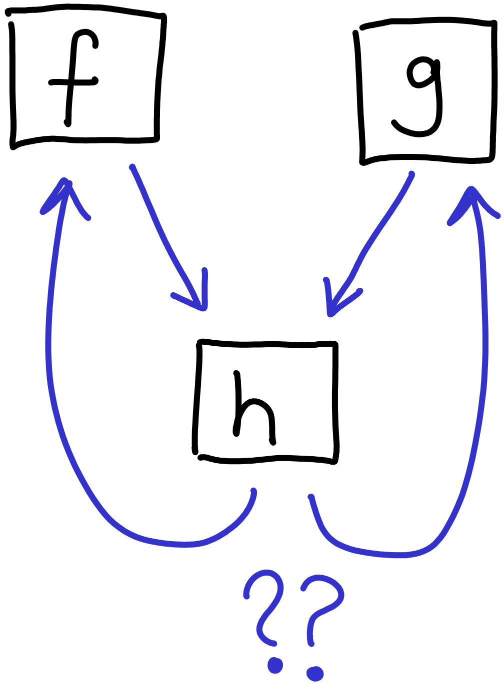
```
function f(x) {
  return h(x) + 1;
}

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

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

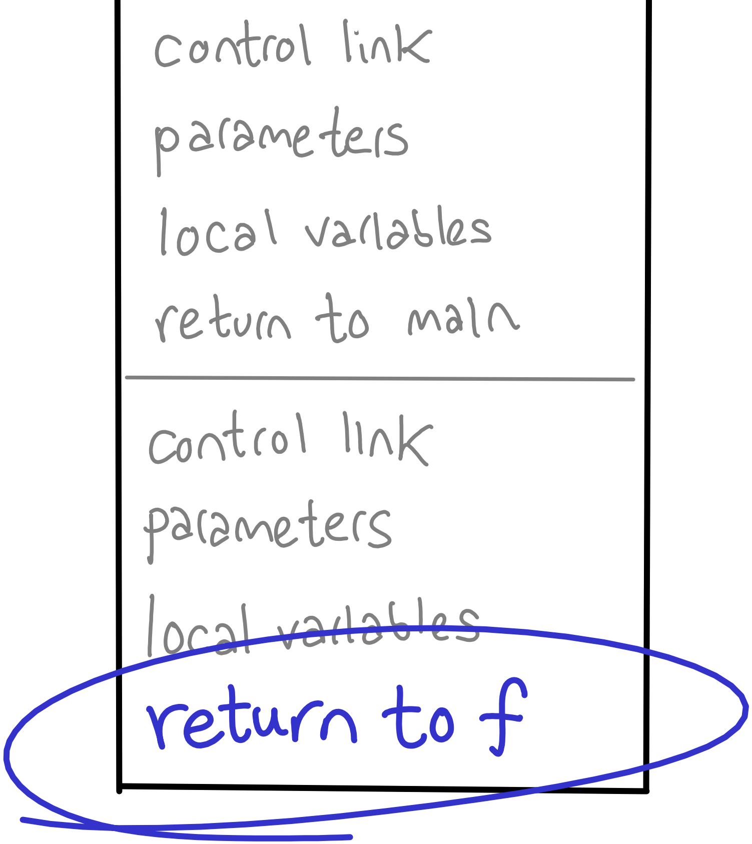


```
function f(x) {
  return h(x) + 1;
}

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

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

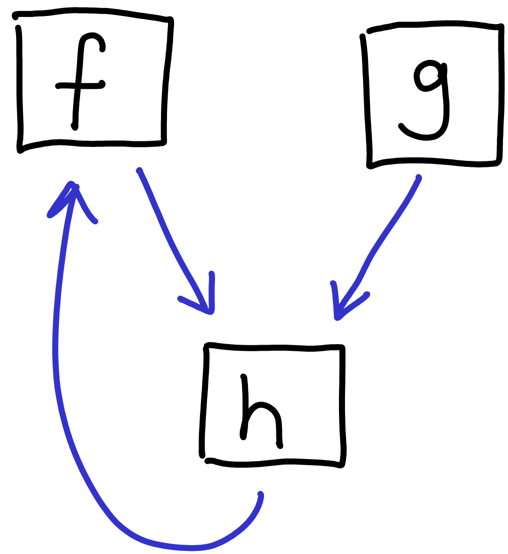


```
function f(x) {
  return h(x) + 1;
}

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

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




control link
parameters
local variables
return to main
control link
parameters
local variables
return to f

Stack!


f
g
h

$\boxed{\text{return}}$ return pointer on the stack

<u>Dynamic</u> control flow

Exceptions

Also The Stack

raise/throw

handler/catch

Dynamic control flow

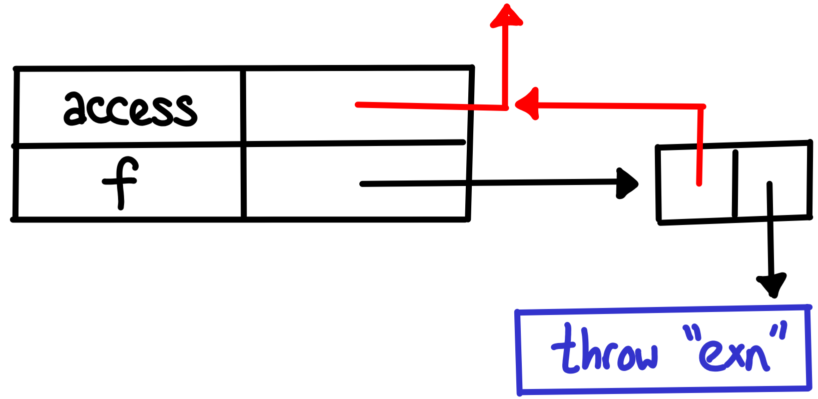
$\boxed{\text{throw}}$

handler frame
on the stack

<u>Dynamic</u> control flow

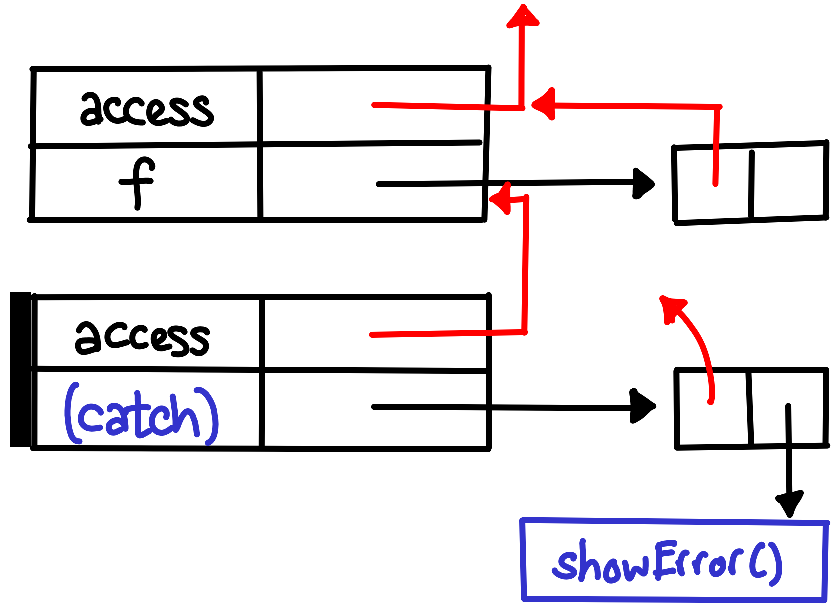
```
function f(y) { throw "exn"; }
try {
    f(1);
} catch(e) { showError(); }
```

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

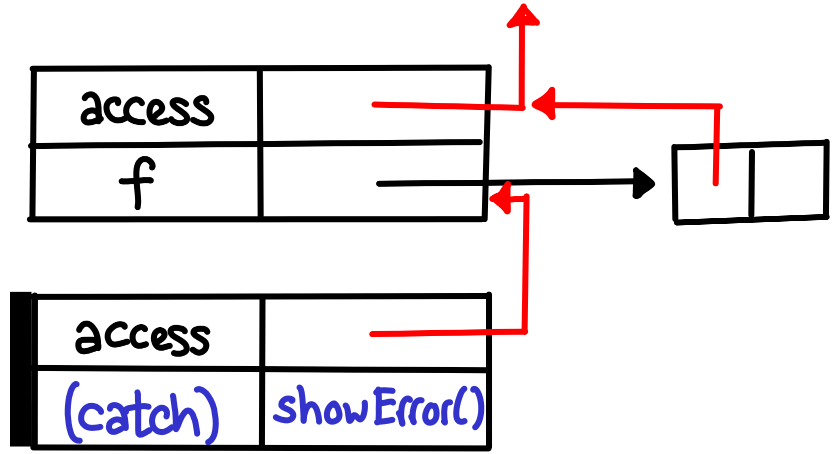



```
function f(y) { throw "exn"; }
try {
    f(1);
} catch(e) { showError(); }
```

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

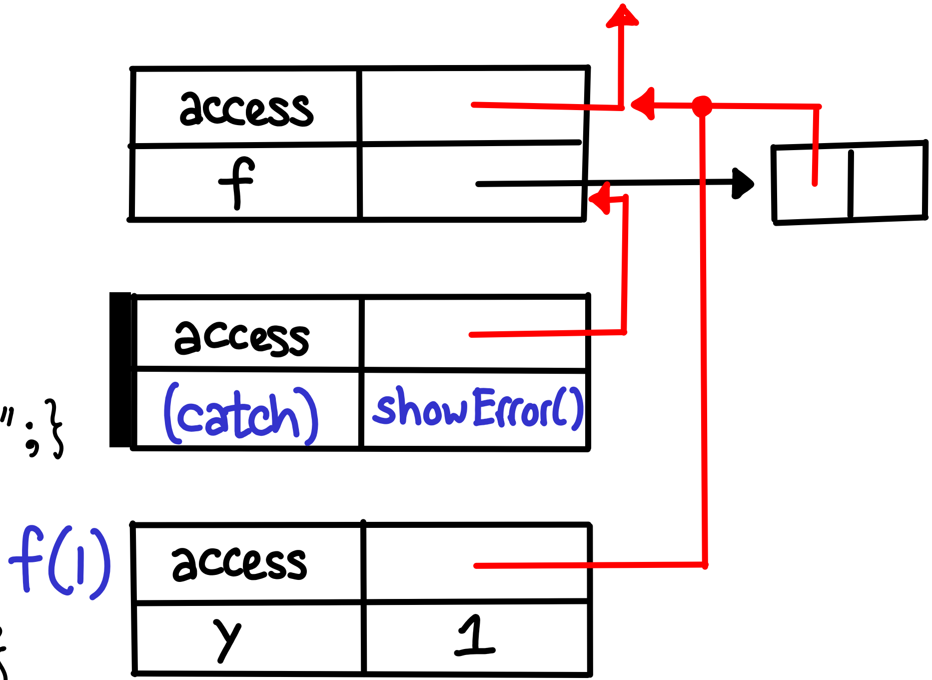


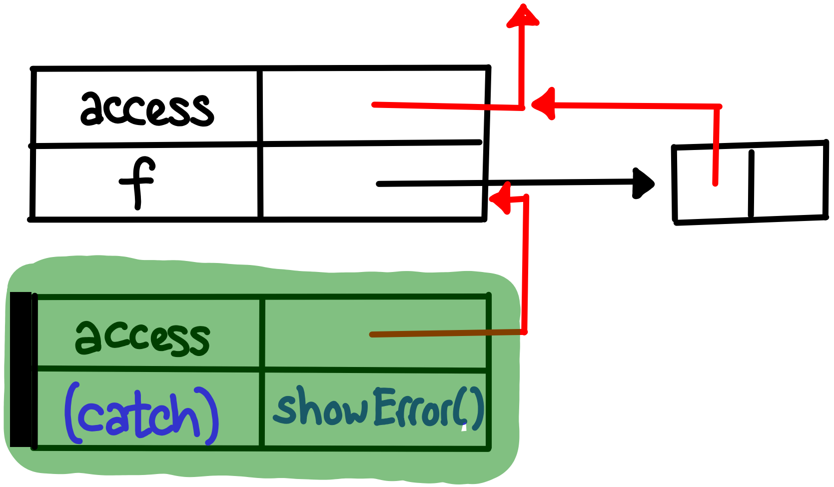
NB: try does not actually introduce scope




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

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

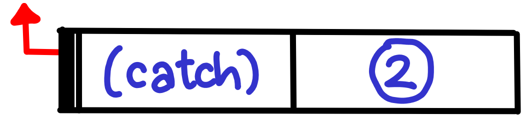




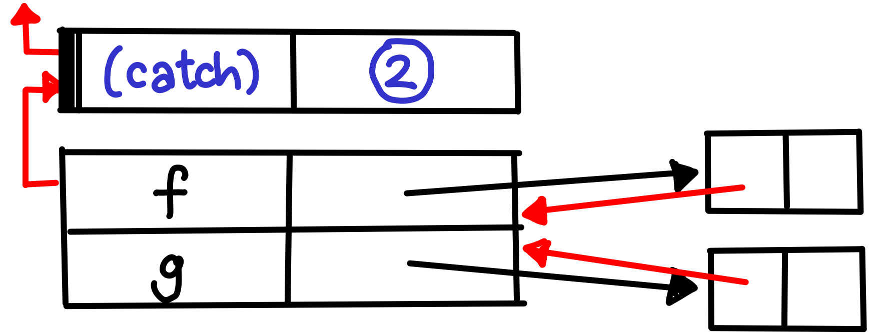

```
function f(y) { throw "exn"; }
try {
    f(1);
} catch(e) { showError(); }
```

```
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 }
```

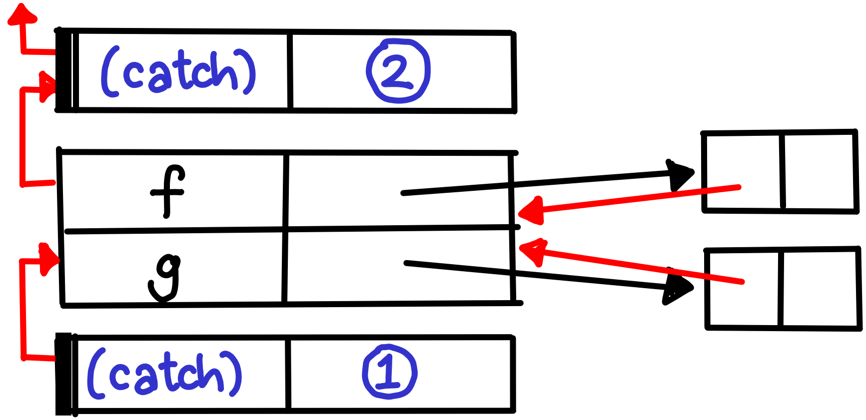


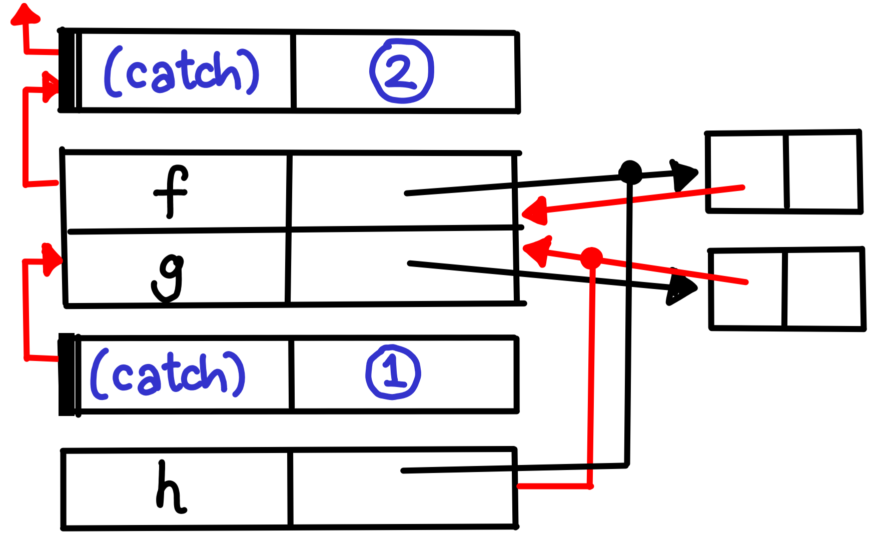

```
try {
    function f(y) { throw "exn"; }
    function g(h) { try { h(1); }
                    catch(e) { 3 } }
    try {
        g(f);
    } catch(e) { 1 }
} catch(e) { 2 }
```




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

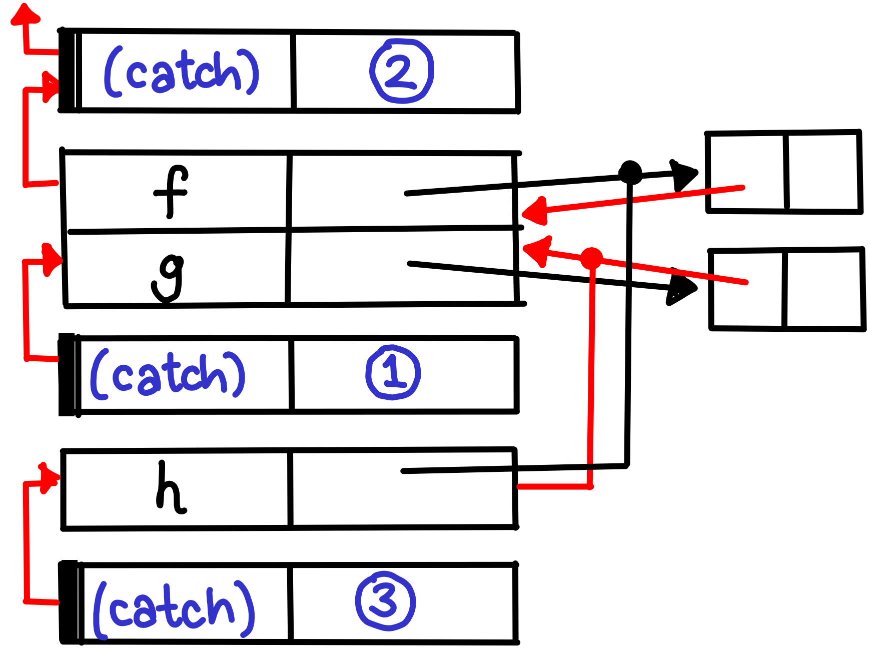
```
try {
    function f(y) { throw "exn"; }
    function g(h) { try { h(1); }
                    catch(e) {3} }
    try {
        g(f);
    } catch(e) { 1 }
} catch(e) { 2 }
```



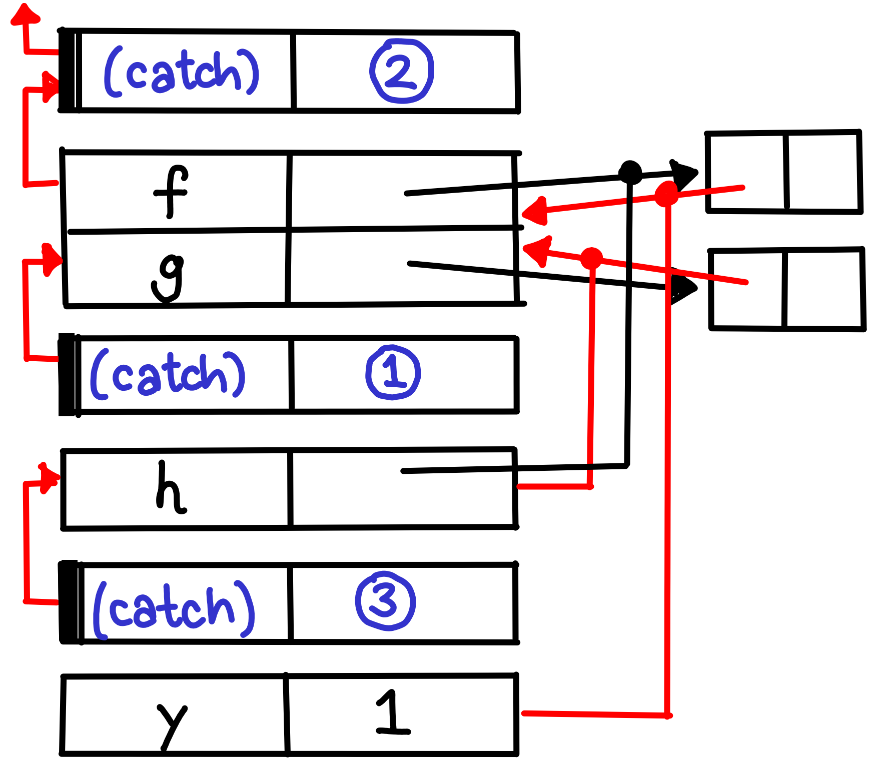



```
try {
    function f(y) { throw "exn"; }
    function g(h) { try { h(1); }
                    catch(e) { 3 } }
    try {
        g(f);
    } catch(e) { 1 }
} catch(e) { 2 }
```

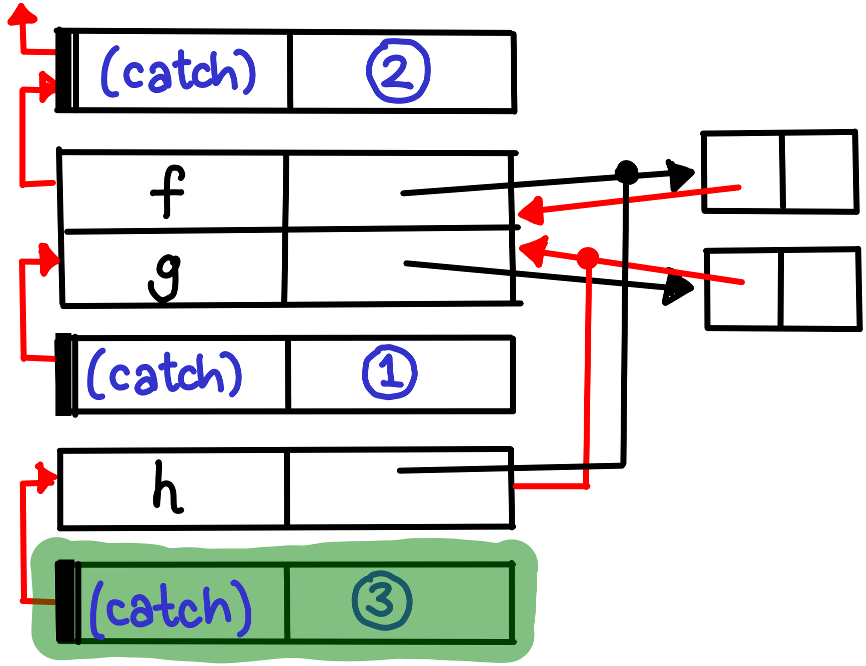
```
try {
    function f(y) { throw "exn"; }
    function g(h) { try { h(1); }
                    catch(e) { 3 } }
    try {
        g(f);
    } catch(e) { 1 }
} catch(e) { 2 }
```



```
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 }
```



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

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 }
```

LEXICAL

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

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

}
```

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);
   }

}
```

DYNAMIC

Continuations

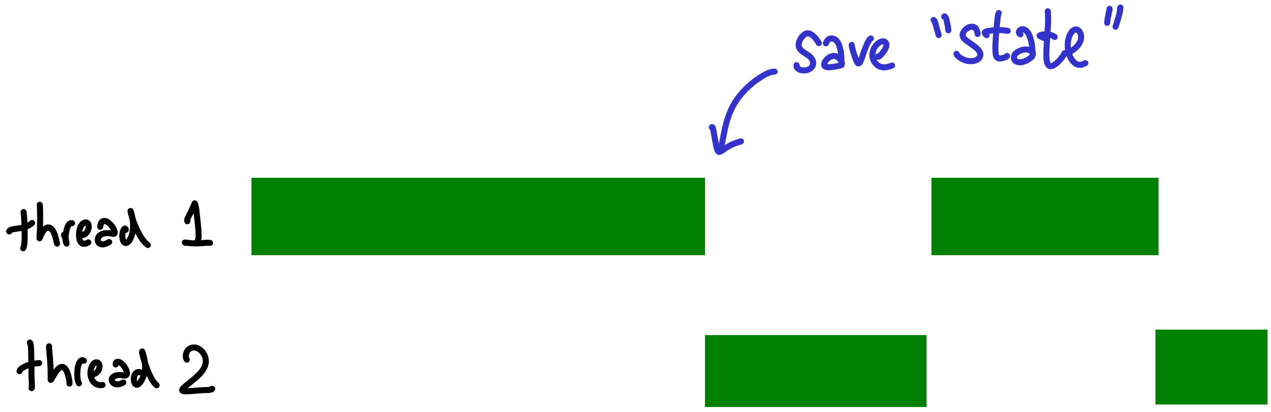
(it's more likely than
you think!)

Example: "Async" programming

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

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

Example: Cooperative Multithreading



Example: GUI/Web Programming

```
waitForButton();
firstOperation();
sleep(1000);
secondOperation();
```

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

Example: Debugger

...

var $x = y + 3$

...

Hit Breakpoint

x	2
y	4

What is a continuation?

Continuation-passing style

Implementing control flow with continuations

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

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

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

MAXIM

NEVER USE RETURN

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

original

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

CPS

Original

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

CPS

current continuation

```
function zero(cc) {

}
```

Original

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

CPS

current continuation

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

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	CPS

`function fact(n) {`
`  if (n==0) {`
`    return 1;`
`  } else {`
`    return n*fact(n-1);`
`  }`
`}`
	`function fact(n, cc) {`
	`  if (n==0) {`
	`    cc(1);` ← as before
	`  } 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 {
    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!

$$\text{do}\,\{\,x \leftarrow e;\, s\,\} \equiv e \gg\!= \backslash x \to \text{do}\,\{s\}$$

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

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

$$\text{let } m \gg\!= f = \backslash cc \to m\,(\backslash r \to f\ r\ cc)$$

$$\text{return } x = \backslash cc \to cc\ x$$

Do-notation CPSes for you!

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

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

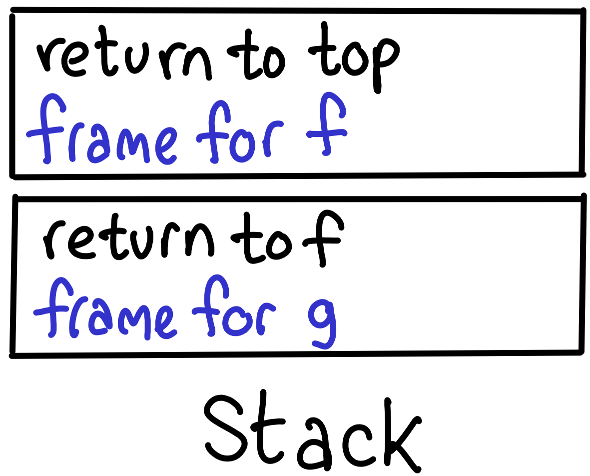
$$\text{do}\{e\} \equiv \lambda cc.\, e\ cc$$

```
let m >>= f  = \cc -> m (\r -> f r cc)
    return x = \cc -> cc x
```

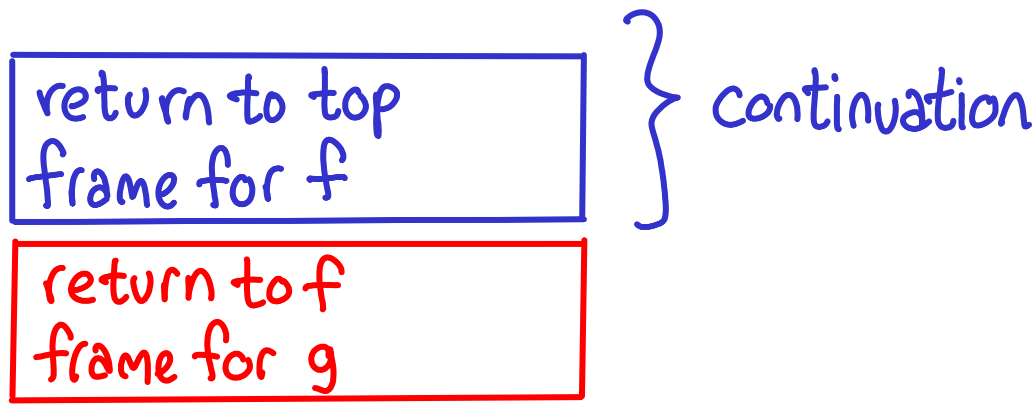
Tail call optimization

$$f\ x = \text{if } p\ x \text{ then } g\ x \quad \text{tail position}$$
$$\text{else } g\ x + 2 \quad \text{not tail position}$$

Tail call optimization



Tail call optimization



Stack

Tail call optimization

return to f
frame for g

Stack

TCO removes stack frame

Tail call optimization

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

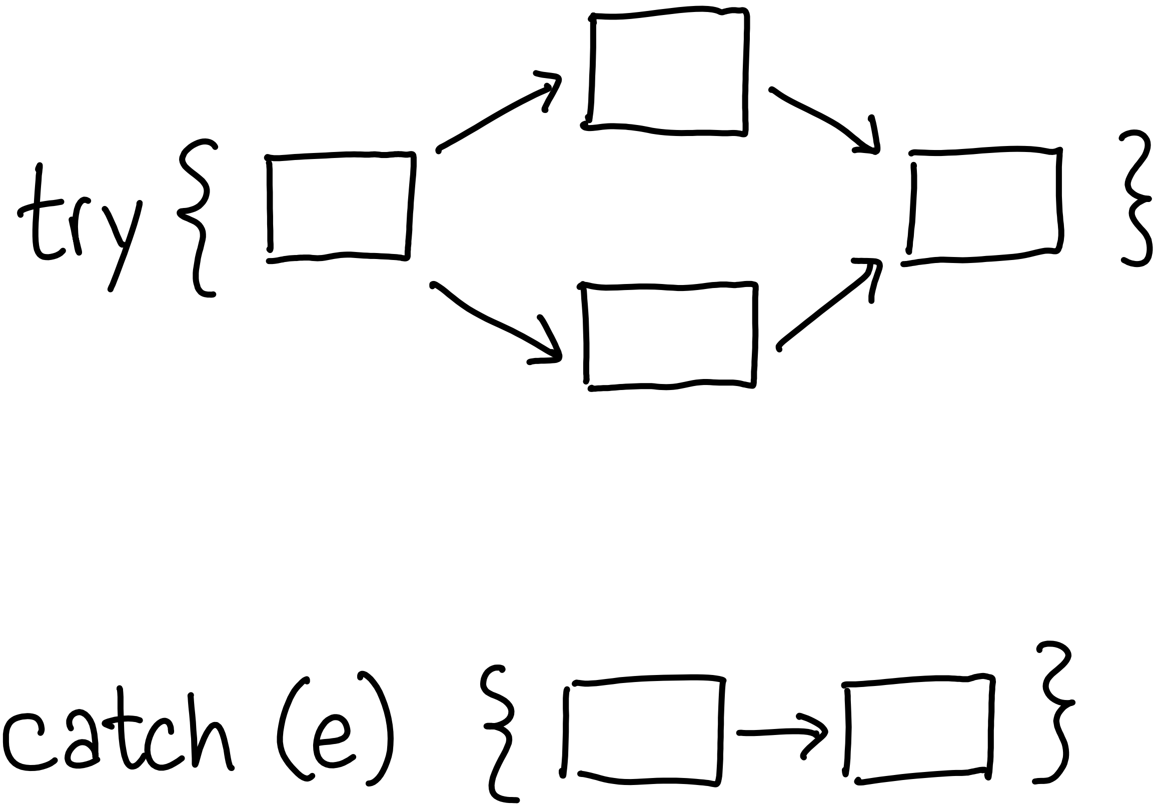
Tail call optimization

tail call optimization

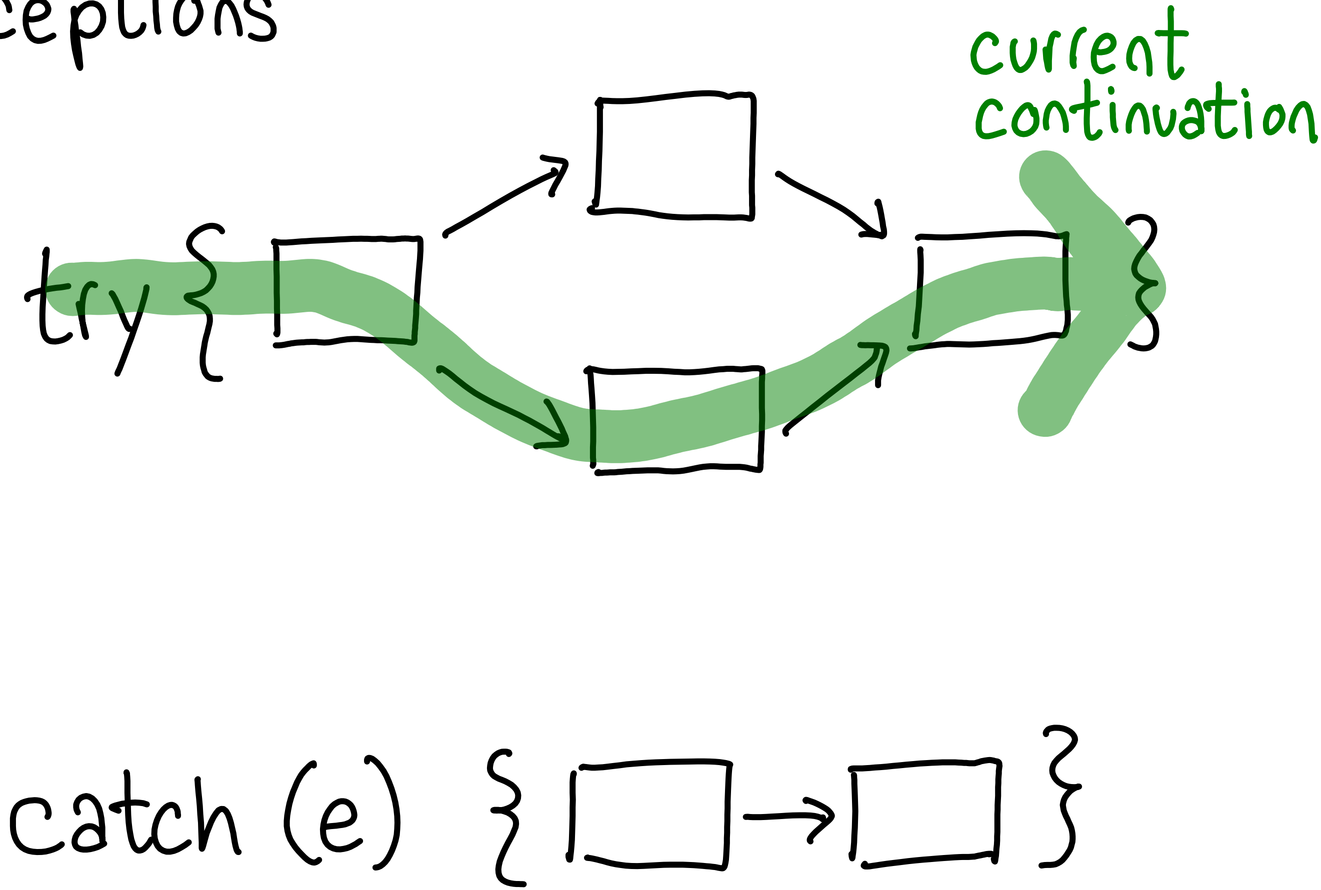
$$f\ x\ cc = \text{if } p\ x \text{ then } g\ x\ cc$$
$$\text{else } g\ x\ (\lambda r.\, cc\,(r+2))$$

Can't eliminate new continuation/stack frame

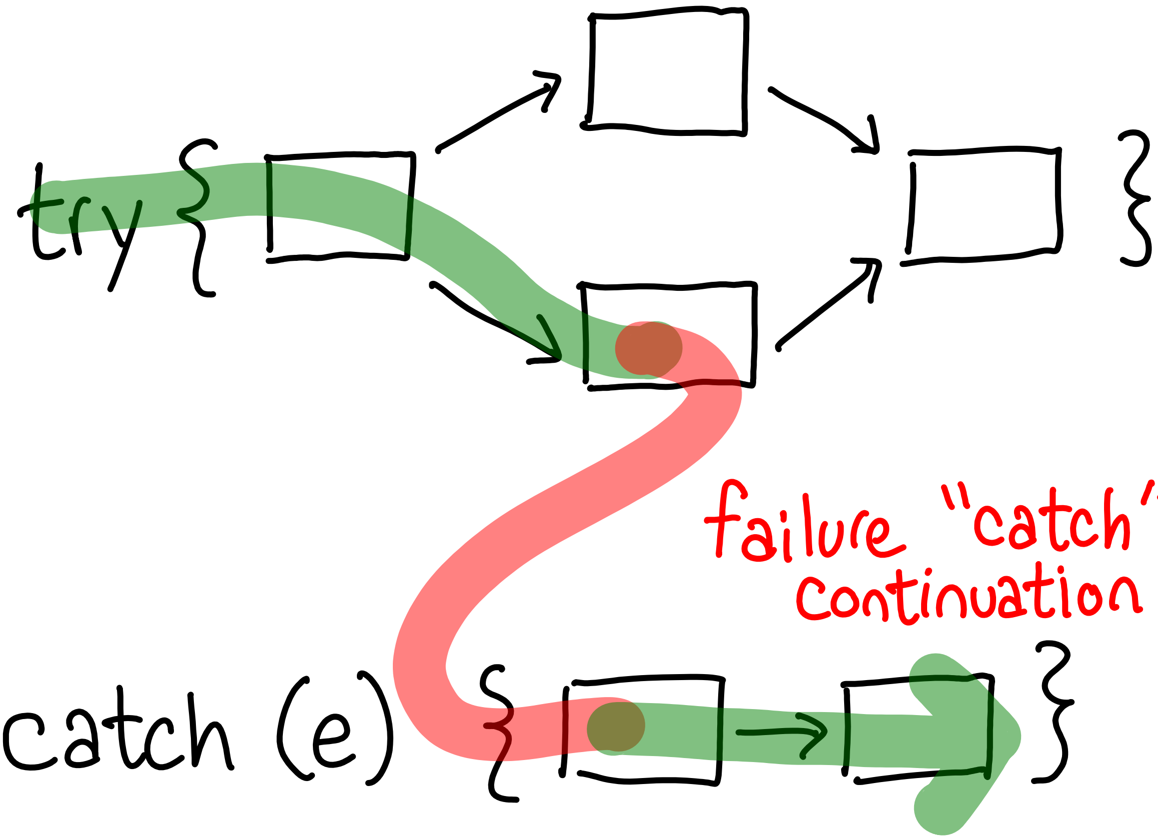
Exceptions



Exceptions



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

ThrowStatement

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

current continuation

Finally!

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

failure continuation

current continuation

Finally!

```
try {
    throw 0;
} 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!

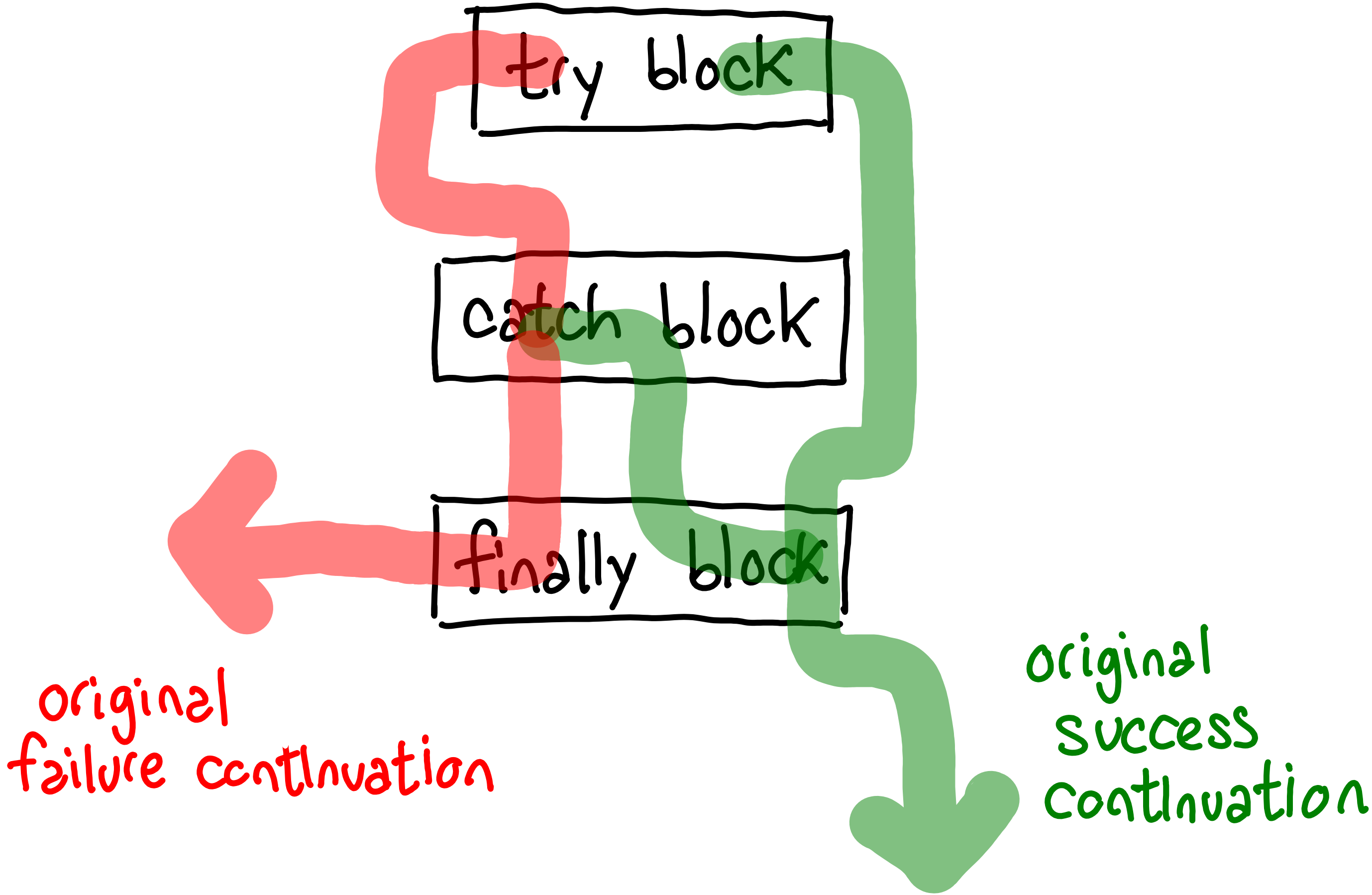
```
try {
    throw 0;
} catch (e) {
    throw 1;
} finally {
    console.log("b");
}
```

original failure continuation

original success continuation

Finally!

Continuations are generalized goto!

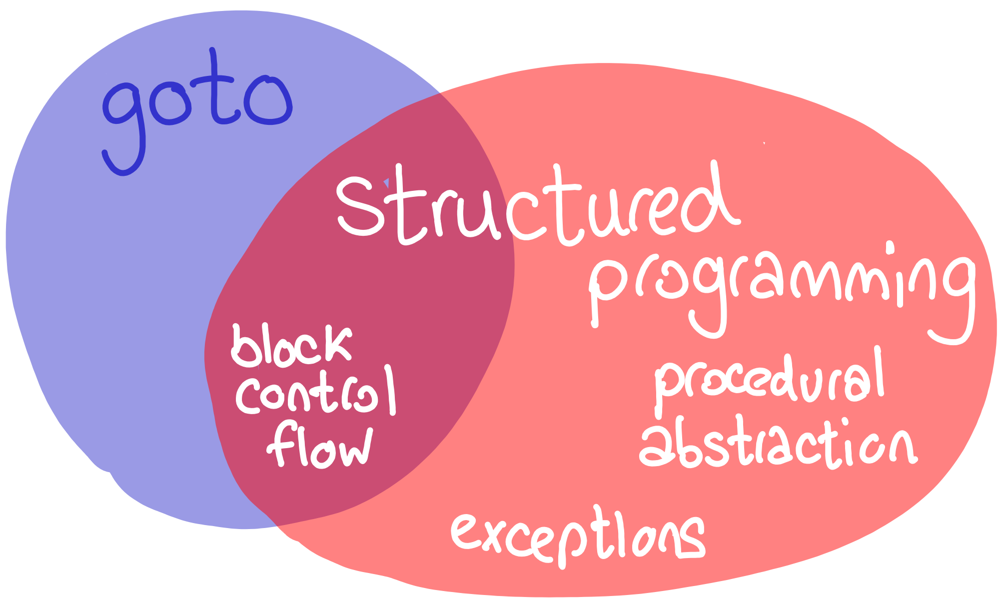


Other examples

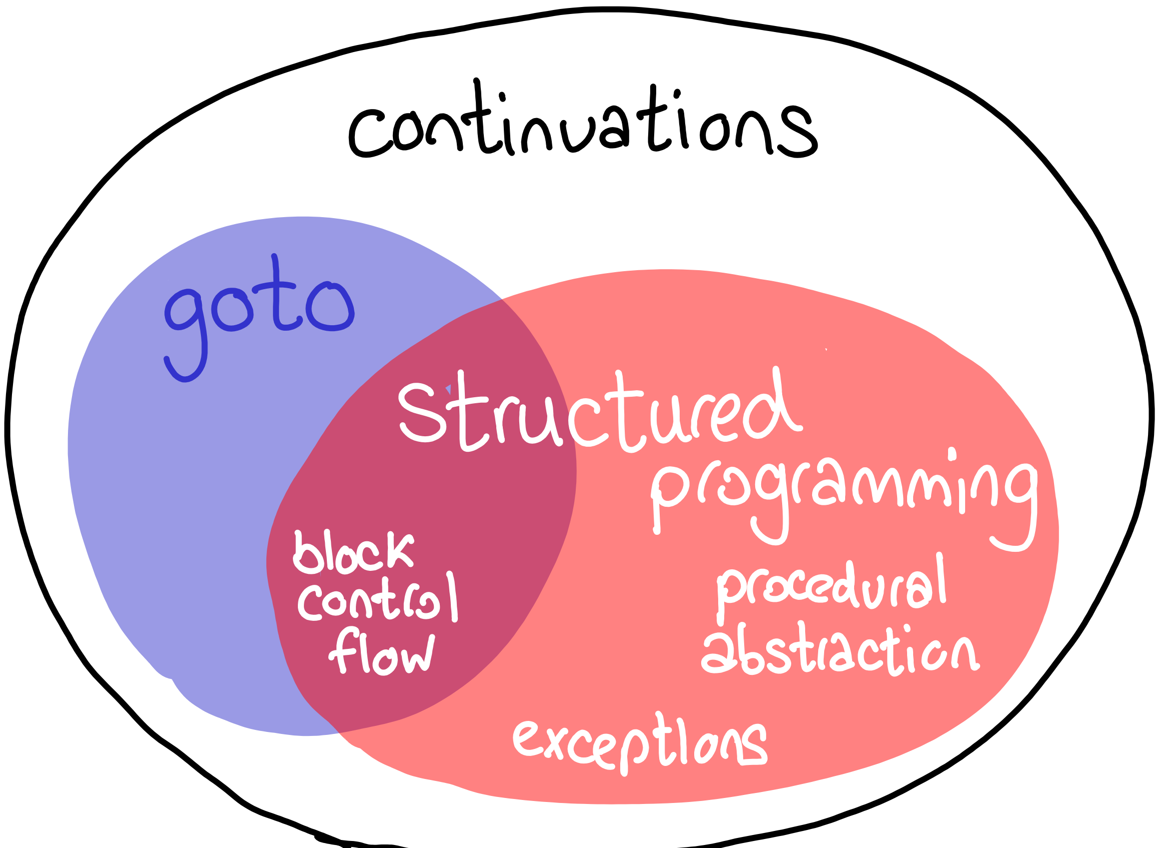
- Call with current continuation
- Nondeterministic choice
- Generators
- Coroutines

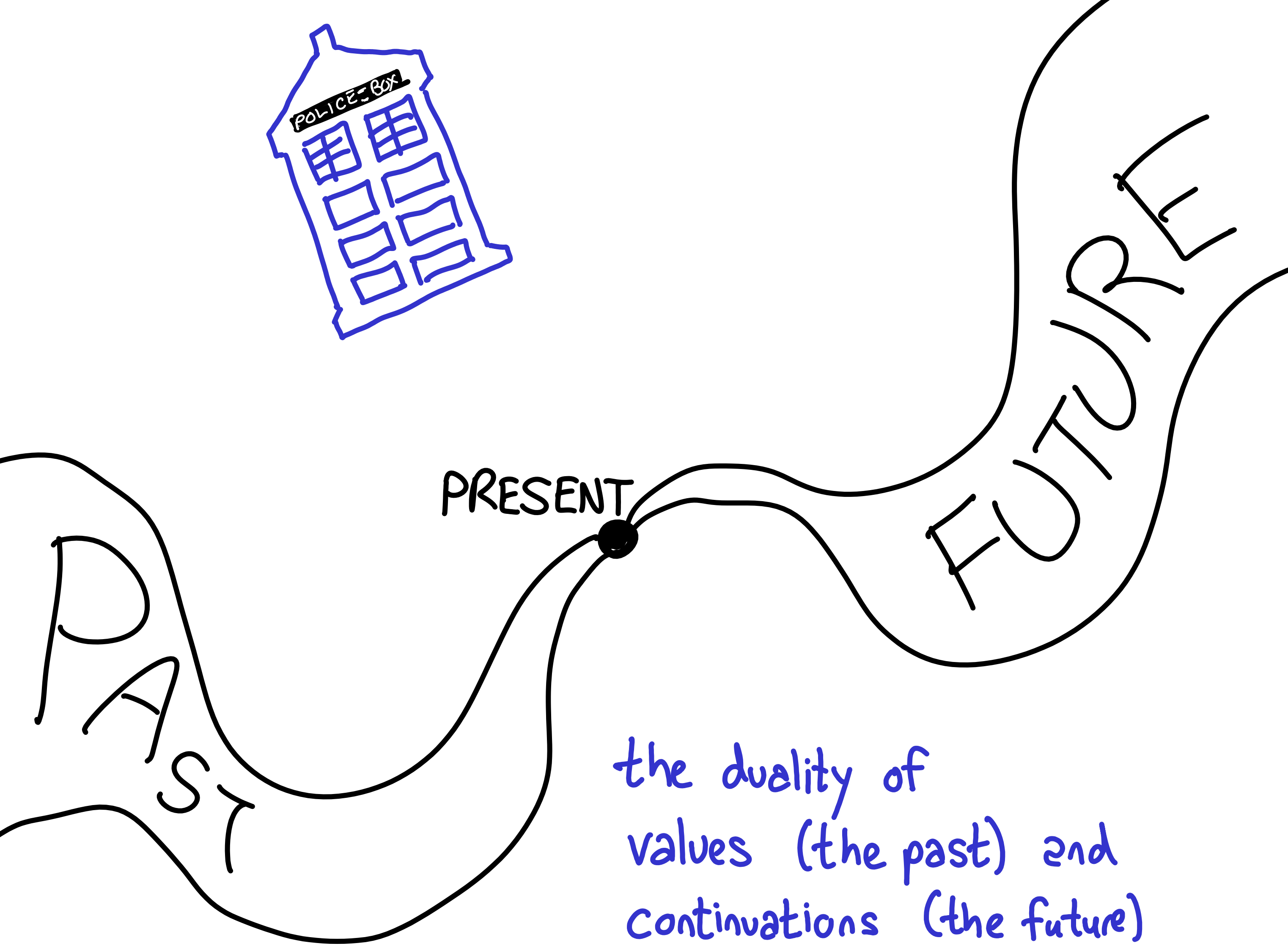
Conclusion

Conclusion


goto
structured programming
block control flow
procedural abstraction
exceptions

Conclusion


continuations
goto
structured programming
block control flow
procedural abstraction
exceptions



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