Control flow, continuations, monads

Week 5a

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
- Monads

Turning the clock back

- To understand what structured programming is, let's look at what programs looked like before it:

```
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
     …
```

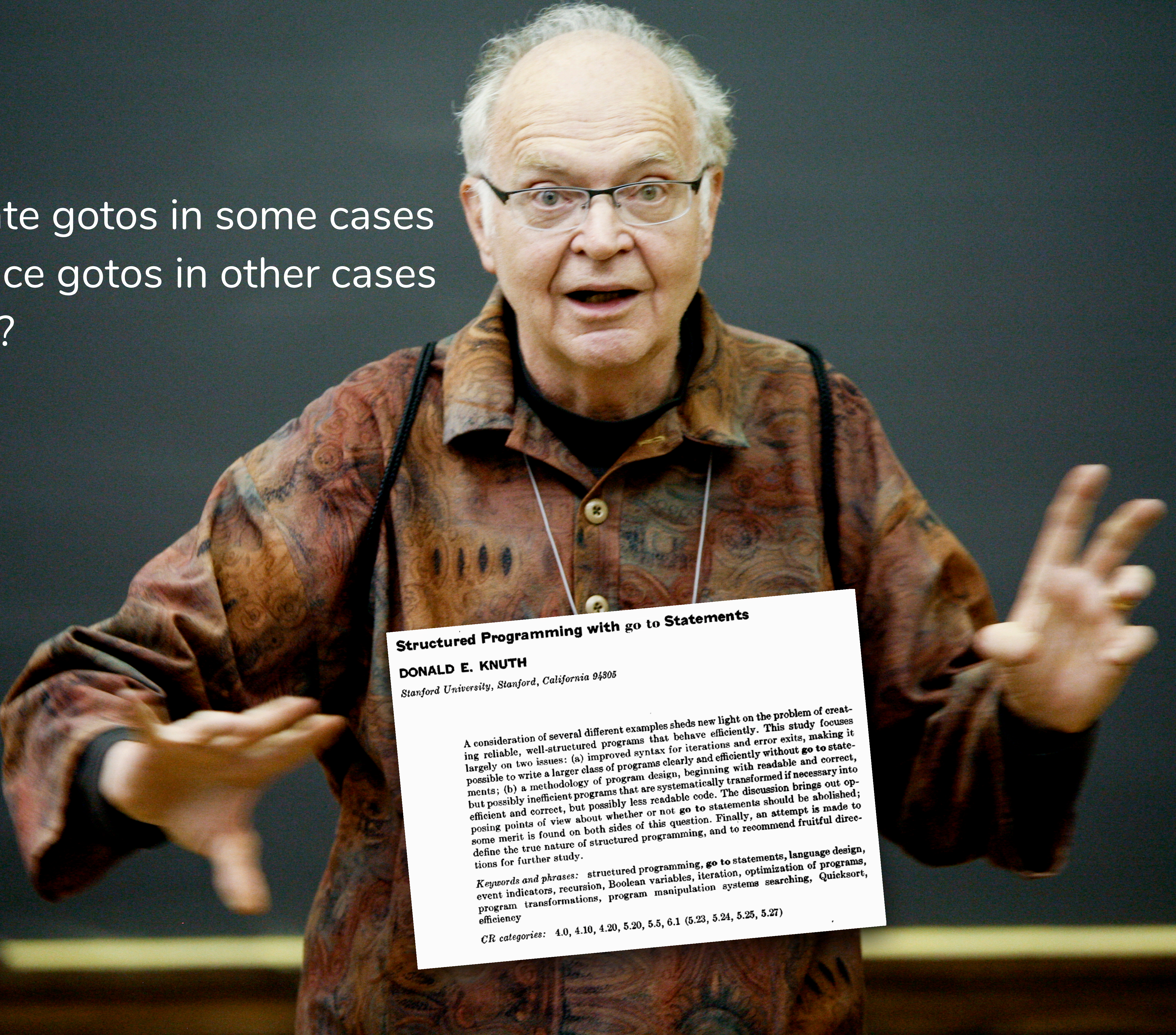
- Do you know what this Fortran program does?
  - A: yes, B: no

Go To Statement Considered Harmful

1. Primary way to understand programs: processing code in sequence
2. goto programs: can jump anywhere => spaghetti
3. Eliminate gotos!
4. Need higher level control flow constructs!

1. Eliminate gotos in some cases
2. Introduce gotos in other cases

Where?



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)

Structured programming

- What is structured programming?
    - Programming paradigm consisting of:
      control structures, procedures, and blocks
- Why?
    -

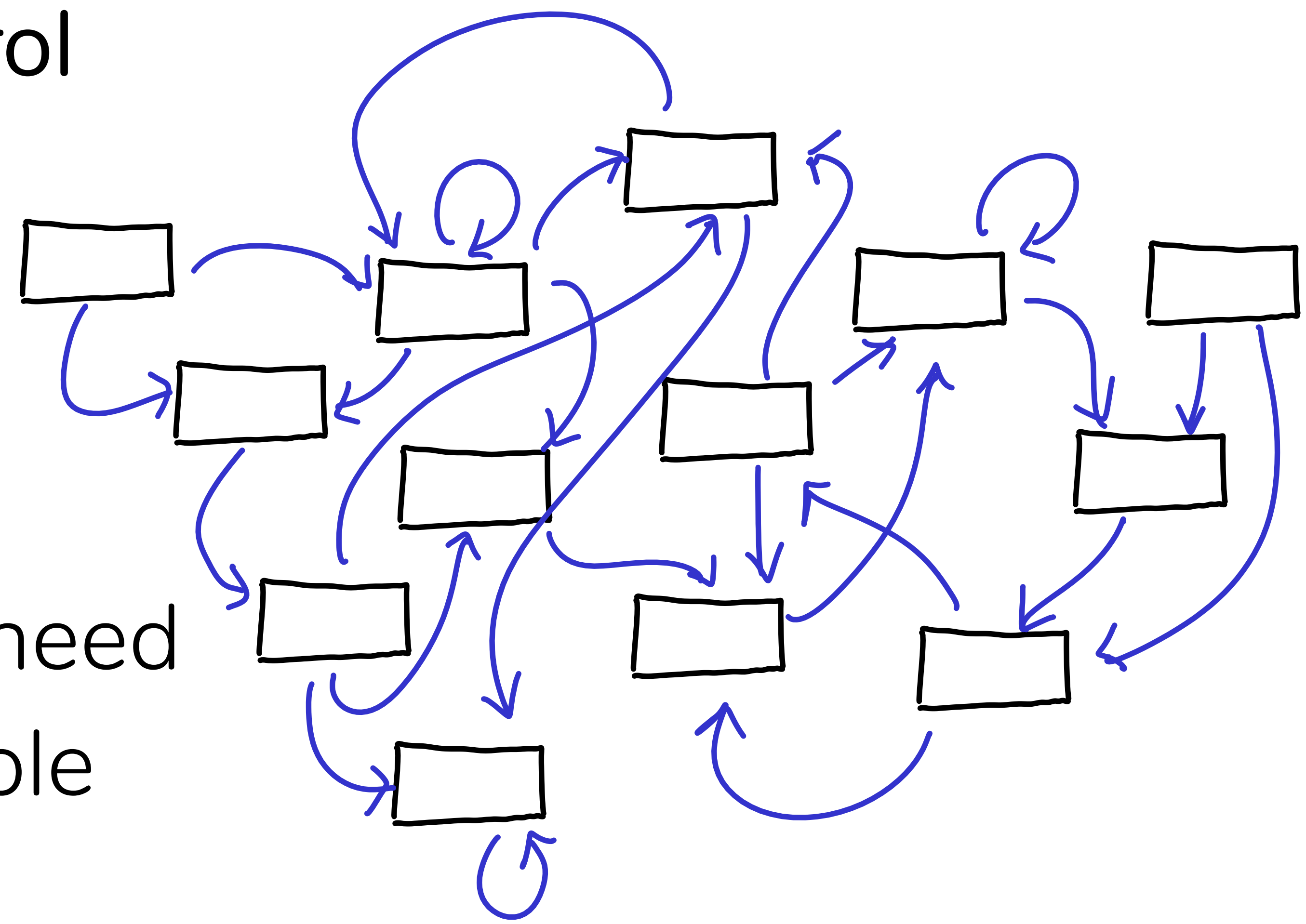
Programming with gotos

- Largely unstructured control flow graphs

- But: extremely flexible!

  Q: What constructs do we need to express any computable function?

  A:



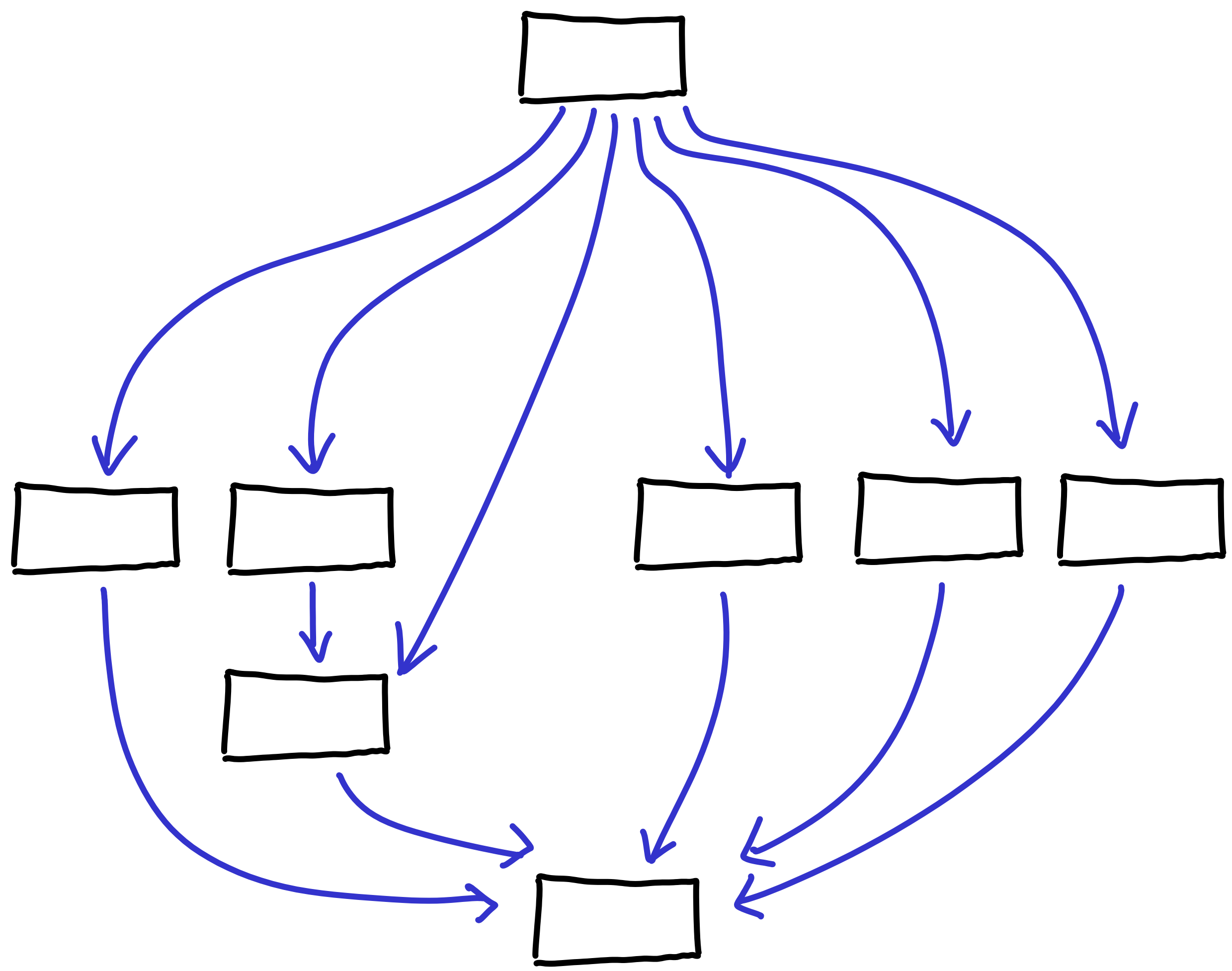
CFG of sequencing

CFG of if-else statement

CFG of while loop

CFG of for loop

What is this a CFG of?



Week 5a

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
- Monads

Procedural abstraction

- We can chain blocks using the previous constructs to program any computable function
  - ➤ Downside:
-

CFG for function calls

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

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

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

CFG for function returns

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

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

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

What about return?

- There may be a lot of places where we could return to from a function call
- In general: determining the interprocedural CFG of a program is hard and super important
    - Modern attacks hijack control flow to execute arbitrary code (control-flow integrity)
- At runtime: how do we know where to go?
    - We keep track of return pointer on the stack!

Dynamic control flow

- The return pointer on the stack dictates where control goes
- Do we need to keep track of where control goes for if-statements, while loops, etc. on the stack as well?
  - A: yes, B: no

Dynamic control flow

- The return pointer on the stack dictates where control goes
- Do we need to keep track of where control goes for if-statements, while loops, etc. on the stack as well?
  - No! We know exactly where execution will go if condition is true or false!

Week 5a

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
- Monads

Exceptions

- Two main language constructs
  - raise/throw
  - handler/catch
- Used to terminate part of a computation
  - Jump out a construct
  - Pass data as part of the jmp
  - Return to the most recent site set up to handle ex

Exceptions

- Can we determine statically where to return to when we throw?
    - A: yes, $\boxed{\text{B: no}}$

Exceptions

- How do we know where to return?
  - keep track of handler information on the stack
  - throw returns to the handler frame that is found on the stack
- How is exception handling scoped?
  - User knows how to handle error
  - Author of library function does not

Simple example

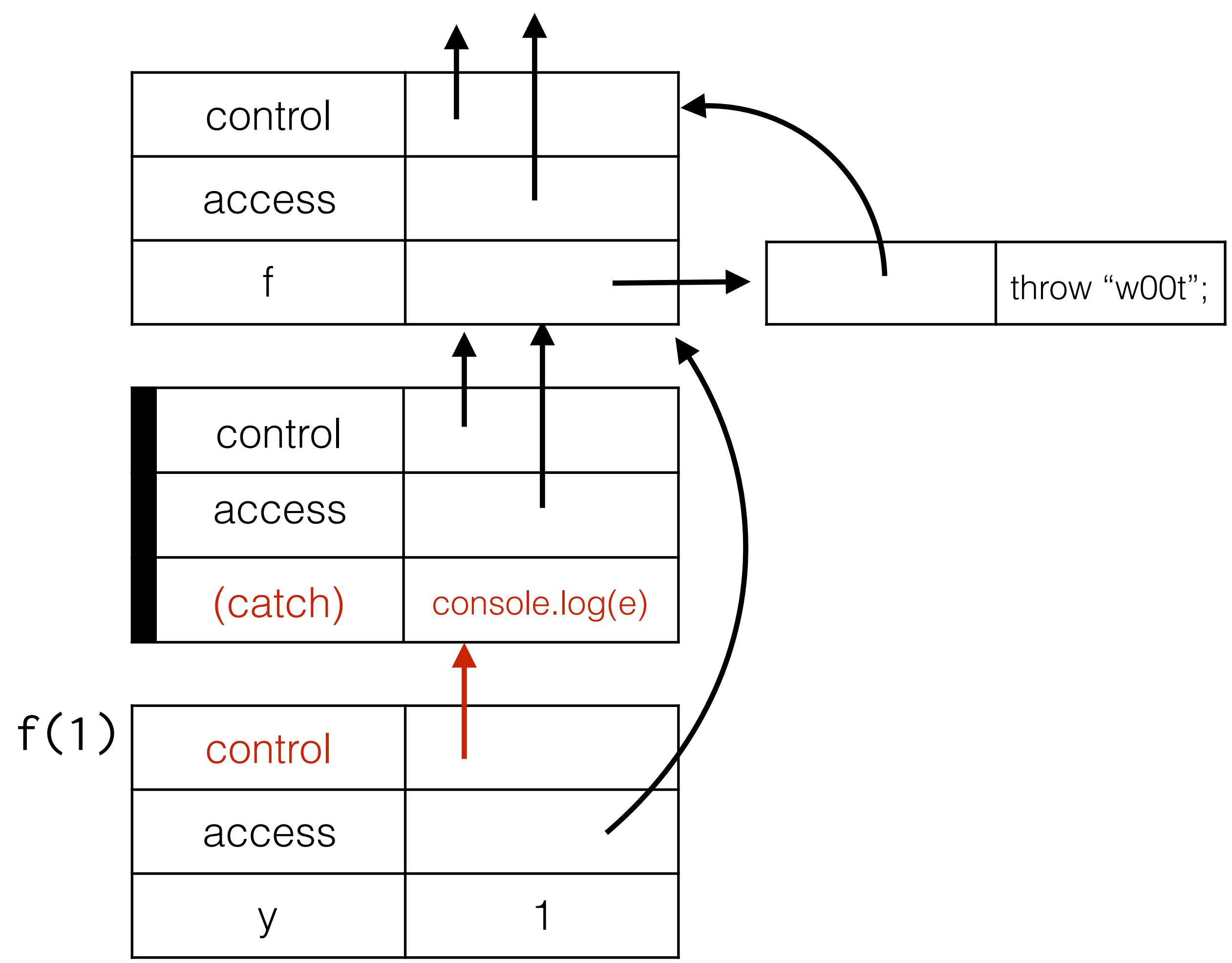
```
function f(y) {
  throw “w00t”;
}

try {
  f(1);
} catch (e) {
  console.log(e);
}
```

Simple example

```
function f(y) {
  throw “w00t”;
}

try {
  f(1);
} catch (e) {
  console.log(e);
}
```



A more complicated example

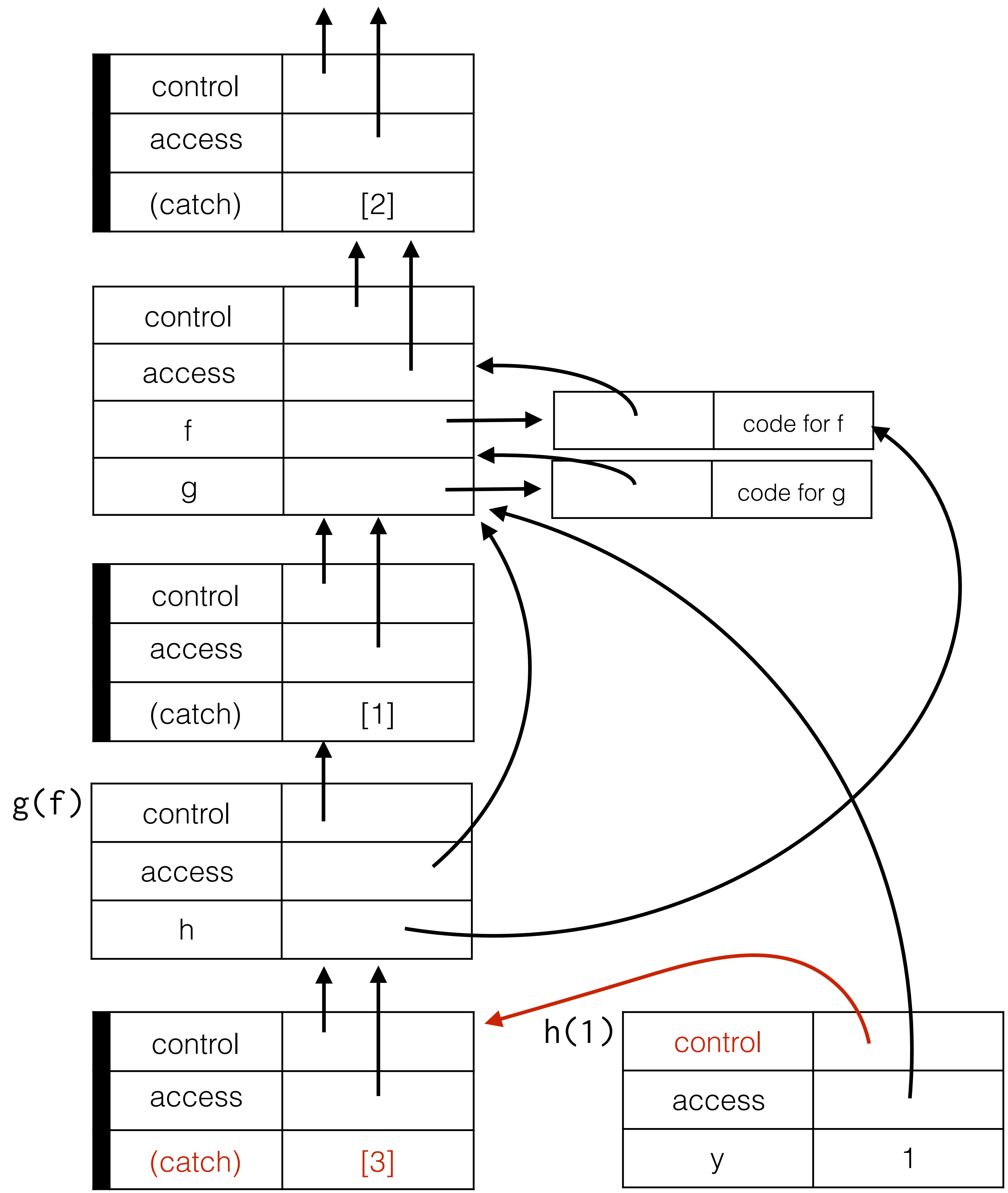
```
try {
 function f(y) {
   throw “w00t”;
 }
 function g(h) {
  try {
   h(1);
  } catch (e) { [3] }
 }

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

A more complicated example

```
try {
 function f(y) {
   throw “w00t”;
 }
 function g(h) {
  try {
   h(1);
  } catch (e) { [3] }
 }

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



Dynamic vs. static scoping

- Again: exceptions follow dynamic scoping rules!
- Which handler would have been called if we had used static/lexical scoping rules?
  - A: [1]
  - B: [2]
  - C: [3]

```
try {
 function f(y) {
   throw "w00t";
 }
 function g(h) {
  try {
   h(1);
  } catch (e) { [3] }
 }

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

Week 5a

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
- Monads

Continuations

- Historical accident: to perform “asynchronous” computations many languages forced users to write their code in continuation passing style
  - Algol 60, Landin’s SECD machine, Scheme
  - See Reynolds’ The Discoveries of Continuations
- History repeats itself: JavaScript!

Example: async programming

```
fs.readFile(‘myfile.txt', (err, data) => {
  console.log(data);
  processData(data);
});
```

- When you write an explicit callback:
  - ➤ you are implementing cooperative multithreading!
  - ➤ callback is a way for thread of execution to “save its current sate” and let other code to run while it waits

Example: debugger

- Debugger is tool that builds on continuations
  - ➤ execution pauses at set breakpoint:
    - ➤ can inspect memory
    - ➤ can continue running the program (have continuation to rest of the program!)

Plan

- What is a continuation?
- Continuation-passing style
- Short summary of how to use continuations to implement control flow

Continuations are implicit in your code

- Code you write implicitly manages the future (continuation) of its computation
- Consider: (2*x + 1/y) * 2

  A. Multiply 2 and x

  B. Divide 1 by y

  C. Add A and B

  D. Multiply C and 2

Continuations are implicit in your code

- Code you write implicitly manages the future (continuation) of its computation

- Consider: (2*x + 1/y) * 2

A. Multiply 2 and x

B. Divide 1 by y

current computation

C. Add A and B

D. Multiply C and 2

rest of the program, current continuation

Continuations are implicit in your code

- Code you write implicitly manages the future (continuation) of its computation

- Consider: (2*x + 1/y) * 2

A. Multiply 2 and x

B. Divide 1 by y

C. Add A and B

D. Multiply C and 2

```
let before = 2*x;
let cont = curResult =>
  (before + curResult) * 2;
cont(1/y)
```

Node.js example

- Implicit continuation:

```
const data = fs.readFileSync(‘myfile.txt’)
console.log(data);
processData(data);
```

- Explicit continuation

```
fs.readFile(‘myfile.txt', callback)
function callback (err, data)  {
  console.log(data);
  processData(data);
};
```

Continuation passing style

- Why do we want to do this?
  - Makes control flow explicit: no return!
  -
- So? Why should you care about this?
  -
  -

To CPS, by example

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

To CPS, by example

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

To CPS, by example

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

```
fact(3, id) ->
fact(2, r_A => id(3*r_A)) ->
fact(1, r_b => (r_A => id(3*r_A))(2*r_b)) ->
fact(0, r_c => (r_b => (r_A => id(3*r_A))(2*r_b))(1*r_c)) ->
(r_c => (r_b => (r_A => id(3*r_A))(2*r_b))(1*r_c))(0) ->
(r_b => (r_A => id(3*r_A))(2*r_b))(1*0) ->
(r_A => id(3*r_A))(2*1*0) ->
id(3*2*1*0)
```

To CPS, by example

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

```
function cmp(f, g, x) {
  return f(g(x));
}
```

To CPS, by example

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

To CPS, the rules

- Function decls take extra argument: the continuation
  - `function (x) {` ➡ `function (x, cc) {`
- There are no more returns! Call continuation instead
  - `return x;` ➡ `cc(x);`
- Lift nested function calls out of subexpressions
  - ```
    let r = g(x);
    stmt1
    stmt2
    ```
    ➡
    ```
    g(x, r => {
       stmt1 ; stmt2
     })
    ```
    ```

Why is this useful?

- Makes control flow explicit
    - Compilers like this form since they can optimize code
- Multithreaded programming
- Event based programming such as GUIs

Continuations are extremely powerful

- Generalization of goto!
- Can implement control flow constructs using continuations
- How do we do if statements?
- How do we do exceptions?

Exceptions w/ continuations

```
function f() { throw “w00t”; }

try {
  f();
  console.log(“no way!”);
} catch (e) {
  console.log(e);
}
console.log(“cse130 is lit”);
```

Exceptions w/ continuations

current cont = line 9

```
1. function f() { throw "w00t"; }
2.
3. try {
4.   f();
5.   console.log("no way!");
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log("cse130 is lit");
```

Exceptions w/ continuations

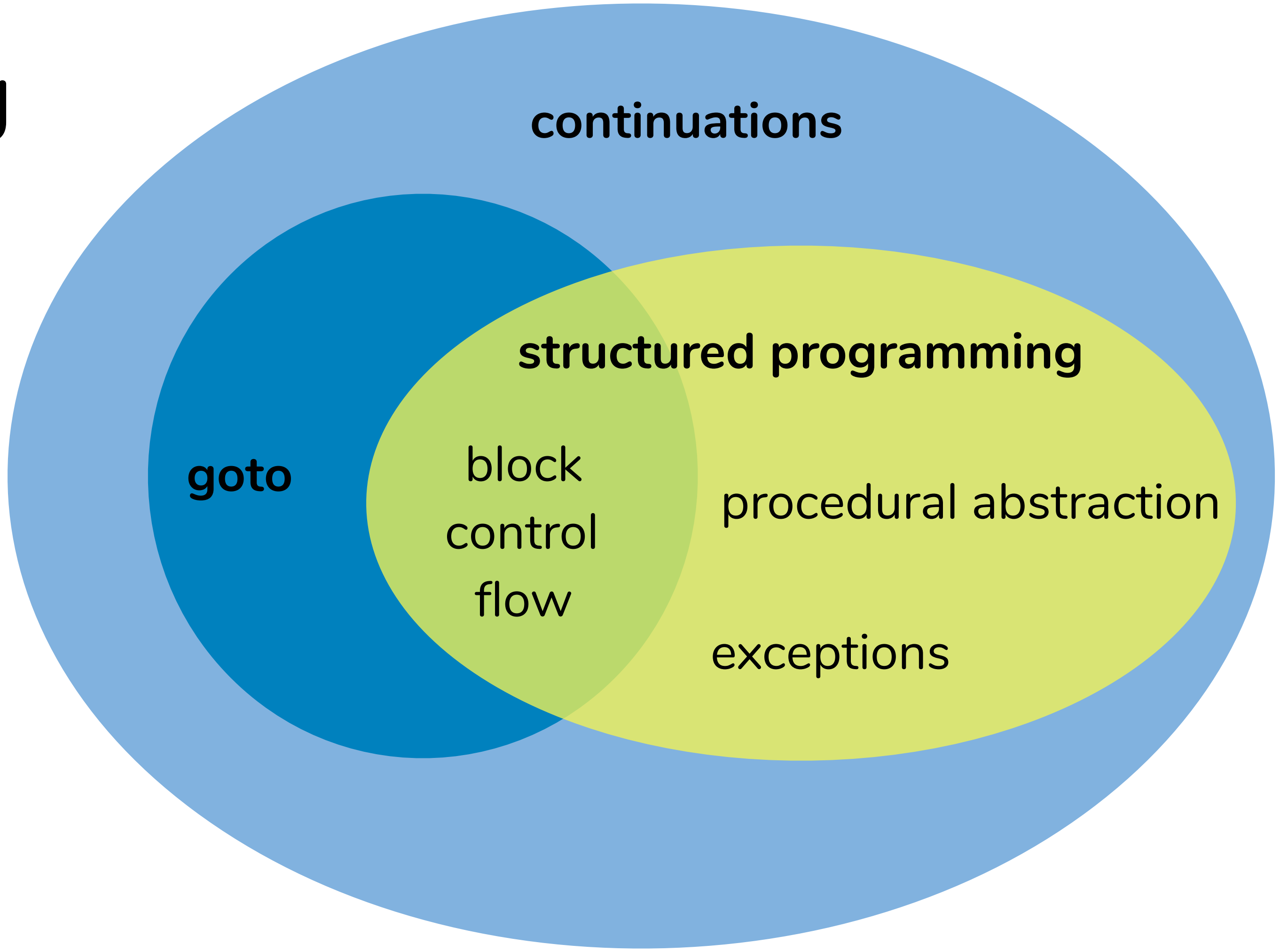
success cont =
line 5;previous cc = lines 5;9

fail cont =
lines 6-8;previous cc = lines 6-9

```
1. function f() { throw “w00t”; }
2.
3. try {
4.   f();
5.   console.log(“no way!”);
6. } catch (e) {
7.   console.log(e);
8. }
9. console.log(“cse130 is lit”);
```

Putting continuations in context

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations



Week 5a

- Structured programming
- Procedural abstraction
- Exceptions
- Continuations
- Monads

Can we do IO as usual?

```
ls :: [(), ()]
ls = [putChar ‘x’, putChar ‘y’]
```

Is this okay? A: yes, B: no

Laziness gets in the way?

- Depending on evaluation order order of effects may vary or may not even be observed
  - E.g., `length ls` vs. `head ls`
- Without laziness, are we okay? A: yes, B: no
  -

Monad IO

- Extend category of values with actions
- A value of type (IO a) is an action
- When performed, the action of type IO a may perform some I/O before it delivers a result of type a
- How to think about actions:
  -

```
getChar :: IO Char
```

IO actions are first-class

- What does this mean? (Recall: first-class functions)
    -
    -
    -

```
putChar :: Char -> IO ()
```

How do we create actions?

- The return function:
  - Worst name ever: has nothing to do with terminating early
  - Given value produce IO action that doesn't perform any IO and only delivers the value
  - Type:

Example: return

- `return 42`
- ```
  f x =  if x
           then return “what”
           else return “no way!”
  ```
  ```

How do we create actions?

- The compose function (>>)
  - Given an IO action act$_1$ and action act$_2$ produce a bigger action, which when executed:
    - executes act$_1$
    - execute act$_2$ and deliver the value produced by act$_2$
  - Type: (

Example: >>

- `return 42 >> putChar ‘A’ >> putChar ‘B’`

- ```
  f x =  putStrLn “hello world” >>
          if x == “hello”
            then return x
            else return “bye bye!”
  ```
  ```

How do we create actions?

- The bind function (>>=)
  - Like (>>), but doesn't drop the result of first action: it chains the result to the next action (which may use it)
  - Type:
- Can we define (>>) in terms of (>>=)? A: yes, B: no

(>>) via (>>=)

- Recall:
  - `(>>=) :: IO a -> (a -> IO b) -> IO b`
  - `(>>)  :: IO a ->      IO b    -> IO b`
- From this:
  - `(>>) act1 act2 =`

Example: >>=

- `return 42 >>= (\i -> putChar (chr i))`

- ```
  echo :: IO ()
  echo =
  ```
  ```

Example: >>=

- ```
  echoTwice :: IO ()
  echoTwice =
  ```

- ```
  getTwoChars :: IO (Char, Char)
  getTwoChars =
  ```

Do notation

- Syntactic sugar to make it easier create big actions from small actions

- ```
  getTwoChars :: IO (Char, Char)
  getTwoChars = do
      c1 <- getChar
      c2 <- getChar
      return (c1, c2)
  ```
  ```

Do notation: de-sugaring

- do x <- e ➡
  s
- do e ➡
  s
- do e ➡

How do we execute actions?

- Haskell program has to define main function
  - `main :: IO ()`
- To execute an action it has to be bound!

wc -l in Haskell

Mutable references in IO monad!

- data IORef a = ….
  - readIORef ::
  - writeIORef ::
  - atomicModifyIORef ::

Can we escape IO monad?

- Is it okay to define a function of type: IO a -> a

No! unsafePerformIO can be used to violate type safety

Monads are cool!

- Principled way to expose imperative programming in FP languages
- Evaluation order is explicit
- Idea goes beyond IO: you can define your own monad
  - Monad is a type class (with return and >>=)
  - E.g., LIO monad does security checks before performing, say, a readFile to prevent data leaks

Monads are a type class?

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

hasmap.hs

Functor type class

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- Laws
  -
  -
- What does this mean?

Monad type class

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

- Laws
    - ➤
    - ➤
    - ➤
- What does this mean?

Why do these matter?

- <u>Theorem:</u> `putStr r >> putStr s = putStr (r ++ s)`
- Proof (base case):

Why do these matter?

- <u>Theorem:</u> `putStr r >> putStr s = putStr (r ++ s)`
- Proof (inductive case):

Example instance that's not IO?

```
instance Monad Maybe where
  return :: a -> Maybe a



  (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```