Today we will learn...

- streams
- functional patterns applied to streams
- compose stream operations
Streams
Stream

A stream is an infinite sequence of values.

Did you know? The concept of streams is also used in:

- Reactive programming (e.g., a stream of GUI events for Android development)
- Stream processing for digital signal processing (e.g., image/video codecs with the language StreamIt)
- Unix pipes (e.g., a pipeline of Unix processes producing and consuming a stream of data)
- See also Microsoft LINQ and Java 8 streams
Streams in Racket

A stream can be recursively defined as a pair holds a value and another stream:

\[
\text{stream} = (\text{cons some-value (thunk stream)})
\]

Powers of two

\[
(\text{cons 1 (thunk (cons 2 (thunk (cons 4 (thunk ...))))))}
\]

Visually

\[
1 \quad 2 \quad 4 \quad ...
\]

Using streams

\[
\begin{align*}
\text{(check-equal? 1 (car (powers-of-two)))} & \quad \text{; the 1st element of the stream} \\
\text{(check-equal? 2 (car ((cdr (powers-of-two)))))} & \quad \text{; the 2nd element of the stream} \\
\text{(check-equal? 4 (car ((cdr ((cdr (powers-of-two)))))))} & \quad \text{; the 3rd element of the stream}
\end{align*}
\]
Revisiting our example with helper functions

; Retrieves the current value of the stream
(define (stream-get stream) (car stream))

; Retrieves the thunk and evaluates it, returning a thunk
(define (stream-next stream) ((cdr stream)))

(check-equal? 1 (stream-get (powers-of-two)))
(check-equal? 2 (stream-get (stream-next (powers-of-two))))
(check-equal? 4 (stream-get (stream-next (stream-next (powers-of-two))))))
Programming with streams

Let us write a function that given a stream and a predicate, counts how many times a predicate holds true until it becomes false.

Spec

(check-equal? 3 (count-until (powers-of-two) (lambda (x) (< x 8))))
(check-equal? 0 (count-until (powers-of-two) (lambda (x) (<= x 0))))
(check-equal? 3 (count-until (powers-of-two) (curryr < 8))) ; Reverse Currying
(check-equal? 0 (count-until (powers-of-two) (curryr <= 0))) ; Reverse Currying
Programming with streams

Let us write a function that given a stream and a predicate, counts how many times a predicate holds true until it becomes false.

Spec

(check-equal? 3 (count-until (powers-of-two) (lambda (x) (< x 8))))
(check-equal? 0 (count-until (powers-of-two) (lambda (x) (<= x 0))))
(check-equal? 3 (count-until (powers-of-two) (curryr < 8)) prop 8)) ; Reverse Currying
(check-equal? 0 (count-until (powers-of-two) (curryr <= 0)) prop 0)) ; Reverse Currying

Solution

(define (count-until stream pred)
  (define (count-until-iter s count)
    (cond [(pred (stream-get s)) (count-until-iter (stream-next s) (+ count 1))]
          [else count]))
  (count-until-iter stream 0))
Example: powers of two

Implement the stream \texttt{powers-of-two}
Example: powers of two

Implement the stream powers-of-two

Solution

```
(define (powers-of-two)
  (define (powers-of-two-iter n)
    (thunk
     (cons n (powers-of-two-iter (* 2 n)))))
  ((powers-of-two-iter 1)))
```
Example: constant

Implement a function `const` that given a value it returns a stream that always yields that value.

```scheme
(check-equal? 20 (stream-get (const 20)))
(check-equal? 20 (stream-get (stream-next (const 20))))
(check-equal? 20 (stream-get (stream-next (stream-next (const 20))))))
```
Example: constant

Implement a function \texttt{const} that given a value it returns a stream that always yields that value.

\begin{verbatim}
(check-equal? 20 (stream-get (const 20)))
(check-equal? 20 (stream-get (stream-next (const 20))))
(check-equal? 20 (stream-get (stream-next (stream-next (const 20)))))
\end{verbatim}

Solution

\begin{verbatim}
(define (const v)
    (define (const-iter) (cons v const-iter))
    (const-iter))
\end{verbatim}
Common mistakes (1)

\[
\text{(define (const v)} \\
\text{ (define const-iter (cons v const-iter))} \\
\text{(const-iter))}
\]
Common mistakes (1)

```
(define (const v)
  (define const-iter (cons v const-iter))
  (const-iter))
```

c-iter is not a thunk. The error is that const-iter is not defined (as the body of the definition is evaluated).
Common mistakes (2)

```scheme
(define (const v)
  (define (const-iter) (cons v (const-iter)))
  (const-iter))
```
Common mistakes (2)

\[
\begin{align*}
& \text{(define (const v)} \\
& \quad \text{(define (const-iter) (cons v (const-iter)))} \\
& \quad \text{(const-iter))}
\end{align*}
\]

in the body of \text{const-iter} the thunk \text{const-iter} is evaluated. This function does not terminate.
Streams in Racket

A stream can be recursively defined as a pair holds a value and another stream:

\[ \text{stream} = (\text{cons some-value } (\text{thunk stream})) \]

A stream of natural numbers:

\[(\text{cons 0 } (\text{thunk (cons 1 } (\text{thunk (cons 2 } (\text{thunk ...})))))))\]

Visually:

0 1 2 3 4 5 6 ...

Using streams:

\[
\begin{align*}
\text{(check-equal? 0 (stream-get (naturals)))} \\
\text{(check-equal? 1 (stream-get (stream-next (naturals))))} \\
\text{(check-equal? 2 (stream-get (stream-next (stream-next (naturals)))))}
\end{align*}
\]
Natural numbers

Implement the stream of non-negative integers

\[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad ... \]

Spec

```racket
#lang racket
(require rackunit)

(define s0 (naturals))
(check-equal? 0 (stream-get s0))

(define s1 (stream-next s0))
(check-equal? 1 (stream-get s1))

(define s2 (stream-next s1))
(check-equal? 2 (stream-get s2))
```
Natural numbers

Implement the stream of non-negative integers

0 1 2 3 4 5 6 7 ...

Spec

```racket
#lang racket
(require rackunit)

(define s0 (naturals))
(define s1 (stream-next s0))
(define s2 (stream-next s1))

(check-equal? 0 (stream-get s0))
(check-equal? 1 (stream-get s1))
(check-equal? 2 (stream-get s2))
```

Solution

```racket
(define (naturals)
    (define (naturals-iter n)
        (thunk
            (cons n (naturals-iter (+ n 1)))))
    ((naturals-iter 0))
)```
Map for streams

- Given a stream $s$ defined as
  
e_0 \ e_1 \ e_2 \ e_3 \ e_4 \ ...

- and a function $f$ the stream $(\text{stream-map } f \ s)$ should yield
  
  $(f \ e_0) \ (f \ e_1) \ (f \ e_2) \ (f \ e_3) \ (f \ e_4) \ ...$
Map for streams

Spec

```racket
#lang racket
(require rackunit)

(define s0
  (stream-map (curry + 2) (naturals)))
(check-equal? (stream-get s0) 2)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 3)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```
Map for streams

Spec

```racket
#lang racket
(require rackunit)

(define s0
  (stream-map (curry + 2) (naturals)))
(check-equal? (stream-get s0) 2)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 3)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```racket
(define (stream-map f s)
  (define (stream-map-iter s)
    (thunk
     (cons
      (f (stream-get s))
      (stream-map-iter (stream-next s)))))
  ((stream-map-iter s)))
```
Even naturals

Build a stream of even numbers. Tip: use stream-map and naturals.

0 2 4 6 8 10 12 ...

Spec

```racket
#lang racket
(require rackunit)
(define s0 (even-naturals))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```
Even naturals

- Build a stream of even numbers. Tip: use `stream-map` and `naturals`.

0 2 4 6 8 10 12 ...

Spec

```racket
#lang racket
(require rackunit)
(define s0 (even-naturals))
(check-equal? (stream-get s0) 0)

(define s1 (stream-next s0))
(check-equal? (stream-get s1) 2)

(define s2 (stream-next s1))
(check-equal? (stream-get s2) 4)
```

Solution

```racket
(define (even-naturals)
  (stream-map
   (curry * 2)
   (naturals)))
```
Zip two streams

Given a stream $s_1$ defined as

$e_1 \ e_2 \ e_3 \ e_4 \ ...$

and a stream $s_2$ defined as

$f_1 \ f_2 \ f_3 \ f_4 \ ...$

the stream $(\text{stream-zip } s_1 \ s_2)$ returns

$(\text{cons } e_1 \ f_1) \ (\text{cons } e_2 \ f_2) \ (\text{cons } e_3 \ f_3) \ (\text{cons } e_4 \ f_4) \ ...$
Zip for streams

Spec

```racket
#lang racket
(require rackunit)
(define s0
  (stream-zip (naturals) (even-naturals)))
(check-equal? (stream-get s0) (cons 0 0))
(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))
(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```
Zip for streams

Spec

```racket
#lang racket
(require rackunit)
(define s0
  (stream-zip (naturals) (even-naturals)))
(check-equal? (stream-get s0) (cons 0 0))
(define s1 (stream-next s0))
(check-equal? (stream-get s1) (cons 1 2))
(define s2 (stream-next s1))
(check-equal? (stream-get s2) (cons 2 4))
```

Solution

```racket
(define (stream-zip s1 s2)
  (define (stream-zip-iter s1 s2)
    (thunk
      (cons
        (cons (stream-get s1)
          (stream-get s2))
        (stream-zip-iter
          (stream-next s1)
          (stream-next s2)))))))
((stream-zip-iter s1 s2)))
```