Today we will learn...

- streams
- functional patterns applied to streams
- compose stream operations
Streams
A stream is an infinite sequence of values.

**Did you know?** The concept of streams is also used in:
- Reactive programming (eg, a stream of GUI events for Android development)
- Stream processing for digital signal processing (eg, image/video codecs with the language StreamIt)
- Unix pipes (eg, a pipeline of Unix process producing and consuming a stream of data)
- See also Microsoft LINQ and Java 8 streams
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 2 4 ...

Using streams:

\[
\begin{align*}
(\text{check-equal?} & \ 1 \ (\text{car} \ (\text{powers-of-two}))) \quad ; \text{the 1st element of the stream} \\
(\text{check-equal?} & \ 2 \ (\text{car} \ ((\text{cdr} \ (\text{powers-of-two}))))) \quad ; \text{the 2nd element of the stream} \\
(\text{check-equal?} & \ 4 \ (\text{car} \ ((\text{cdr} \ ((\text{cdr} \ (\text{powers-of-two}))))))) \quad ; \text{the 3rd element of the stream}
\end{align*}
\]
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))))))
Count elements in stream
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

\[
\begin{align*}
&\text{(check-equal? 3 (count-until (powers-of-two) (lambda (x) (< x 8))))} \\
&\text{(check-equal? 0 (count-until (powers-of-two) (lambda (x) (<= x 0))))} \\
&\text{(check-equal? 3 (count-until (powers-of-two) (curryr < 8))) ; Reverse Currying} \\
&\text{(check-equal? 0 (count-until (powers-of-two) (curryr <= 0))) ; Reverse Currying}
\end{align*}
\]
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

```scheme
(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
```

Solution

```scheme
(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))
```
Implementing powers of two
Example: powers of two

Implement the stream 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)))
```
The stream of constants
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 `const` that given a value it returns a stream that always yields that value.

\[
\begin{align*}
\text{Solution} & \quad (\text{define} \ (\text{const} \ v)) \\
& \quad (\text{define} \ (\text{const-iter}) \ (\text{cons} \ v \ \text{const-iter})) \\
& \quad (\text{const-iter})
\end{align*}
\]
Common mistakes (1)

```
(define (const v)
  (define const-iter (cons v const-iter))
  (const-iter))
```
Common mistakes (1)

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

const-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)

\[
\begin{align*}
&\text{(define (const v)} \\
&\quad \text{(define (const-iter) (cons v (const-iter)))} \\
&\quad \text{(const-iter))}
\end{align*}
\]
Common mistakes (2)

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

In the body of `const-iter` the thunk `const-iter` is evaluated. This function does not terminate.
The stream of natural numbers
Streams in Racket

A stream can be recursively defined as a pair holds a value and another stream:
\[
\text{stream} = (\text{cons some-value (thunk stream)})
\]

A stream of natural numbers:
\[
(\text{cons } 0 (\text{thunk (cons } 1 (\text{thunk (cons } 2 (\text{thunk } \ldots))))))
\]
Visually:
\[
0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad \ldots
\]
Using streams:

```
(check-equal? 0 (stream-get (naturals)))
(check-equal? 1 (stream-get (stream-next (naturals))))
(check-equal? 2 (stream-get (stream-next (stream-next (naturals))))))
```
Natural numbers

Implement the stream of non-negative integers

\[ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ \ldots \]

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

#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))

Solution

(define (naturals)
  (define (naturals-iter n)
    (thunk
     (cons n (naturals-iter (+ n 1))))))
  ((naturals-iter 0)))
The map stream
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

#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))
```
The stream of even numbers
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)))
```
Merge two streams
Zip two streams

- Given a stream \( s_1 \) defined as
  \[ e_1 \ e_2 \ e_3 \ e_4 \ldots \]

- and a stream \( s_2 \) defined as
  \[ f_1 \ f_2 \ f_3 \ f_4 \ldots \]

- the stream \((\text{stream-zip } s_1 \ s_2)\) returns
  \[(\text{cons } e_1 \ f_1) \ (\text{cons } e_2 \ f_2) \ (\text{cons } e_3 \ f_3) \ (\text{cons } e_4 \ f_4) \ldots \]


Spec

#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)))
```