Announcements

There is a **lecture instead of a lab** on Monday Oct 22.

Midterm on Wednesday Oct 24 – check website for details!

stream: an abstract model of a sequence of values over time

Lazy list:

- empty
- a value “cons” another (lazy) list

But the *cons* is lazy here!

```
(define s-null 's-null)
(define (s-null? stream) (equal? stream 's-null))

(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))


(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))
```

```
(define s-null 's-null)
(define (s-null? stream) (equal? stream 's-null))

(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))


(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))
```

Streams are a way to decouple the production and consumption of data.

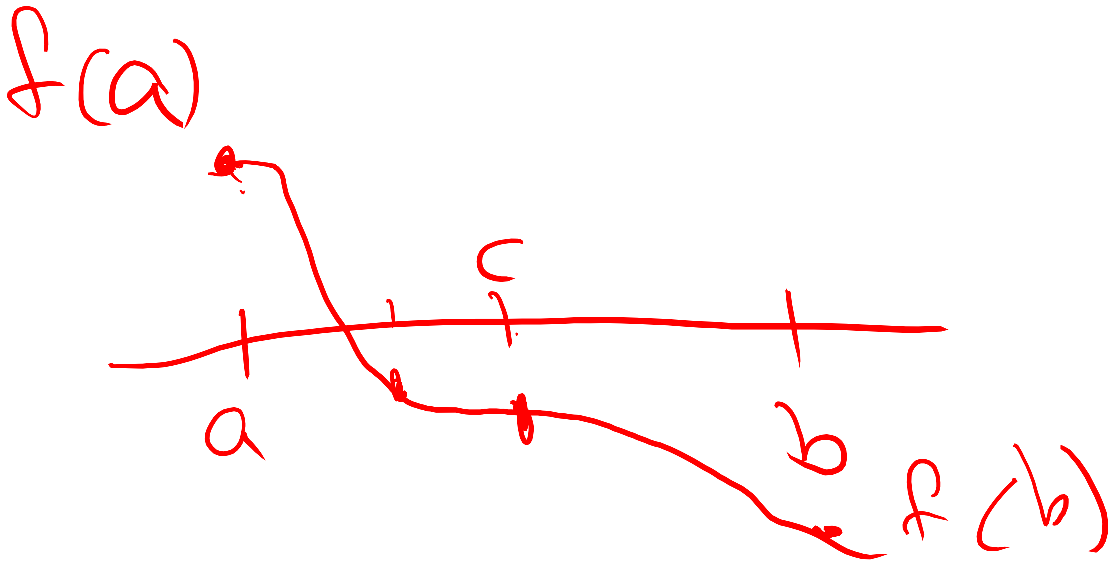
Case study: range vs. in-range

Taking production to the extreme.

The bisection method (iterative)

```
def bisect(f, tol, a, b):
    # Precondition: f(a) and f(b) have different signs.
    c = (a + b)/2
    while abs(f(c)) >= tol:
        if sign(f(a)) == sign(f(c)):
            a = c
        else:
            b = c
        c = (a + b)/2

    return c
```



The bisection method (tail recursive)

```
(define (bisect f tol a b)
  (let* ([c (/ (+ a b) 2)]
         [y (f c)])
    (if (< (abs y) tol)
        c
        (if (equal? (sgn (f a)) (sgn y))
            (bisect f tol c b)
            (bisect f tol a c)))))
```

The bisection method (stream version)

```
(define (bisect f tol a b)
  (let* ([c (/ (+ a b) 2)]
         [y (f c)])
    (if (< (abs y) tol)
        c
        (if (equal? (sgn (f a)) (sgn y))
            (bisect f tol c b)
            (bisect f tol a c)))))
```

BWAH

But what about Haskell?

Choices and backtracking

the ambiguous operator `-<`

```
> (-< 1 2 3)
1
> (next)
2
> (next)
3
> (next)
‘done
```

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 stack-based model of program execution,
the continuation of an expression is the state of the call stack after the expression has been evaluated.

In pure functional programming,
the continuation is a unary function derived from the enclosing expression.

E = (+ (* 2 3) (- 5 4))

5 :

(+ 6 (- _ 4))

Continuation of...

(+ (* 2 _) (- 5 4))

3

(* 2 3)

(+ _ (- 5 4))

5

+

(_ (* 2 3) (- 5 4))

(+ (* 2 3) (- 5 4))

_

`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`).

Note: `let/cc` is *dynamic*.

The “current continuation” is computed when the `let/cc` is evaluated.

```
> (+ 10 (-< 1 2 3))
11
> (next)
12
> (next)
13
> (next)
'done
```