stream: an abstract model of a sequence of values over time
Implementing streams using lazy lists
A list is either:

- empty
- a value “cons” another list
A lazy list is either:

- empty
- a value “cons” a function that returns a lazy list
(s-cons first rest)

→

(cons first (lambda () rest))
(s-cons first rest)

→

(cons first (lambda () rest))
Streams are a way to decouple the production and consumption of data (in a program)
Consequences

1. Don’t need to access all input data at once.
(define (s-sum-tail numbers acc)
  (if (s-null? numbers)
      acc
      (s-sum-tail (s-rest numbers)
                  (+ (s-first numbers) acc)))))

(define (sum-tail numbers acc)
  (if (null? numbers)
      acc
      (sum-tail (rest numbers)
                (+ (first numbers) acc)))))
(define (s-sum-tail numbers acc)
  (if (s-null? numbers)
      acc
      (s-sum-tail (s-rest numbers)
                  (+ (s-first numbers) acc))))

sumTail numbers !acc =
  if null numbers
  then acc
  else sumTail (tail numbers)
              (head numbers + acc)
Consequences

2. Facilitate parallelization of stages in a pipeline
Consequences

2. Facilitate parallelization of stages in a pipeline
Consequences

3. Represent **infinite** sequences (in finite time and space!)
Choices and backtracking
the ambiguous operator -<
> (\texttt{-< 1 2 3})
1
> (\texttt{next})
2
> (\texttt{next})
3
> (\texttt{next})
\texttt{done}
Aside: time limit for Ex6

```bash
wolf:~$ time runhaskell Ex6.hs
+++ OK, passed 1 tests.
+++ OK, passed 1 tests.
+++ OK, passed 1 tests.
+++ OK, passed 1 tests.
+++ OK, passed 1 tests.

real  0m11.459s <--- 12s * 20 = 240s
user  0m11.294s
sys   0m0.150s
```
Code walkthrough

Warning: mutation ahead!
> (+ 10 (<< 1 2 3))
11
> (next)
12
> (next)
13
> (next)
'done
**Problem**: can’t just store choices (\(-< 1 \ 2 \ 3\))

Also need to store *execution context* (+ 10 [] )
Execution context (of an expression): a representation of what remains to be computed after the expression is evaluated

Also known as the expression’s continuation
In the traditional stack-based model, the continuation is the call stack
In (abstract) pure functional programming, the continuation is a **unary function** derived from the enclosing expression.
Examples!
let/cc ("let current continuation")

(let/cc <id>
    <expr> ...
)

1. Binds <id> to the continuation of the let/cc expression.
2. Evaluates each <expr> ... and returns the last one (like begin).