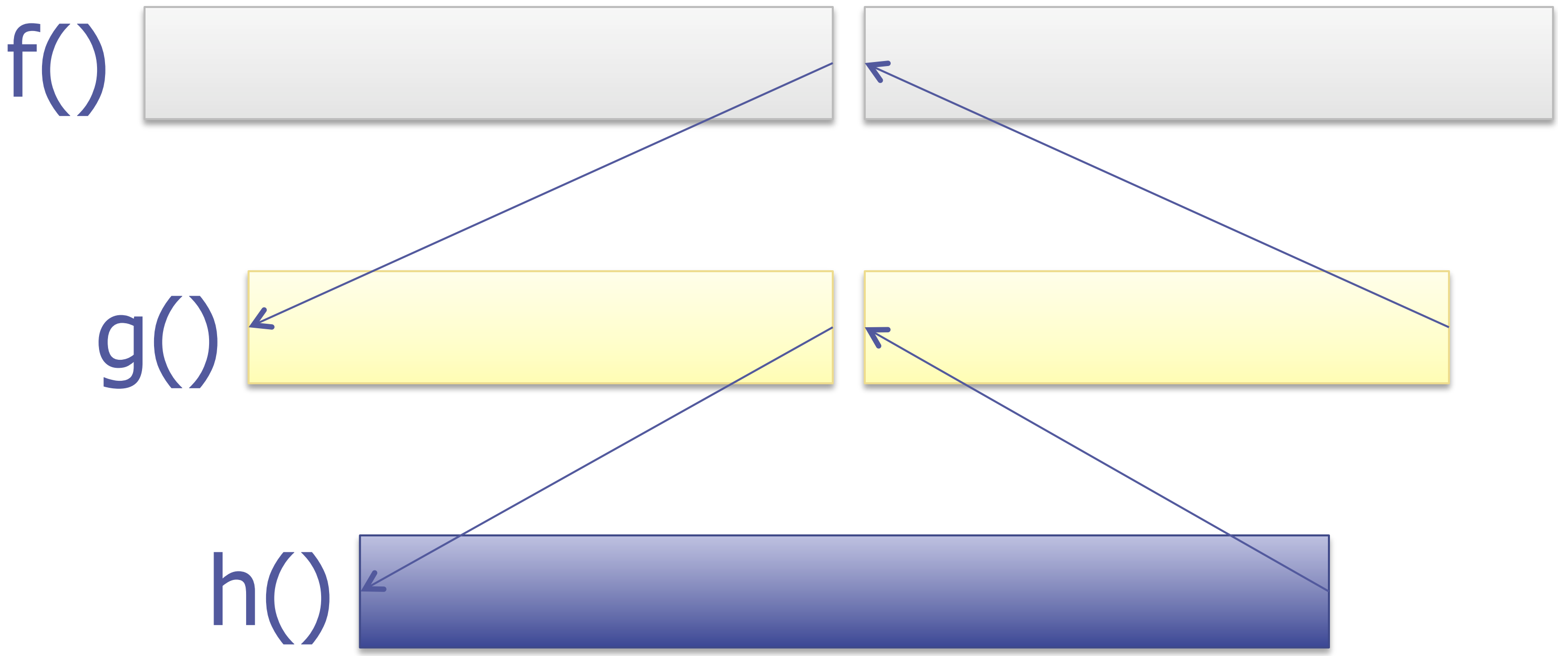
CS 410/510: Advanced Programming

Continuations …

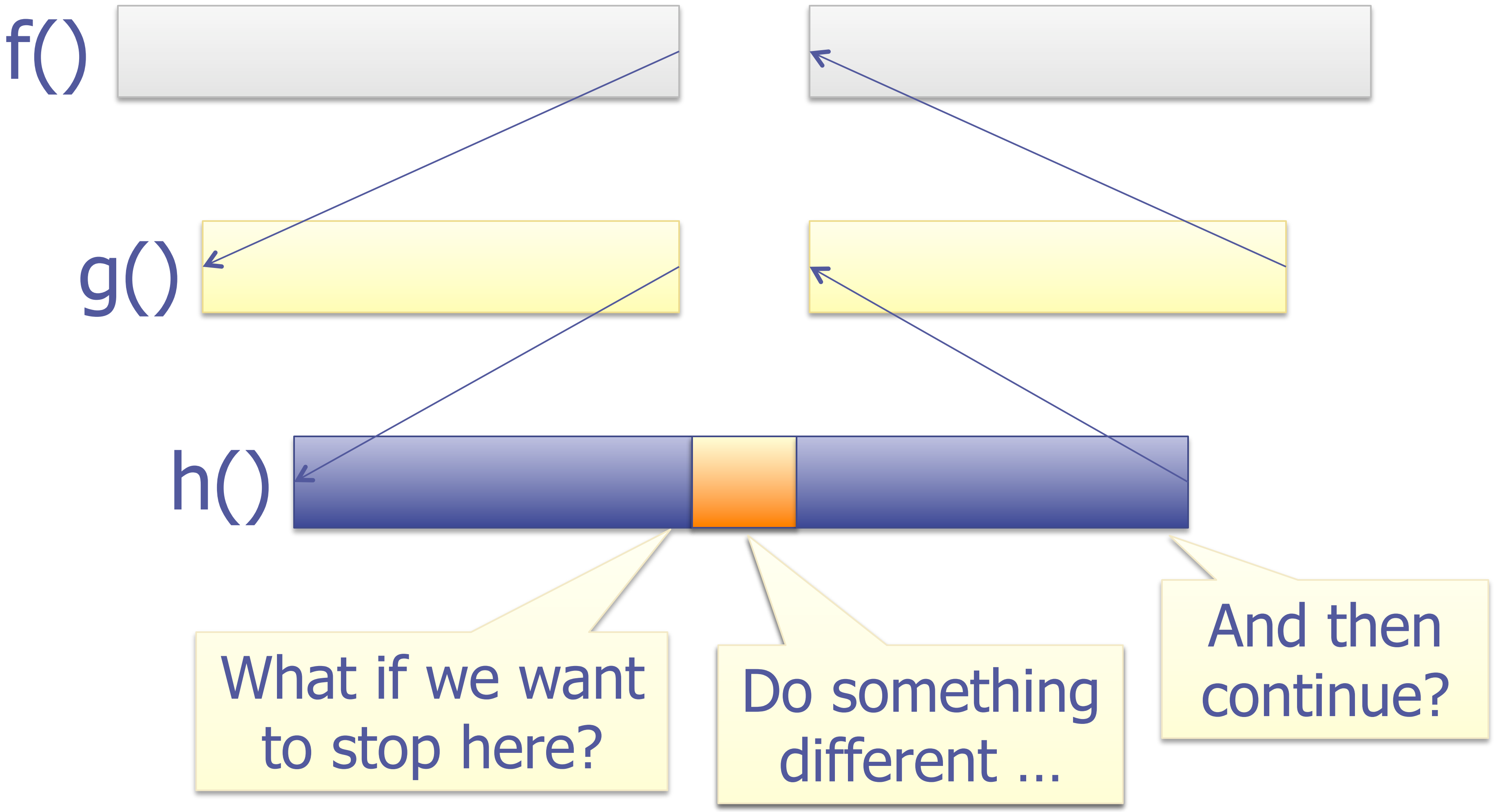
Mark P Jones
Portland State University

Continuing a Computation:



Standard nested function call pattern

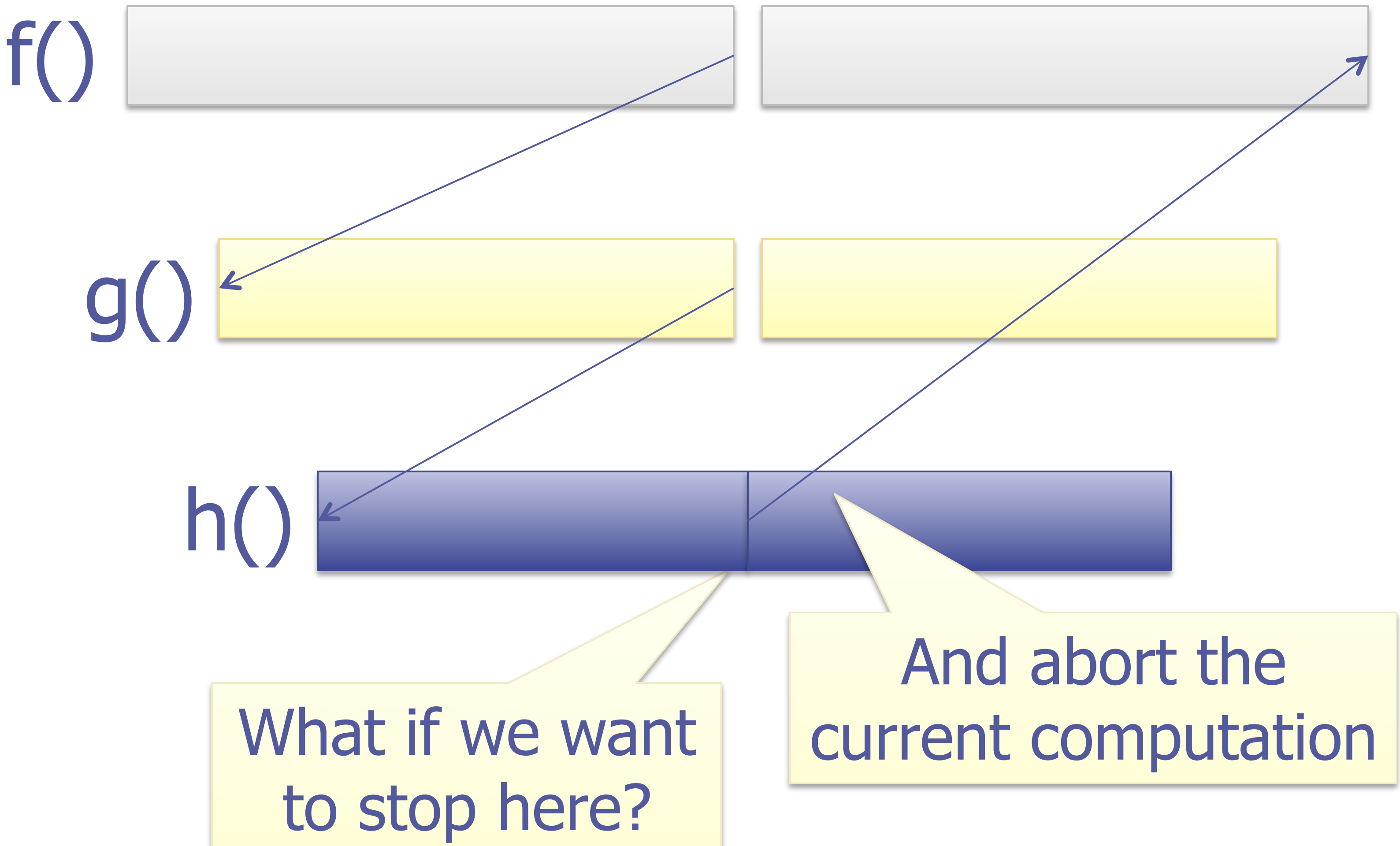
Continuing a Computation:



What might we want to do?

- Run a higher-priority task?
- Yield the processor to another task because our timeslice is up?
- Use the processor for some other activity while we wait for input?
- Pause current activity to allow for updates/ maintenance?
- Insert some debugging code/hand off control to a debugger?

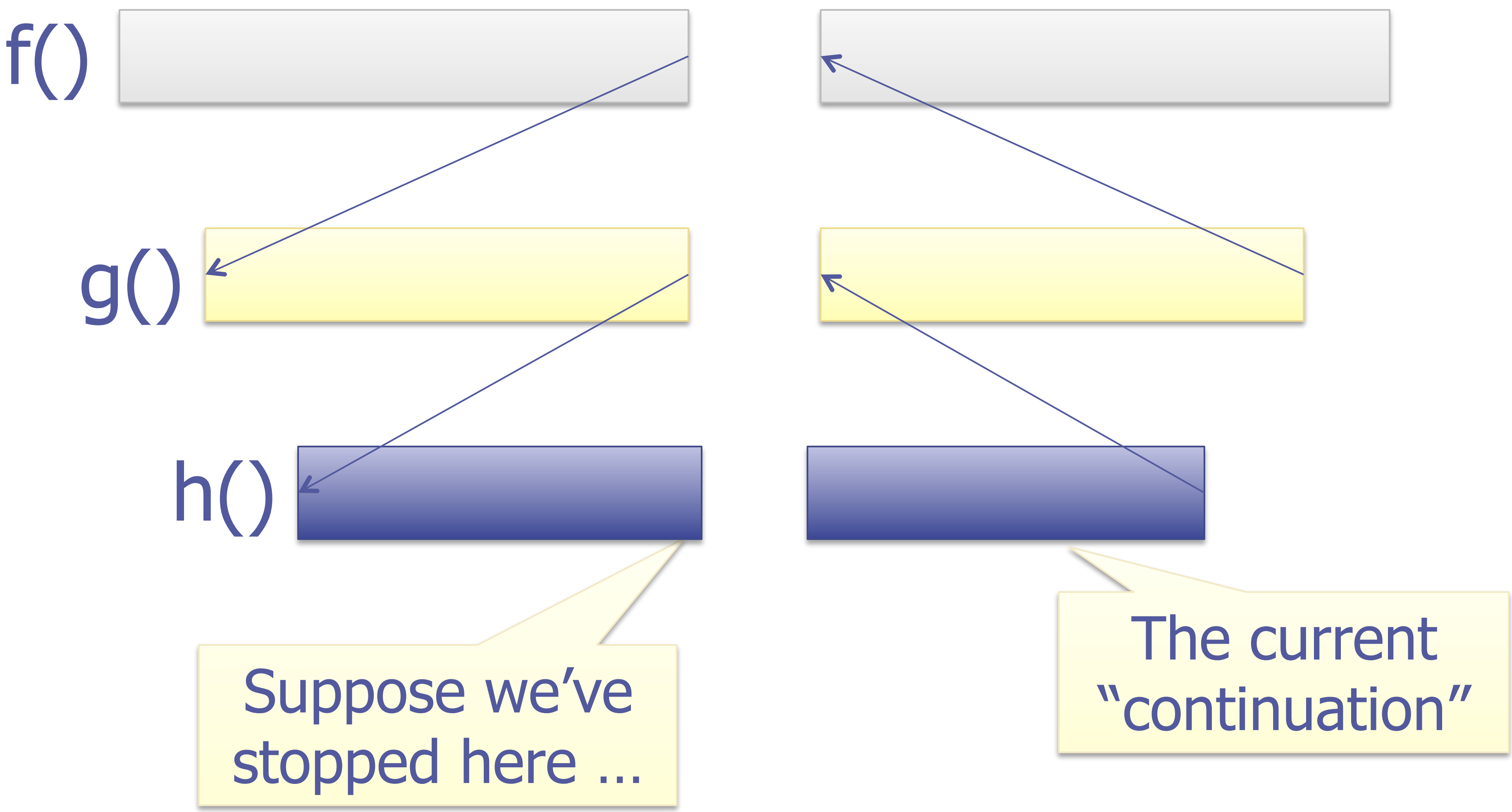
Continuing a Computation:



Why might we want to abort?

- We were trying to solve a problem and have found the solution?
- We were trying to solve a problem and have found that there is no solution?
- The current activity is no longer required (e.g., it has been interrupted)?
- An exception has occurred?
- We don’t want to percolate a series of return codes back up the call graph?

Continuing a Computation:



How can we use a continuation?

- Call it!
- Replace it!
- Store it!
- Modify it!
- Inspect it!
- The set of choices that we have depends on the language and the implementation

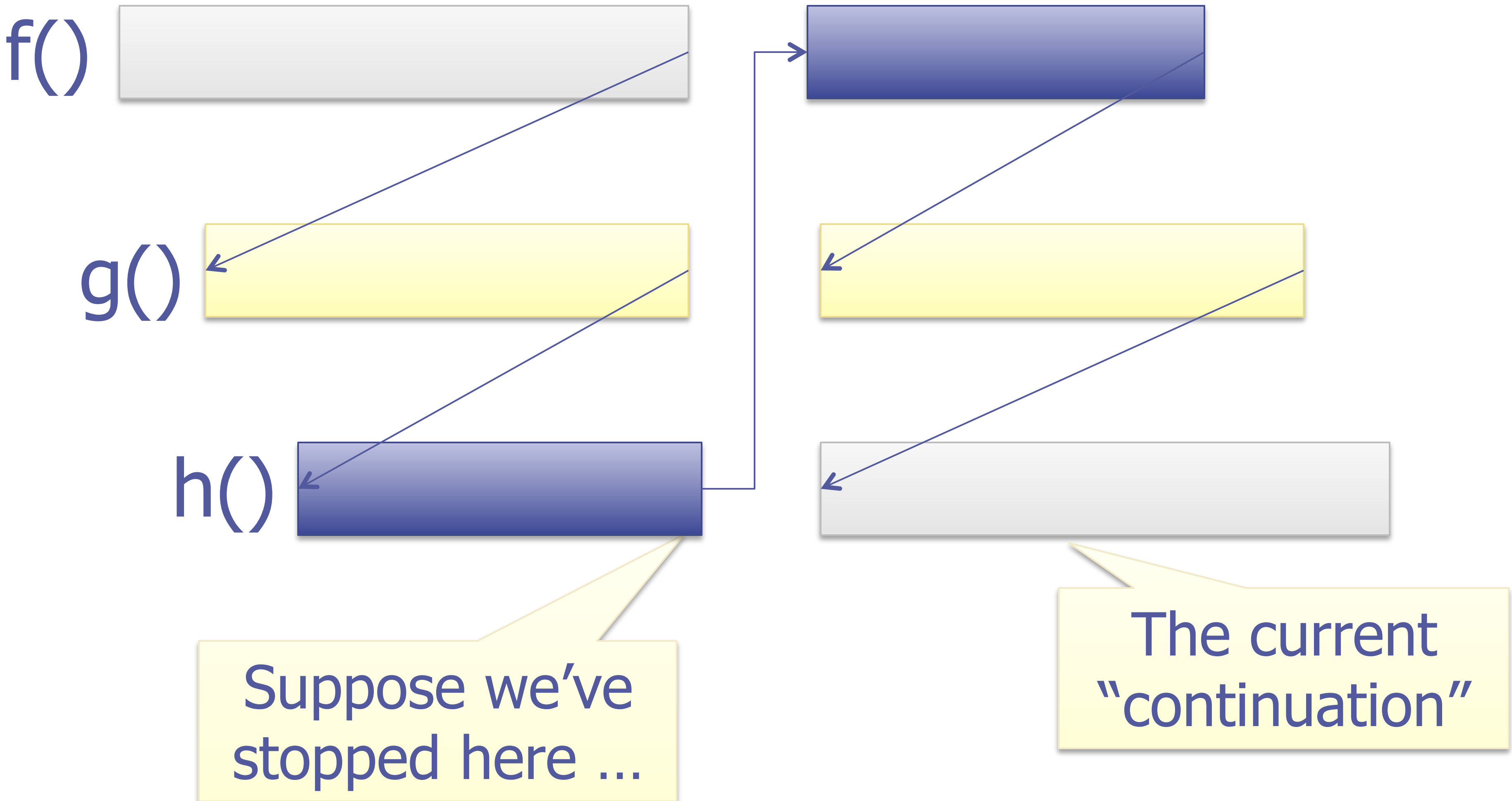
Capturing Continuations:

- In a conventional programming language implementation, the current continuation corresponds to a sequence of "stack frames" + some other state

- In theory, we can copy/store/reify the current continuation as a data structure

- In practice, we can have a separate stack for each process and switch stack pointers (+ local state) to move between them

Capturing Continuations:



Call/cc:

- Conventional languages typically do not provide a way for a program to access its own continuation

- Scheme and SML/NJ are among the exceptions, providing a very powerful mechanism called call-with-current-continuation:

```
(define (f return)  (return 2)  3)
(display (f (lambda (x) x)))
(display (call-with-current-continuation f))
```

Example thanks to Wikipedia

- In other languages, we simulate continuations via functions, blocks, closures, objects, etc…

Continuations as Functions:

- Alternatively, continuations can be described by functions of type (a -> Ans), for some fixed answer type Ans

- Every program takes a continuation as an extra argument

- Functions “return” by passing a result to their continuation (or, if appropriate, to some other continuation)

Continuation Passing Style:

```
fact  :: Integer -> Integer
fact n = if n==0
           then 1
           else n * fact (n-1)

kfact    :: Integer -> (Integer -> result) -> result
kfact n k = if n==0
              then k 1
              else kfact (n-1) (\x -> k (n * x))
```

Continuation Passing Style:

```
kfact    :: Integer -> (Integer -> result) -> result
kfact n k = eq n 0 (\b ->
              if b
                then k 1
                else minus n 1 (\m ->
                       kfact m (\f ->
                       mult n f k)))
```

In this version, even primitive operations have a continuation argument; note that the order of evaluation is explicit in this code

Why is CPS interesting?

- A good intermediate language for compilers
  - Makes order of evaluation and control flow explicit
  - Conversion to CPS can be automated
  - Capturing continuations is cheap; a pointer to a heap-allocated closure, no stack copying
- Correspondence with classical logic
- Applications in linguistics
- A useful tool for program structuring …

Sequencing Side Effects:

```
getchar      :: (Char -> Ans) -> Ans

putchar      :: Char -> (() -> Ans) -> Ans

echo = getchar   (\c  ->
       putchar c (\() ->
       echo))
```

Sequencing Side Effects (implementation):

```
type Ans = IO ()

getchar       :: (Char -> Ans) -> Ans
getchar k     = do c <- getChar; k c

putchar       :: Char -> (() -> Ans) -> Ans
putchar c k   = do v <- putChar c; k v

echo = getchar    (\c  ->
       putchar c  (\() ->
       echo))
```

Returning Multiple Results:

- Many languages allow functions to take multiple arguments, but restrict returns to a single result

- What if you want to return two results?
  - ~~**Option 0**: save one result in a global variable (shudder)~~
  - **Option 1**: return an object that holds multiple values

    ```
    roots :: Float -> Float -> Float -> (Float, Float)
    ```

  - **Option 2**: provide a multiple argument continuation

    ```
    roots :: Float -> Float -> Float
                      -> (Float -> Float -> a) -> a
    ```

Multiple Continuations:

- For a variety of reasons, an attempt to open a file may either succeed or fail
- How should an API reflect these possibilities?
  - ~~**Option 0**: indicate success via a return code (shudder)~~
  - **Option 1**: return an object that represents alternatives

    ```
    openFile :: String -> IO (Either Handle Error)
    ```

  - **Option 2**: provide multiple continuations

    ```
    openFile :: String -> (Handle -> IO a)
                      -> (Error -> IO a) -> IO a
    ```

Other Application:

- Non-local exits
- Exceptions
- Coroutines
- Concurrency

A Concurrency Monad:

```
instance Monad C
execute      :: C a -> IO ()
done         :: C ()
display      :: Show a => a -> C ()

fork         :: C a -> C b -> C (a, b)
(<||>)       :: C a -> C b -> C ()

newChan      :: C (Chan a)
input        :: Chan a -> C a
output       :: Chan a -> a -> C ()
```

Cooperative Concurrency:

- Our implementation is pure Haskell
- Simulated concurrency, no preemption

```
Main> execute (display "hello" <||> display "world")
"world"
"hello"
done
Main>
```

- A truly concurrent, preemptive implementation is possible but requires new run-time system primitives

Process Queues:

```
data Procs = Procs { procs :: [Proc] }
type Proc  = Procs -> IO ()

resched                         :: () -> Proc
resched () (Procs [])           = error "deadlock!"
resched () (Procs (q:qs))       = q (Procs  qs)

sched                           :: Proc -> Proc -> Proc
sched p q (Procs ps)            = q (Procs (ps++[p]))
```

A Continuation Monad:

```
type Cont a     = a -> Proc
data C a        = C { runC :: Cont a -> Proc }

instance Monad C where
  return x = C (\k -> k x)
  c >>= f  = C (\k -> runC c (\a -> runC (f a) k))

execute      :: C a -> IO ()
execute c    = runC c (\a w -> putStrLn "done")
                    (Procs [])
```

… continued :-)

```
done          :: C ()
done          = return ()

display       :: Show a => a -> C ()
display x     = C (\k w -> do print x; k () w)
```

References:

```
newvar  :: a -> Cont (Ref a) -> Proc

assign  :: Ref a -> a -> Cont () -> Proc

deref   :: Ref a -> Cont a -> Proc
```

References (implementation):

```
type Ref    = IORef

newvar      :: a -> Cont (Ref a) -> Proc
newvar x    = \k w -> do r <- newIORef x; k r w

assign      :: Ref a -> a -> Cont () -> Proc
assign r x  = \k w -> do writeIORef r x; k () w

deref       :: Ref a -> Cont a -> Proc
deref r     = \k w -> do a <- readIORef r; k a w
```

Fork Implementation:

```
data Fork a b = Running | LDone a | RDone b

fork        :: C a -> C b -> C (a, b)
fork p q    = C (\k -> newvar Running (\v ->
                  sched (runC p (lDone k v))
                        (runC q (rDone k v))))

lDone        :: ((a,b) -> Proc) -> Ref (Fork a b) -> a -> Proc
lDone k v a = deref v (\f ->
               case f of
                 Running  -> assign v (LDone a) resched
                 RDone b  -> k (a, b))
-- rDone similar
```

Parallel Execution:

```
(<||>)          :: C a -> C b -> C ()
p <||> q        = fork p q >> done

parList         :: [C a] -> C [a]
parList []      = return []
parList (p:ps)  = do  (x, xs) <- fork p (parList ps)
                      return (x:xs)

parCmds         :: [C a] -> C ()
parCmds         = foldr (<||>) done
```

Channels:

```
type Chan a          = Ref (ChanStatus a)
data ChanStatus a    = Inactive
                      | InReady (a -> Proc)
                      | OutReady a (() -> Proc)

newChan              :: C (Chan a)
newChan              = C (newvar Inactive)

newChans             :: [a] -> C [Chan b]
newChans cs          = parList [ newChan | c <- cs ]
```

Input from a Channel:

```
input :: Chan a -> C a
input c
 = C (\k -> deref c (\cs ->
        case cs of
          Inactive -> assign c (InReady k) resched
          OutReady v k' -> sched (k' ())
                           (sched (k v)
                           (assign c Inactive resched))
          InReady k'   -> error "simult. inputs"))
```

Output to a Channel:

```
output    :: Chan a -> a -> C ()
output c v
 = C (\k -> deref c (\cs ->
        case cs of
          Inactive -> assign c (OutReady v k) resched
          InReady k' -> sched (k' v)
                           (sched (k ())
                           (assign c Inactive resched))
          OutReady v' k' -> error "simult. outputs"))
```

Example:

```
chanEx = do c <- newChan
            output c "hello, world" <||>
             (do msg <- input c; display msg)
```

Can also be written:

```
chanEx = do c <- newChan
            output c "hello, world" <||>
             (input c >>= display)
```

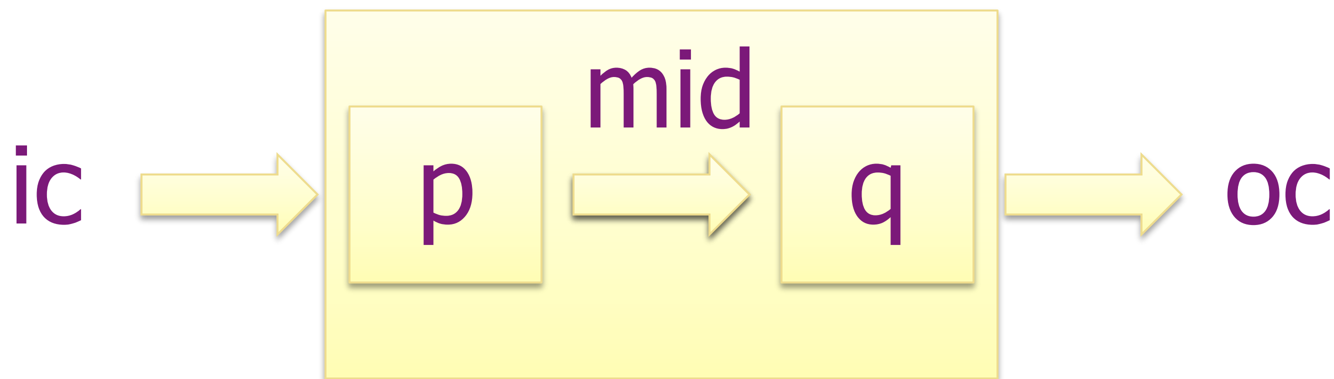
Now try: `execute chanEx`

Pipes:

```
type Pipe a b = Chan a -> Chan b -> C ()

mapChan          :: (a -> b) -> Pipe a b
mapChan f i o    = loop (do x <- input i; output o (f x))

filterChan       :: (a -> Bool) -> Pipe a a
filterChan p i o = loop (do x <- input ic
                            when (p x) (output oc x))

loop             :: C a -> C ()
loop p           = do p; loop p
```

Plumbing:

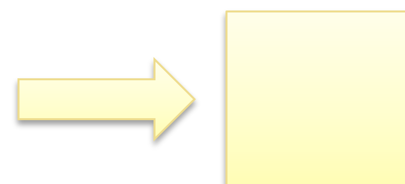
```
(>>>)        :: Pipe a b -> Pipe b c -> Pipe a c
p >>> q      = \ic oc  -> do mid <- newChan
                             p ic mid <||> q mid oc
```



Pumps and Drains:

```
pump          :: [a] -> Chan a -> C ()
pump xs c     = mapM_ (output c) xs
```

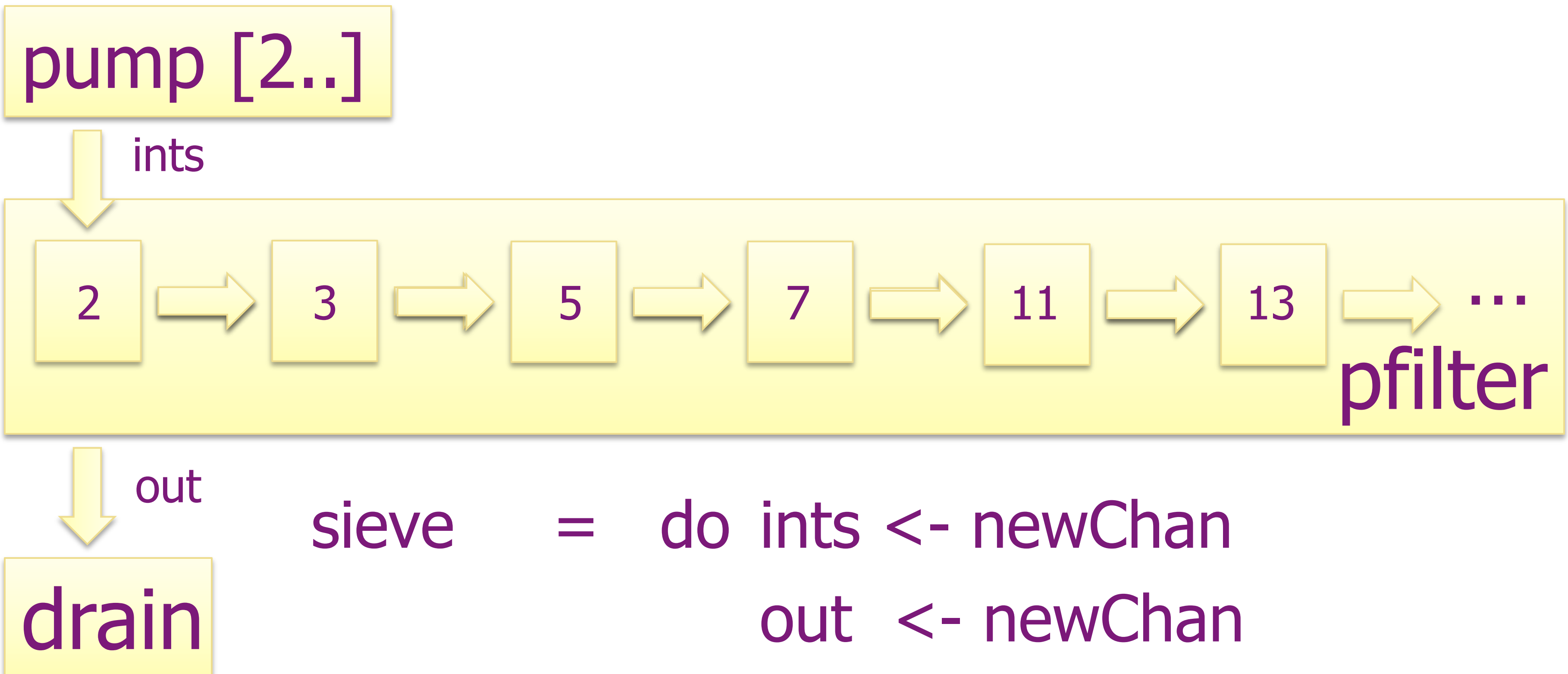
```
drain         :: Show a => Chan a -> C ()
drain c       = loop (input c >>= display)
```



The Sieve of Eratosthenes:

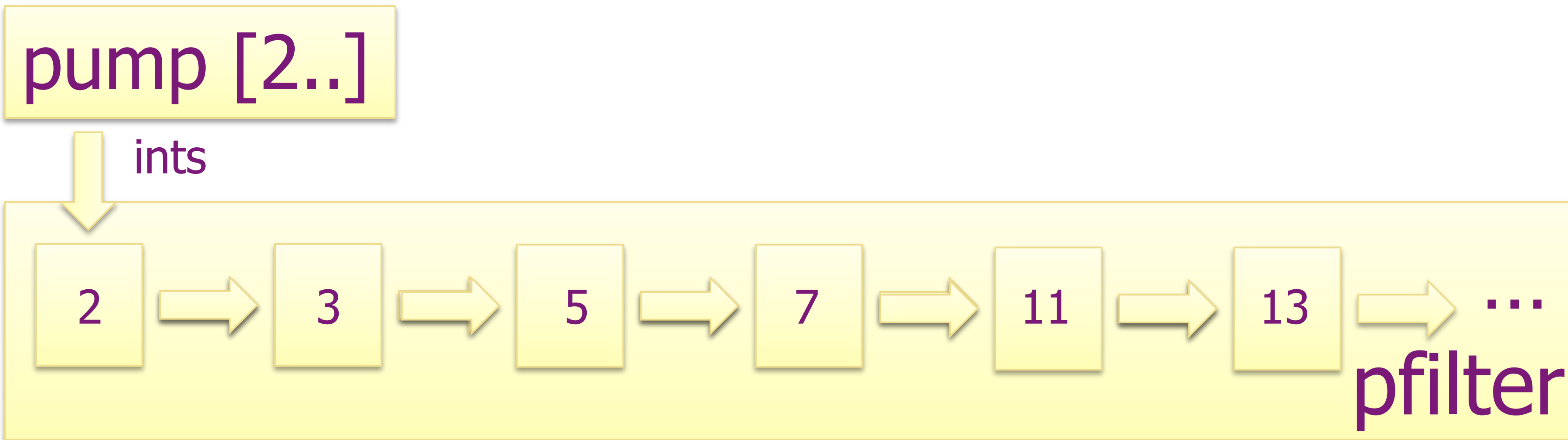
```
sieve      =   do  ints <- newChan
                   out  <- newChan
                   pump [2..] ints
                     <||> (ints `pfilter` out)
                     <||> drain out

pfilter      :: Pipe Int Int
pfilter i o = do  p <- input i
                  output o p
                  (filterChan (divis p) >>> pfilter) i o
  where divis n m = (m `mod` n) /= 0
```

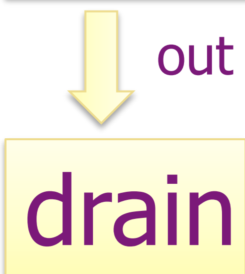
The Sieve of Eratosthenes:




```
sieve    =   do  ints <- newChan
                 out  <- newChan
                 pump [2..] ints
                    <||> (ints `pfilter` out)
                    <||> drain out
```

The Sieve of Eratosthenes:

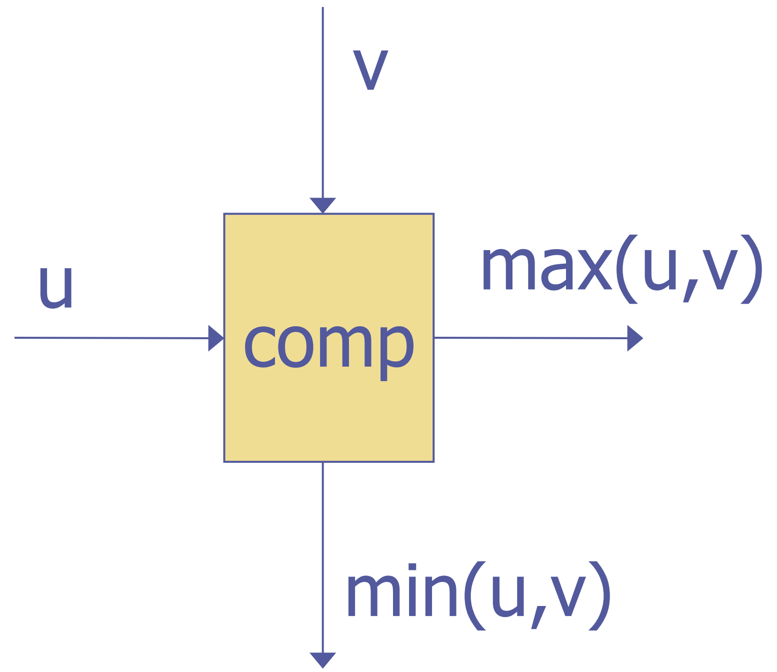




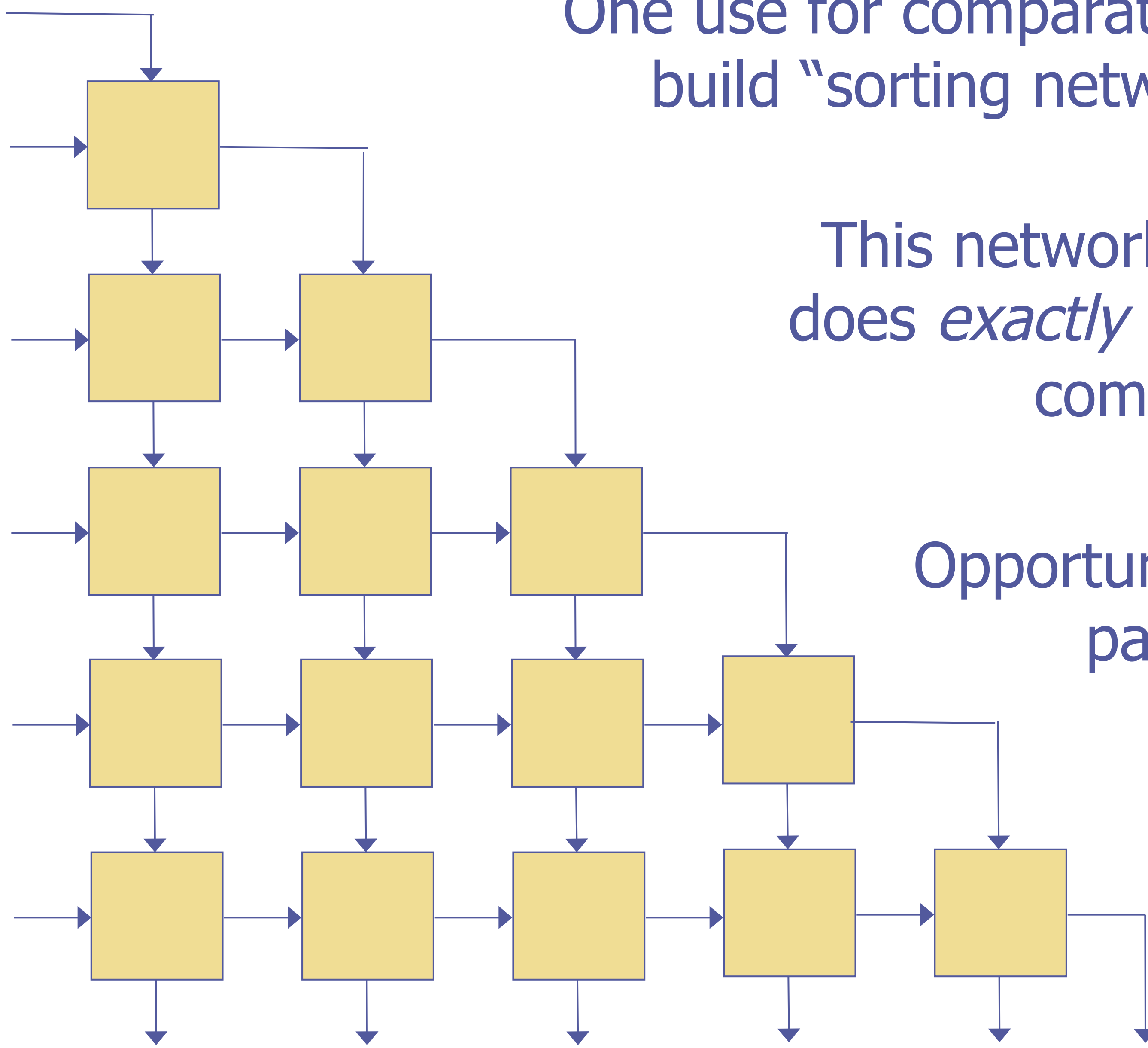

```
pfilter      :: Pipe Int Int
pfilter i o = do  p <- input i
                  output o p
                  (filterChan (divis p)
                     >>> pfilter) i o
```

Comparators:

Suppose that we have access to a supply of comparator components, each of which can be used to arrange two given values into sorted order.




```
comparator x y lo hi
  = loop (do (u,v) <- fork (input x) (input y)
             fork (output lo (min u v))
                  (output hi (max u v)))
```

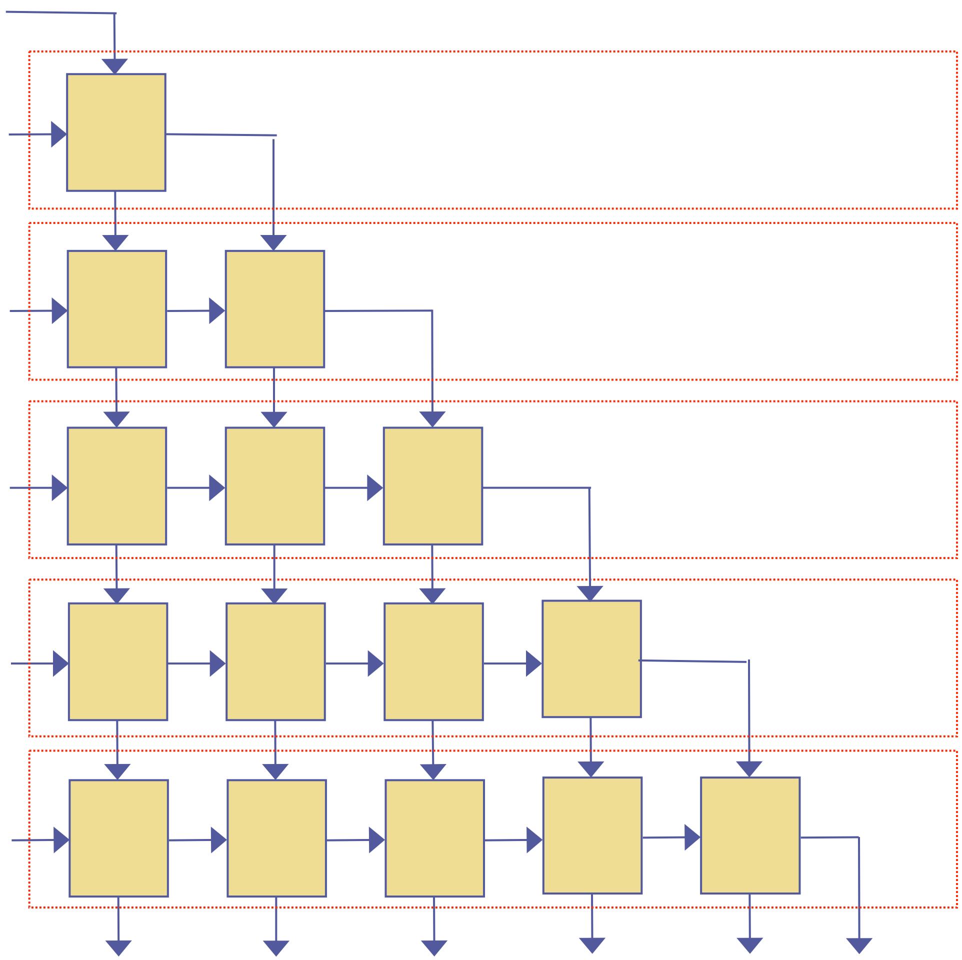


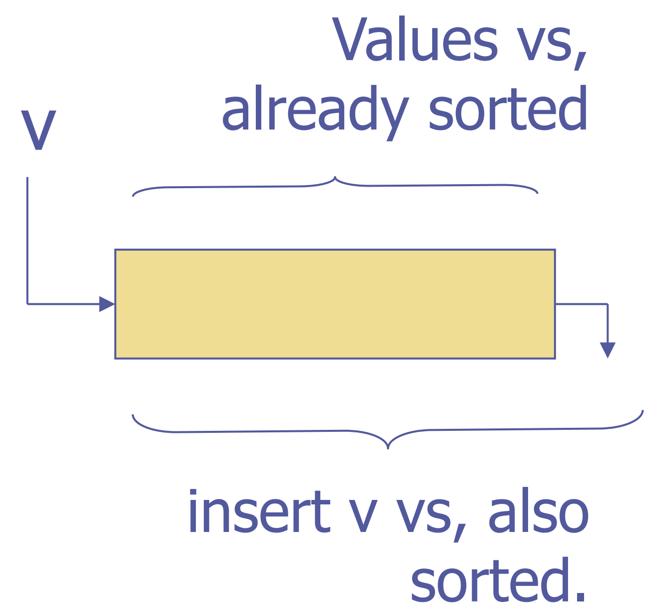
One use for comparators is to build "sorting networks" …

This network always does *exactly* n(n-1)/2 comparisons

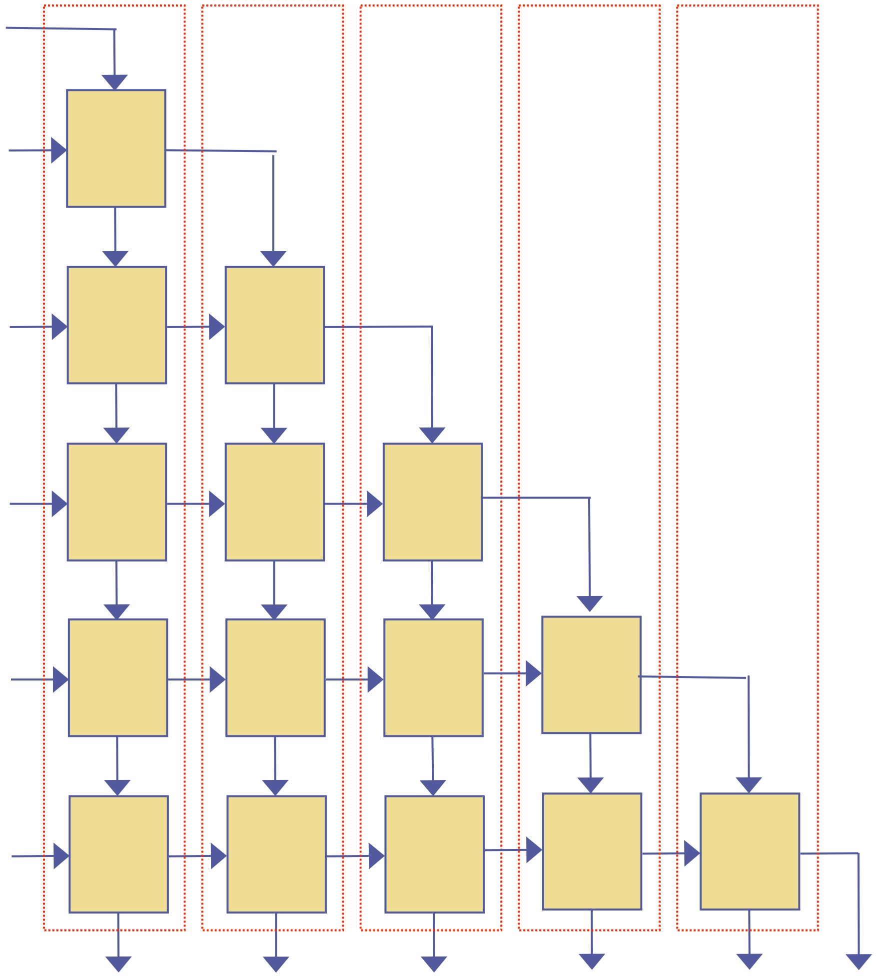
Opportunities for parallelism abound!

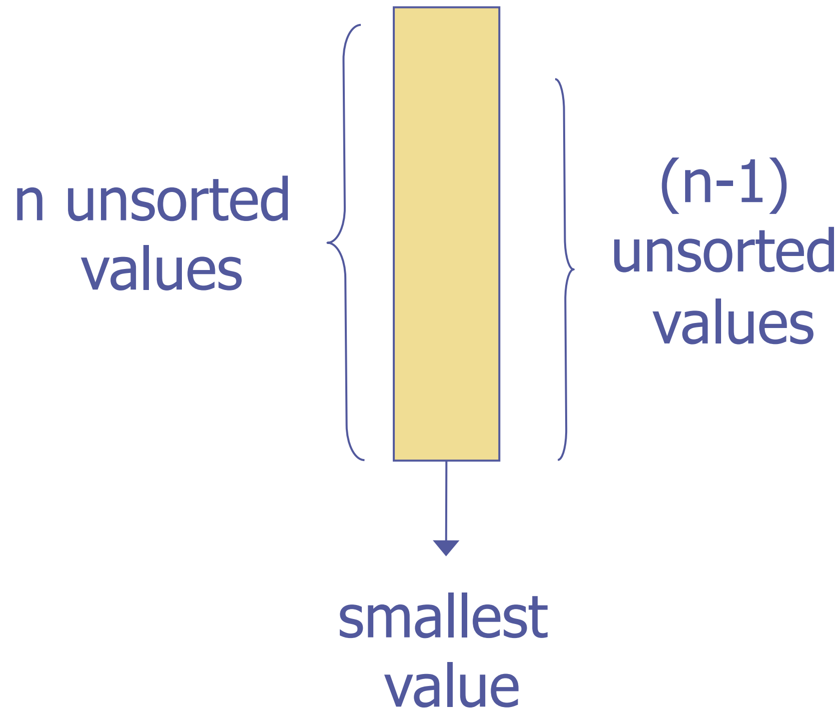
Insertion Sort:





Selection Sort:

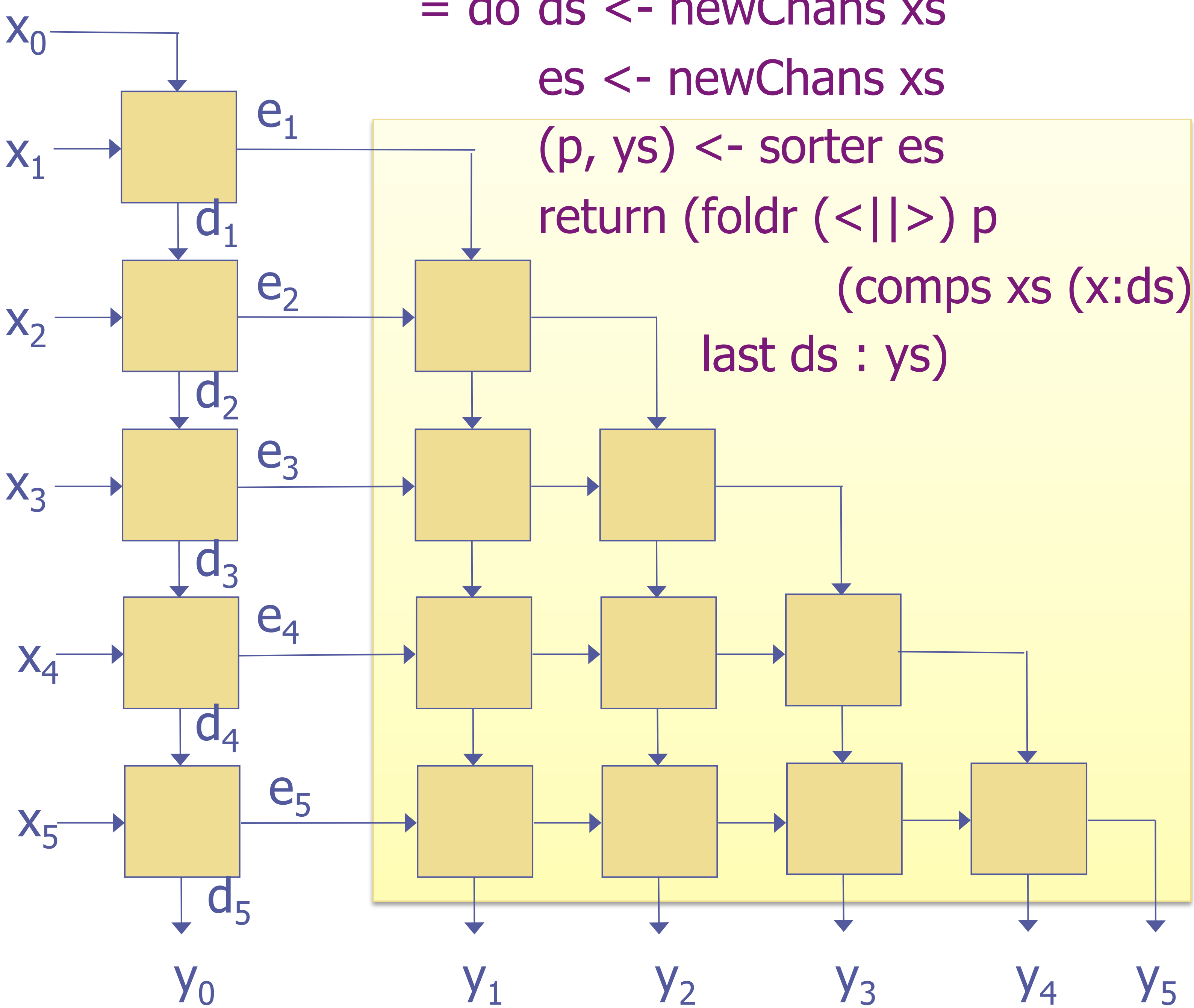




Constructing a Network:

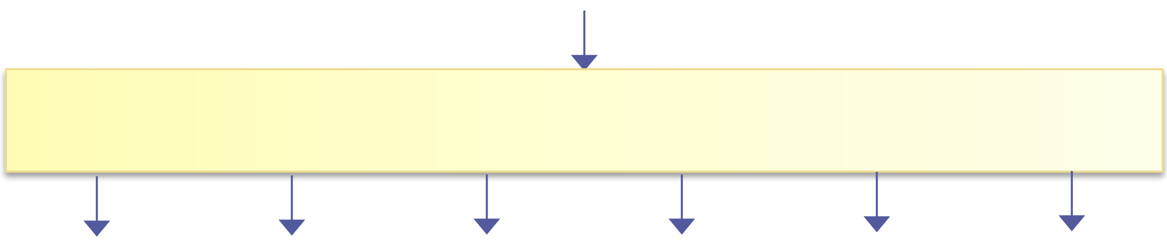
```
sorter        :: Ord a => [Chan a] -> C (C(), [Chan a])
sorter [x]  = return (done, [x])
sorter (x:xs)
  = do  ds <- newChans xs
        es <- newChans xs
        (p, ys) <- sorter es
        return (foldr (<||>) p
                   (zipWith4 comparator xs (x:ds) ds es),
                 last ds : ys)
```

```
sorter [x]     = return (done, [x])
sorter (x:xs)
  = do ds <- newChans xs
       es <- newChans xs
       (p, ys) <- sorter es
       return (foldr (<||>) p
                  (comps xs (x:ds) ds es),
            last ds : ys)
```

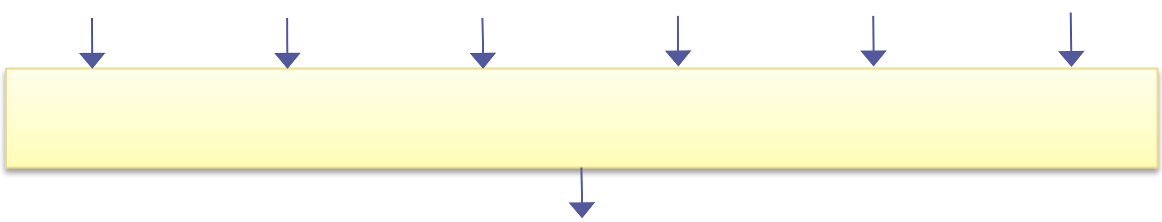


Spread and Gather:

```
spread        :: Chan [a] -> [Chan a] -> C ()
spread c cs   = loop (input c >>= (parCmds . zipWith output cs))
```

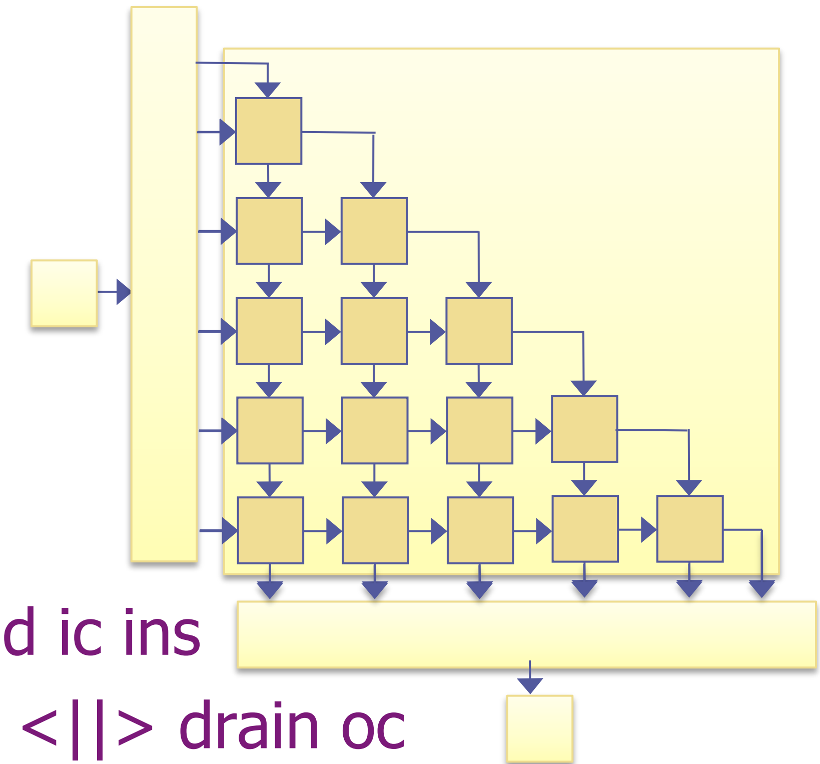


```
gather        :: [Chan a] -> Chan [a] -> C ()
gather cs c   = loop (parList (map input cs) >>= output c)
```



Testing a Network:

```
test ns
  = do ic         <- newChan
       ins        <- newChans ns
       (net, outs) <- sorter ins
       oc         <- newChan
       pump (perms ns) ic <||> spread ic ins
         <||> net <||> gather outs oc <||> drain oc
```



```
Main> execute (test [1..3])
[1,2,3]
[1,2,3]
[1,2,3]
[1,2,3]
[1,2,3]
[1,2,3]
Program error: deadlock!
Main>
```

Summary:

- A continuation describes "the rest of the program"
- Continuations can be used to implement important programming abstractions: concurrency, exceptions, exits, etc… (Indeed, some people consider them too powerful, much like gotos …)
- Some languages provide direct support for continuations
- Other languages allow us to simulate the use of continuations with functions
- [Aside: The concurrency monad in these slides is an updated version of the library described in "Implicit and Explicit Parallel Programming in Haskell," Jones and Hudak, 1993, which is available from my web pages.]