5: Lists

A structure lets us group, in a Racket program, a **fixed** number of values (as many as there are fields in the structure).

A structure lets us group, in a Racket program, a **fixed** number of values (as many as there are fields in the structure).

What if the number of values isn't known at the time the program is written?

Perhaps it is learned later, or computed during the running of the program.

A structure lets us group, in a Racket program, a **fixed** number of values (as many as there are fields in the structure).

What if the number of values isn't known at the time the program is written?

Perhaps it is learned later, or computed during the running of the program.

Just as we used structures to represent unbounded numbers, we can use structures to represent unbounded data.

First, we give a mathematical definition.

We would like to represent a sequence of values, such as $3, 5, 7, 5, 4$ (this will be our running example).

In this example, the values are all natural numbers, but in general we can mix different types of values.

We would like to represent a sequence of values, such as $3, 5, 7, 5, 4$ (this will be our running example).

In this example, the values are all natural numbers, but in general we can mix different types of values.

First, we define in general terms what a sequence is.

We adapt the definition of natural number:
either 0 or $n + 1$, where n is a natural number.

We would like to represent a sequence of values, such as $3, 5, 7, 5, 4$ (this will be our running example).

In this example, the values are all natural numbers, but in general we can mix different types of values.

First, we define in general terms what a sequence is.

We adapt the definition of natural number:
either 0 or $n + 1$, where n is a natural number.

A sequence is either the empty sequence, or it is a value followed by a sequence.

We would like to represent a sequence of values, such as $3, 5, 7, 5, 4$ (this will be our running example).

In this example, the values are all natural numbers, but in general we can mix different types of values.

First, we define in general terms what a sequence is.

We adapt the definition of natural number:
either 0 or $n + 1$, where n is a natural number.

A sequence is either the empty sequence, or it is a value followed by a sequence.

For simplicity, we will use sequences of integers as examples, though a sequence may mix elements of different types.

There is no agreed-upon mathematical symbol to represent the empty sequence.

We will use ϵ, for “empty”.

There is no agreed-upon mathematical symbol to represent the empty sequence.

We will use ϵ, for "empty".

A sequence S is either ϵ or it is a value v followed by a sequence S'.

- In the case where S' is ϵ, we write S as v.
- Otherwise, we write S as v, S'.

There is no agreed-upon mathematical symbol to represent the empty sequence.

We will use ϵ, for “empty”.

A sequence S is either ϵ or it is a value v followed by a sequence S'.

- In the case where S' is ϵ, we write S as v.
- Otherwise, we write S as v, S'.

Our example $3, 5, 7, 5, 4$ satisfies this definition:

- ϵ, the empty sequence, is a sequence.
- 4 is a sequence, because it is the value 4 followed by ϵ.
- $5, 4$ is a sequence, because it is the value 5 followed by the sequence 4.
- And so on.

We adapt our idea of representing natural numbers.

```
(define-struct Empty ())
(define-struct Cons (fst rst))
```

Data definition:
An **S-list** is either a `(make-Empty)` or it is `(make-Cons v slist)`, where `v` is a Racket value and `slist` is an S-list.

We adapt our idea of representing natural numbers.

```
(define-struct Empty ())
(define-struct Cons (fst rst))
```

Data definition:
An **S-list** is either a `(make-Empty)` or it is `(make-Cons v slist)`, where `v` is a Racket value and `slist` is an S-list.

The S-list `make-Empty` represents the sequence ϵ.

The S-list `(make-Cons v slist)` represents the sequence v, S where `slist` represents S.

We adapt our idea of representing natural numbers.

```
(define-struct Empty ())
(define-struct Cons (fst rst))
```

Data definition:
An **S-list** is either a `(make-Empty)` or it is `(make-Cons v slist)`, where `v` is a Racket value and `slist` is an S-list.

The S-list `make-Empty` represents the sequence ϵ.

The S-list `(make-Cons v slist)` represents the sequence v, S where `slist` represents S.

The sequence 4 is represented by `(make-Cons 4 (make-Empty))`.

The sequence $5, 4$ is represented by
`(make-Cons 5 (make-Cons 4 (make-Empty)))`.

Our example as an S-list

The S-list representation of the sequence $3, 5, 7, 5, 4$ is

```
(make-Cons 3
  (make-Cons 5
    (make-Cons 7
      (make-Cons 5
        (make-Cons 4
          (make-Empty))))))
```

If we give this value a name, we can work with it using the accessor functions for the structures we have defined.

The S-list representation of the sequence $3, 5, 7, 5, 4$ is

```
(make-Cons 3
  (make-Cons 5
    (make-Cons 7
      (make-Cons 5
        (make-Cons 4
          (make-Empty))))))
```

If we give this value a name, we can work with it using the accessor functions for the structures we have defined.

```
(define slst (make-Cons 3 (make-Cons 5 ...)))

(Cons-fst slist) ⇒ 3
(Cons-rst slist) ⇒ (make-Cons 5 ...)
```

Having defined the S-list representation of sequences, we now abandon it, because Racket has built-in support for a similar representation.

- Where we have used `make-Empty`, Racket uses the value `'()`. Racket also provides the name `empty` bound to this value.
- Where we have used `make-Cons`, Racket uses `cons`.

Having defined the S-list representation of sequences, we now abandon it, because Racket has built-in support for a similar representation.

- Where we have used `make-Empty`, Racket uses the value `'()`. Racket also provides the name `empty` bound to this value.
- Where we have used `make-Cons`, Racket uses `cons`.

The representations thus constructed are called **lists**.

Data definition:
A list is either `empty` or it is `(cons fst rst)`, where `fst` is a Racket value and `rst` is a list.

Having defined the S-list representation of sequences, we now abandon it, because Racket has built-in support for a similar representation.

- Where we have used `make-Empty`, Racket uses the value `'()`. Racket also provides the name `empty` bound to this value.
- Where we have used `make-Cons`, Racket uses `cons`.

The representations thus constructed are called **lists**.

Data definition:
A list is either `empty` or it is `(cons fst rst)`, where `fst` is a Racket value and `rst` is a list.

Racket enforces the restriction that the second argument to `cons` must be a list.

Having defined the S-list representation of sequences, we now abandon it, because Racket has built-in support for a similar representation.

- Where we have used `make-Empty`, Racket uses the value `'()`. Racket also provides the name `empty` bound to this value.
- Where we have used `make-Cons`, Racket uses `cons`.

The representations thus constructed are called **lists**.

Data definition:
A list is either `empty` or it is `(cons fst rst)`, where `fst` is a Racket value and `rst` is a list.

Racket enforces the restriction that the second argument to `cons` must be a list.

A list may contain values of different types (though in our examples we will stick with integers for now).

Our example as a list

The list representation of the sequence $3, 5, 7, 5, 4$ is

```
(cons 3
  (cons 5
    (cons 7
      (cons 5
        (cons 4 empty)))))
```

Our example as a list

The list representation of the sequence $3, 5, 7, 5, 4$ is

```
(cons 3
  (cons 5
    (cons 7
      (cons 5
        (cons 4 empty)))))
```

To create a list in a more concise fashion, we can use the `list` function.

```
(list 3 5 7 5 4)
```

Accessors:
We access the first element of a nonempty S-list with `Cons-fst`.

We access the first element of a nonempty list with `first`.

Accessors:
We access the first element of a nonempty S-list with `Cons-fst`.

We access the first element of a nonempty list with `first`.

We access the rest of a nonempty S-list with `Cons-rst`.

We access the rest of a nonempty list with `rest`.

Accessors and type predicates for lists

Accessors:

We access the first element of a nonempty S-list with `Cons-fst`.

We access the first element of a nonempty list with `first`.

We access the rest of a nonempty S-list with `Cons-rst`.

We access the rest of a nonempty list with `rest`.

Type Predicates:

The list equivalent of `Empty?` is `empty?`.

The list equivalent of `Cons?` is `cons?`.

Accessors:

We access the first element of a nonempty S-list with `Cons-fst`.

We access the first element of a nonempty list with `first`.

We access the rest of a nonempty S-list with `Cons-rst`.

We access the rest of a nonempty list with `rest`.

Type Predicates:

The list equivalent of `Empty?` is `empty?`.

The list equivalent of `Cons?` is `cons?`.

Racket also provides the predicate `list?`.

What is the equivalent of `list?`, but for S-lists?

Accessors:
We access the first element of a nonempty S-list with `Cons-fst`.

We access the first element of a nonempty list with `first`.

We access the rest of a nonempty S-list with `Cons-rst`.

We access the rest of a nonempty list with `rest`.

Type Predicates:
The list equivalent of `Empty?` is `empty?`.

The list equivalent of `Cons?` is `cons?`.

Racket also provides the predicate `list?`.

What is the equivalent of `list?`, but for S-lists?

```
(define (List? v)
  (or (Empty? v) (Cons? v)))
```

We can develop a template from the data definition for lists in the same way that we did for Nats and natural numbers.

A template for recursive functions that consume a list

We can develop a template from the data definition for lists in the same way that we did for Nats and natural numbers.

Data definition:
A list `lst` is either `empty` or it is `(cons fst rst)`, where `fst` is a Racket value and `rst` is a list.

A template for recursive functions that consume a list

We can develop a template from the data definition for lists in the same way that we did for Nats and natural numbers.

Data definition:
A list `lst` is either `empty` or it is `(cons fst rst)`, where `fst` is a Racket value and `rst` is a list.

Given a nonempty list `lst`, `fst` is `(first lst)`, and `rst` is `(rest lst)`.

We can develop a template from the data definition for lists in the same way that we did for Nats and natural numbers.

Data definition:
A list `lst` is either `empty` or it is `(cons fst rst)`, where `fst` is a Racket value and `rst` is a list.

Given a nonempty list `lst`, `fst` is `(first lst)`, and `rst` is `(rest lst)`.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

Let's write the function `len` that computes the length of a list.

Let's write the function `len` that computes the length of a list.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

Let's write the function `len` that computes the length of a list.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

```
(define (len lst)
  (cond
    [(empty? lst) 0]
    [(cons? lst) (+ 1 (len (rest lst)))]))
```

A recursive function consuming a list

Let's write the function `len` that computes the length of a list.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

```
(define (len lst)
  (cond
    [(empty? lst) 0]
    [(cons? lst) (+ 1 (len (rest lst)))]))
```

The built-in Racket function `length` does the same thing. It is good practice to work out the implementation of built-in functions where possible, so that they don't seem magical.

```
(define (len lst)
  (cond
    [(empty? lst) 0]
    [(cons? lst) (+ 1 (len (rest lst)))]))

(len (cons 3 (cons 5 (cons 7 (cons 5 (cons 4 empty))))))
⇒* (+ 1 (len (cons 5 (cons 7 (cons 5 (cons 4 empty))))))
⇒* (+ 1 (+ 1 (len (cons 7 (cons 5 (cons 4 empty))))))
⇒* (+ 1 (+ 1 (+ 1 (len (cons 5 (cons 4 empty))))))
⇒* (+ 1 (+ 1 (+ 1 (+ 1 (len (cons 4 empty))))))
⇒* (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (len empty))))))
⇒* (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0)))))
⇒* 5
```

A function producing a list

The function `sqr-all` consumes a list of numbers and produces a list of the same length, with each element being the square of the corresponding element of the argument list.

```
(sqr-all (cons 4 (cons -2 (cons 3 empty))))
⇒  (cons 16 (cons 4 (cons 9 empty)))
```

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

The function `sqr-all` consumes a list of numbers and produces a list of the same length, with each element being the square of the corresponding element of the argument list.

```
(sqr-all (cons 4 (cons -2 (cons 3 empty))))
⇒  (cons 16 (cons 4 (cons 9 empty)))
```

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (first lst) ...
                 ... (sqr-all (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (first lst) ...
                 ... (sqr-all (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (first lst) ...
                 ... (sqr-all (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (sqr (first lst)) ...
                 ... (sqr-all (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (first lst) ...
                 ... (sqr-all (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (sqr (first lst)) ...
                 ... (sqr-all (rest lst)) ...]))
```

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (sqr (first lst))
                       (sqr-all (rest lst)))]))
```

A condensed trace of the sqr-all function

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (sqr (first lst))
                       (sqr-all (rest lst)))]))
```

```
(sqr-all (cons 4 (cons -2 (cons 3 empty))))
⇒* (cons 16 (sqr-all (cons -2 (cons 3 empty))))
⇒* (cons 16 (cons 4 (sqr-all (cons 3 empty))))
⇒* (cons 16 (cons 4 (cons 9 (sqr-all empty))))
⇒* (cons 16 (cons 4 (cons 9 empty)))
```

How would we write the contract for `len`?

How would we write the contract for `len`?

`len` consumes a list of elements of any type and produces a number.

We could use the following notation:

```
; len: (Listof Any) -> Number
```

How would we write the contract for `len`?

`len` consumes a list of elements of any type and produces a number.

We could use the following notation:

```
; len: (Listof Any) -> Number
```

In general, for any type `T`, a list of elements of type `T` would have type `(Listof T)`.

How would we write the contract for `len`?

`len` consumes a list of elements of any type and produces a number.

We could use the following notation:

```
; len: (Listof Any) -> Number
```

In general, for any type `T`, a list of elements of type `T` would have type `(Listof T)`.

What is the contract for `sqr-all`?

How would we write the contract for `len`?

`len` consumes a list of elements of any type and produces a number.

We could use the following notation:

```
; len: (Listof Any) -> Number
```

In general, for any type `T`, a list of elements of type `T` would have type `(Listof T)`.

What is the contract for `sqr-all`?

Answer:

```
; sqr-all: (Listof Number) -> (Listof Number)
```

The `pos-elts` function consumes a list of numbers and produces a list of the positive elements of the argument list in the same order.

```
(pos-elts (cons 4 (cons -2 (cons 3 empty))))
⇒* (cons 4 (cons 3 empty))
```

The `pos-elts` function consumes a list of numbers and produces a list of the positive elements of the argument list in the same order.

```
(pos-elts (cons 4 (cons -2 (cons 3 empty))))
⇒* (cons 4 (cons 3 empty))
```

Once again, we use the template to guide development.

```
(define (pos-elts lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (pos-elts (rest lst)) ...]))
```

Developing the pos-elts function

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (first lst) ...
                 ... (pos-elts (rest lst)) ...]))
```

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) ... (first lst) ...
                 ... (pos-elts (rest lst)) ...]))
```

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
      (cond
        [(positive? (first lst))
            ... (pos-elts (rest lst)) ...]
        [else
            ... (pos-elts (rest lst)) ...])]))
```

Developing the pos-elts function

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
       (cond
         [(positive? (first lst))
             ... (pos-elts (rest lst)) ...]
         [else
             ... (pos-elts (rest lst)) ...])]))
```

Developing the pos-elts function

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
       (cond
         [(positive? (first lst))
             ... (pos-elts (rest lst)) ...]
         [else
             ... (pos-elts (rest lst)) ...])]))
```

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
       (cond
         [(positive? (first lst))
            (cons (first lst) (pos-elts (rest lst)))]
         [else
             (pos-elts (rest lst))])]))
```

A condensed trace of the pos-elts function

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
       (cond
         [(positive? (first lst))
            (cons (first lst) (pos-elts (rest lst)))]
         [else
             (pos-elts (rest lst))])]))
```

```
(pos-elts (cons 4 (cons -2 (cons 3 empty))))
⇒* (cons 4 (pos-elts (cons -2 (cons 3 empty))))
⇒* (cons 4 (pos-elts (cons 3 empty)))
⇒* (cons 4 (cons 3 (pos-elts empty)))
⇒* (cons 4 (cons 3 empty))
```

We saw how to add a second parameter to the template for Nats. We can do the same thing for lists.

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                ... (my-list-fn (rest lst1) lst2) ...]))
```

We saw how to add a second parameter to the template for Nats. We can do the same thing for lists.

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                 ... (my-list-fn (rest lst1) lst2) ...]))
```

The first example we will work out is the `app` function, which consumes two lists and produces the list with all of the elements of the first list followed by all of the elements of the second list.

We saw how to add a second parameter to the template for Nats. We can do the same thing for lists.

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                 ... (my-list-fn (rest lst1) lst2) ...]))
```

The first example we will work out is the `app` function, which consumes two lists and produces the list with all of the elements of the first list followed by all of the elements of the second list.

```
(app (cons 3 (cons 5 empty))
     (cons 7 (cons 5 (cons 4 empty))))
⇒* (cons 3 (cons 5 (cons 7 (cons 5 (cons 4 empty)))))
```

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                  ... (my-list-fn (rest lst1) lst2) ...]))
```

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                  ... (my-list-fn (rest lst1) lst2) ...]))


(define (app lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                  ... (app (rest lst1) lst2) ...]))
```

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                  ... (my-list-fn (rest lst1) lst2) ...]))
```

```
(define (app lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1) ... (first lst1) ... lst2 ...
                  ... (app (rest lst1) lst2) ...]))
```

```
(define (app lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cons (first lst1) (app (rest lst1) lst2))]))
```

Racket's built-in `append` function does the same thing.

A condensed trace of app

```
(define (app lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cons (first lst1) (app (rest lst1) lst2))]))
```

```
(app (cons 3 (cons 5 empty)) (cons 7 (cons 5 (cons 4 empty))))
⇒* (cons 3 (app (cons 5 empty)
                          (cons 7 (cons 5 (cons 4 empty)))))
⇒* (cons 3 (cons 5 (app empty
                         (cons 7 (cons 5 (cons 4 empty))))))
⇒* (cons 3 (cons 5 (cons 7 (cons 5 (cons 4 empty)))))
```

Sets resemble sequences, but no repetition of values is allowed, and the order in which values appear is unimportant.

$\{4, 2, 3\}$ is the same set as $\{2, 3, 4\}$. We often write sets with their values in order, but this is just for the convenience of the reader.

Sets resemble sequences, but no repetition of values is allowed, and the order in which values appear is unimportant.

$\{4, 2, 3\}$ is the same set as $\{2, 3, 4\}$. We often write sets with their values in order, but this is just for the convenience of the reader.

Let's briefly review some concepts and notation for sets.

Sets resemble sequences, but no repetition of values is allowed, and the order in which values appear is unimportant.

$\{4, 2, 3\}$ is the same set as $\{2, 3, 4\}$. We often write sets with their values in order, but this is just for the convenience of the reader.

Let's briefly review some concepts and notation for sets.

The statement "4 is an element of $\{2, 3, 4\}$ is written $4 \in \{2, 3, 4\}$."

$5 \notin \{2, 3, 4\}$ means "5 is not an element of $\{2, 3, 4\}$".

Sets resemble sequences, but no repetition of values is allowed, and the order in which values appear is unimportant.

$\{4, 2, 3\}$ is the same set as $\{2, 3, 4\}$. We often write sets with their values in order, but this is just for the convenience of the reader.

Let's briefly review some concepts and notation for sets.

The statement "4 is an element of $\{2, 3, 4\}$ is written $4 \in \{2, 3, 4\}$."

$5 \notin \{2, 3, 4\}$ means "5 is not an element of $\{2, 3, 4\}$".

A set S is a subset of a set T if every element of S is an element of T. We write $S \subseteq T$.

$\{2, 4\} \subseteq \{2, 3, 4\}$

$\{3, 5\} \nsubseteq \{2, 3, 4\}$ means $\{3, 5\}$ is not a subset of $\{2, 3, 4\}$.

Definition

The **union** of two sets S and T, written $S \cup T$, contains all elements that are in either set.

$S \cup T = \{e \mid e \in S \texttt{ or } e \in T\}$

Example: $\{3, 5\} \cup \{2, 5\} = \{2, 3, 5\}$.

Definition

The **intersection** of two sets S and T, written $S \cap T$, contains all elements that are in both sets.

$S \cap T = \{e \mid e \in S \texttt{ and } e \in T\}$

Example: $\{3, 5\} \cap \{2, 5\} = \{5\}$.

Representing sets using lists

We will discuss two different representations of sets.

We will discuss two different representations of sets.

Our first representation will store the elements of the set in a list.

For example, the set $\{2, 3, 4\}$ might be represented by the list `(cons 4 (cons 2 (cons 3 empty)))`.

We will discuss two different representations of sets.

Our first representation will store the elements of the set in a list.

For example, the set $\{2, 3, 4\}$ might be represented by the list `(cons 4 (cons 2 (cons 3 empty)))`.

The representation is not unique. The same set could be represented by the list `(cons 2 (cons 3 (cons 4 empty)))`.

We will discuss two different representations of sets.

Our first representation will store the elements of the set in a list.

For example, the set $\{2, 3, 4\}$ might be represented by the list `(cons 4 (cons 2 (cons 3 empty)))`.

The representation is not unique. The same set could be represented by the list `(cons 2 (cons 3 (cons 4 empty)))`.

We can develop Racket functions implementing the various set predicates and operations we have discussed, using the template for list functions.

The `elem-of` function consumes a value `v` and a list `lst` representing a set, and produces `true` if the value is in the set; otherwise, it produces `false`.

The `elem-of` function consumes a value `v` and a list `lst` representing a set, and produces `true` if the value is in the set; otherwise, it produces `false`.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

Developing the elem-of function (continued)

```
(define (elem-of v lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ... v
                 ... (elem-of v (rest lst)) ...]))

(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst) ... (first lst) ... v
                 ... (elem-of v (rest lst)) ...]))
```

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst) ... (first lst) ... v
                 ... (elem-of v (rest lst)) ...]))

(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (cond
         [(equal? (first lst) v) ...]
         [else ... (elem-of v (rest lst)) ...])]))
```

Developing the elem-of function (continued)

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (cond
         [(equal? (first lst) v) ...]
         [else ... (elem-of v (rest lst)) ...])]))

(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (cond
         [(equal? (first lst) v) true]
         [else (elem-of v (rest lst))])]))
```

Simplifying the elem-of function

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (cond
         [(equal? (first lst) v) true]
         [else (elem-of v (rest lst))])]))

(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (or (equal? (first lst) v)
           (elem-of v (rest lst)))]))
```

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (or (equal? (first lst) v)
           (elem-of v (rest lst)))]))

(define (elem-of v lst)
  (and (not (empty? lst))
       (or
         (equal? (first lst) v)
         (elem-of v (rest lst)))))
```

The subset function consumes two lists lst1 and lst2 representing sets, and produces true if and only if the first set is a subset of the second set.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

```
(define (subset lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ..]
    [(cons? lst1) ... (first lst1) ... lst2
                  ... (subset (rest lst1) lst2) ...]))
```

```
(define (subset lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ..]
    [(cons? lst1) ... (first lst1) ... lst2
                  ... (subset (rest lst1) lst2) ...]))
```

```
(define (subset lst1 lst2)
  (cond
    [(empty? lst1) true]
    [(cons? lst1) ... (first lst1) ... lst2
                  ... (subset (rest lst1) lst2) ...]))
```

```
(define (subset lst1 lst2)
  (cond
    [(empty? lst1) true]
    [(cons? lst1) ... (first lst1) ... lst2
                  ... (subset (rest lst1) lst2) ...]))
```

```
(define (subset lst1 lst2)
  (cond
    [(empty? lst1) true]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2)
            (subset (rest lst1) lst2)]
         [else false])]))
```

```
(define (subset lst1 lst2)
  (cond
    [(empty? lst1) true]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2)
             (subset (rest lst1) lst2)]
         [else false])]))

(define (subset lst1 lst2)
  (cond
    [(empty? lst1) true]
    [(elem-of (first lst1) lst2)
       (subset (rest lst1) lst2)]
    [else false]))
```

The `union` function consumes two lists `lst1` and `lst2` representing sets, and produces a list representing the union of the two sets.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ..]
    [(cons? lst1) ... (first lst1) ... lst2
                  ... (union (rest lst1) lst2) ...]))
```

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ... lst2
                  ... (union (rest lst1) lst2) ...]))
```

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2)
                   (union (rest lst1) lst2)]
         [else ...])]))
```

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2)
                   (union (rest lst1) lst2)]
         [else ...])]))

(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2)
                   (union (rest lst1) lst2)]
         [else (cons (first lst1)
                     (union (rest lst1) lst2))])]))
```

The implementation we have developed seems inefficient. Consider the computation of the `elem-of` function when the value does not appear in the set.

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (or (equal? (first lst) v)
           (elem-of v (rest lst)))]))
```

```
(elem-of 4 (list 1 2 3)) ⇒* false
```

The implementation we have developed seems inefficient. Consider the computation of the `elem-of` function when the value does not appear in the set.

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (or (equal? (first lst) v)
           (elem-of v (rest lst)))]))
```

```
(elem-of 4 (list 1 2 3)) ⇒* false
```

The number of recursive applications of `elem-of` is equal to the size of the set.

The implementation we have developed seems inefficient. Consider the computation of the `elem-of` function when the value does not appear in the set.

```
(define (elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (or (equal? (first lst) v)
           (elem-of v (rest lst)))]))
(elem-of 4 (list 1 2 3)) ⇒* false
```

The number of recursive applications of `elem-of` is equal to the size of the set.

Now consider the computation of the `union` function when the two sets do not share any values.

The efficiency of the union function

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2) (union (rest lst1) lst2)]
         [else
           (cons (first lst1) (union (rest lst1) lst2))])]))
```

```
(union (list 1 2 3) (list 4 5 6 7))
```

$\Rightarrow^*$ `(list 1 2 3 4 5 6 7)`

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2) (union (rest lst1) lst2)]
         [else
           (cons (first lst1) (union (rest lst1) lst2))])]))
```

```
(union (list 1 2 3) (list 4 5 6 7))
                  ⇒* (list 1 2 3 4 5 6 7)
```

The number of recursive applications of `union` is the size of the set represented by `lst1`.

The efficiency of the union function

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2) (union (rest lst1) lst2)]
         [else
           (cons (first lst1) (union (rest lst1) lst2))])]))
```

```
(union (list 1 2 3) (list 4 5 6 7))
                    ⇒* (list 1 2 3 4 5 6 7)
```

The number of recursive applications of `union` is the size of the set represented by `lst1`.

Each application of `elem-of` in `union` results in a number of recursive applications of `elem-of` that is the size of the set represented by `lst2`.

```
(define (union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1)
       (cond
         [(elem-of (first lst1) lst2) (union (rest lst1) lst2)]
         [else
           (cons (first lst1) (union (rest lst1) lst2))])]))
(union (list 1 2 3) (list 4 5 6 7))
                  ⇒* (list 1 2 3 4 5 6 7)
```

The number of recursive applications of `union` is the size of the set represented by `lst1`.

Each application of `elem-of` in `union` results in a number of recursive applications of `elem-of` that is the size of the set represented by `lst2`.

The work done is at least the product of the sizes of the two sets.
Can we do better?

It is more natural for humans to see $\{2, 3, 4\}$ instead of $\{4, 2, 3\}$.

Suppose we decide that the set $\{4, 2, 3\}$ must be represented by `(list 2 3 4)` and not `(list 4 2 3)`.

Another implementation: using ordered lists

It is more natural for humans to see $\{2, 3, 4\}$ instead of $\{4, 2, 3\}$.

Suppose we decide that the set $\{4, 2, 3\}$ must be represented by `(list 2 3 4)` and not `(list 4 2 3)`.

It doesn't matter to the computer, but can this representation be more efficient? We start, as before, with the `elem-of` function.

To avoid confusion, we will prefix the names of functions using the new representation with `o-`, as in `o-elem-of`.

It is more natural for humans to see $\{2, 3, 4\}$ instead of $\{4, 2, 3\}$.

Suppose we decide that the set $\{4, 2, 3\}$ must be represented by `(list 2 3 4)` and not `(list 4 2 3)`.

It doesn't matter to the computer, but can this representation be more efficient? We start, as before, with the `elem-of` function.

To avoid confusion, we will prefix the names of functions using the new representation with `o-`, as in `o-elem-of`.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))
```

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))

(define (o-elem-of v lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ... v ...
                 ... (o-elem-of v (rest lst)) ...]))
```

Developing the elem-of function for ordered lists

```
(define (o-elem-of v lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ... v ...
                 ... (o-elem-of v (rest lst)) ...]))

(define (o-elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst) ... (first lst) ... v ...
                 ... (o-elem-of v (rest lst)) ...]))
```

Developing the elem-of function for ordered lists

```
(define (o-elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst) ... (first lst) ... v ...
                 ... (o-elem-of v (rest lst)) ...]))
```

```
(define (o-elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst) ... (first lst) ... v ...
                 ... (o-elem-of v (rest lst)) ...]))
```

```
(define (o-elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
       (cond
         [(< (first lst) v) ...
          (o-elem-of v (rest lst)) ...]
         [(= (first lst) v) ...
          (o-elem-of v (rest lst)) ...]
         [(> (first lst) v) ...
          (o-elem-of v (rest lst)) ...])]))
```

```
(define (o-elem-of v lst)
  (cond
    [(empty? lst) false]
    [(cons? lst)
      (cond
        [(< (first lst) v) (o-elem-of v (rest lst))]
        [(= (first lst) v) true]
        [(> (first lst) v) false])]))
```

Are ordered lists a more efficient set representation?

Consider the computations

```
(elem-of 1 (list 2 3 4 5 6 7 8 9))
(o-elem-of 1 (list 2 3 4 5 6 7 8 9))
```

The first one recursively applies `elem-of` nine times, but the second one does no recursive applications.

Are ordered lists a more efficient set representation?

Consider the computations

```
(elem-of 1 (list 2 3 4 5 6 7 8 9))
(o-elem-of 1 (list 2 3 4 5 6 7 8 9))
```

The first one recursively applies `elem-of` nine times, but the second one does no recursive applications.

On the other hand, we can describe two other computations such that each function does nine recursive applications.

Consider the computations

```
(elem-of 1 (list 2 3 4 5 6 7 8 9))
(o-elem-of 1 (list 2 3 4 5 6 7 8 9))
```

The first one recursively applies `elem-of` nine times, but the second one does no recursive applications.

On the other hand, we can describe two other computations such that each function does nine recursive applications.

```
(elem-of 10 (list 2 3 4 5 6 7 8 9))
(o-elem-of 10 (list 2 3 4 5 6 7 8 9))
```

Consider the computations

```
(elem-of 1 (list 2 3 4 5 6 7 8 9))
(o-elem-of 1 (list 2 3 4 5 6 7 8 9))
```

The first one recursively applies `elem-of` nine times, but the second one does no recursive applications.

On the other hand, we can describe two other computations such that each function does nine recursive applications.

```
(elem-of 10 (list 2 3 4 5 6 7 8 9))
(o-elem-of 10 (list 2 3 4 5 6 7 8 9))
```

So there is a possible saving, but it is not guaranteed. However, for some of the other operations, we are guaranteed to do much better.

```
(define (my-list-fn lst)
  (cond
    [(empty? lst) ...]
    [(cons? lst) ... (first lst) ...
                 ... (my-list-fn (rest lst)) ...]))

(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ...  lst2 ...
                  ... (o-union (rest lst1) lst2) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(cons? lst1) ... (first lst1) ...  lst2 ...
                  ... (o-union (rest lst1) lst2) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1) ... (first lst1)
                  ... (first lst2) ... (rest lst2) ...
                  ... (o-union (rest lst1) lst2) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(cons? lst1) ... (first lst1)
                  ... (first lst2) ... (rest lst2) ...
                  ... (o-union (rest lst1) lst2) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1) ... (first lst1)
                  ... (first lst2) ... (rest lst2) ...
                  ... (o-union (rest lst1) lst2) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1) ... (first lst1)
                  ... (first lst2) ... (rest lst2) ...
                  ... (o-union (rest lst1) lst2) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1) ... (first lst1) ...  (first lst2) ...
                  ... (o-union (rest lst1) lst2) ...
                  ... (o-union lst1 (rest lst2)) ...
                  ... (o-union (rest lst1) (rest lst2)) ...]))
```

Developing the union function for ordered lists

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1) ... (first lst1) ... (first lst2) ...
                  ... (o-union (rest lst1) lst2) ...
                  ... (o-union lst1 (rest lst2)) ...
                  ... (o-union (rest lst1) (rest lst2)) ...]))
```

Developing the union function for ordered lists

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1) ... (first lst1) ... (first lst2) ...
                  ... (o-union (rest lst1) lst2) ...
                  ... (o-union lst1 (rest lst2)) ...
                  ... (o-union (rest lst1) (rest lst2)) ...]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1)
       (cond
         [(< (first lst1) (first lst2))
            ... (o-union (rest lst1) lst2) ...
            ... (o-union lst1 (rest lst2)) ...
            ... (o-union (rest lst1) (rest lst2)) ...]
         [(= (first lst1) (first lst2)) ...]
         [(> (first lst1) (first lst2)) ...])]))
```

Developing the union function for ordered lists

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1)
       (cond
         [(< (first lst1) (first lst2))
            ... (o-union (rest lst1) lst2) ...
            ... (o-union lst1 (rest lst2)) ...
            ... (o-union (rest lst1) (rest lst2)) ...]
         [(= (first lst1) (first lst2)) ...]
         [(> (first lst1) (first lst2)) ...])]))
```

Developing the union function for ordered lists

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1)
       (cond
         [(< (first lst1) (first lst2))
            ... (o-union (rest lst1) lst2) ...
            ... (o-union lst1 (rest lst2)) ...
            ... (o-union (rest lst1) (rest lst2)) ...]
         [(= (first lst1) (first lst2)) ...]
         [(> (first lst1) (first lst2)) ...])]))
```

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1)
       (cond
         [(< (first lst1) (first lst2))
            (cons (first lst1) (o-union (rest lst1) lst2))]
         [(= (first lst1) (first lst2))
            (cons (first lst1) (o-union (rest lst1) (rest lst2)))]
         [(> (first lst1) (first lst2))
            (cons (first lst2) (o-union lst1 (rest lst2)))])]))
```

Developing the union function for ordered lists

```
(define (o-union lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(cons? lst1)
       (cond
         [(< (first lst1) (first lst2))
            (cons (first lst1) (o-union (rest lst1) lst2))]
         [(= (first lst1) (first lst2))
            (cons (first lst1) (o-union (rest lst1) (rest lst2)))]
         [(> (first lst1) (first lst2))
            (cons (first lst2) (o-union lst1 (rest lst2)))])]))
```

In each recursive application, at least one of the lists is smaller.

The total number of recursive applications is bounded above by the sum of the sizes of the sets.

Using ordered lists to represent sets is more efficient.

The template for structural recursion on two lists

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(empty? lst2) ... lst1 ...]
    [(cons? lst1)
        ... (first lst1) ... (first lst2) ...
        ... (my-list-fn (rest lst1) lst2) ...
        ... (my-list-fn lst1 (rest lst2)) ...
        ... (my-list-fn (rest lst1) (rest lst2)) ...]))
```

The template for structural recursion on two lists

```
(define (my-list-fn lst1 lst2)
  (cond
    [(empty? lst1) ... lst2 ...]
    [(empty? lst2) ... lst1 ...]
    [(cons? lst1)
         ... (first lst1) ... (first lst2) ...
         ... (my-list-fn (rest lst1) lst2) ...
         ... (my-list-fn lst1 (rest lst2)) ...
         ... (my-list-fn (rest lst1) (rest lst2)) ...]))
```

We can develop similar templates for functions that do structural recursion on a list and a natural number, or on two natural numbers.

What if we want to represent sets of points?

What if we want to represent sets of points?

We need to define an ordering on points in order to use ordered lists.

Is $(3, 4) < (1, 6)$?

What if we want to represent sets of points?

We need to define an ordering on points in order to use ordered lists.

Is $(3, 4) < (1, 6)$?

The ordering doesn't have to make sense.

What if we want to represent sets of points?

We need to define an ordering on points in order to use ordered lists.

Is $(3, 4) < (1, 6)$?

The ordering doesn't have to make sense.

Lexicographic order:
$(x_1, y_1) < (x_2, y_2)$ if and only if $x_1 < x_2$, or $x_1 = x_2$ and $y_1 < y_2$.

What if we want to represent sets of points?

We need to define an ordering on points in order to use ordered lists.

Is $(3,4) < (1,6)$?

The ordering doesn't have to make sense.

Lexicographic order:
$(x_1, y_1) < (x_2, y_2)$ if and only if $x_1 < x_2$, or $x_1 = x_2$ and $y_1 < y_2$.

We can extend this idea to other structures, lists, and mixtures of types.

The next language level, Beginning Student With List Abbreviations, provides concise alternatives for working with lists.

The next language level, Beginning Student With List Abbreviations, provides concise alternatives for working with lists.

Beginning Student already provides `list`.

`(cons 1 (cons 2 (cons 3 empty)))` is abbreviated by `(list 1 2 3)`.

The next language level, Beginning Student With List Abbreviations, provides concise alternatives for working with lists.

Beginning Student already provides `list`.

`(cons 1 (cons 2 (cons 3 empty)))` is abbreviated by `(list 1 2 3)`.

Note that `cons` and `list` have different results and different purposes.

The next language level, Beginning Student With List Abbreviations, provides concise alternatives for working with lists.

Beginning Student already provides `list`.

`(cons 1 (cons 2 (cons 3 empty)))` is abbreviated by `(list 1 2 3)`.

Note that `cons` and `list` have different results and different purposes.

We use `list` to construct a list of fixed size (whose length is known when we write the program).

The next language level, Beginning Student With List Abbreviations, provides concise alternatives for working with lists.

Beginning Student already provides `list`.

`(cons 1 (cons 2 (cons 3 empty)))` is abbreviated by `(list 1 2 3)`.

Note that `cons` and `list` have different results and different purposes.

We use `list` to construct a list of fixed size (whose length is known when we write the program).

We use `cons` to construct a list from one new element (the first) and a list of arbitrary size (whose length is known only when the second argument to `cons` is evaluated during the running of the program).

Beginning Student With List Abbreviations also provides some shortcuts for accessing specific elements of lists.

`(second my-list)` is an abbreviation for `(first (rest my-list))`.

`third`, `fourth`, and so on up to `eighth` are also defined.

Use these **sparingly** to improve readability.

Quoting lists

Putting a quote in front of something that looks like an identifier makes it into a symbol: `red` becomes `'red`.

Numbers quote to themselves: `'4` is just `4`.

We've already seen `'()`, the value of the empty list.

Putting a quote in front of several items enclosed in parentheses makes a list that contains each item quoted.

`'(red 4 blue)` is the same as `(list 'red 4 'blue)`.

Lists containing lists

These three two-element lists are all the same.
Each element is itself a two-element list.

```
(cons (cons 1 (cons 2 empty))
      (cons (cons 3 (cons 4 empty))
            empty))
```

```
(list (list 1 2) (list 3 4))
```

```
'((1 2) (3 4))
```

Quote notation really starts to pay off when dealing with lists containing lists.

Exercise

What is `(first '((1 2) (3 4)))`?

Exercise

What is `(first '((1 2) (3 4)))`?

Exercise Solution

It is `'(1 2)`, or `'(list 1 2)`, or `(cons 1 (cons 2 empty))`.

Exercise

What is `(first '((1 2) (3 4)))`?

Exercise Solution

It is `'(1 2)`, or `'(list 1 2)`, or `(cons 1 (cons 2 empty))`.

Exercise

What is `(rest '((1 2) (3 4)))`?

Exercise

What is `(first '((1 2) (3 4)))`?

Exercise Solution

It is `'(1 2)`, or `'(list 1 2)`, or `(cons 1 (cons 2 empty))`.

Exercise

What is `(rest '((1 2) (3 4)))`?

Exercise Solution

It is `'((3 4))`, or `(list (list 3 4))`, or
`(cons (cons 3 (cons 4 empty)) empty)`.

Exercise

What is the difference, if any,
between `'(1 (+ 1 2) 5)` and `(list 1 (+ 1 2) 5)`?

Exercise

What is the difference, if any,
between `'(1 (+ 1 2) 5)` and `(list 1 (+ 1 2) 5)`?

Exercise Solution

`'(1 (+ 1 2) 5)` is `(list 1 (list '+ 1 2) 5)`.

`(list 1 (+ 1 2) 5)` is `(list 1 3 5)`.

1. You should be able to write functions that consume and produce lists.

1. You should be able to write functions that consume and produce lists.
2. You should be able to use the template for functions that do structural recursion on one list (possibly with additional parameters) and the template for functions that do structural recursion on two lists.

1. You should be able to write functions that consume and produce lists.
2. You should be able to use the template for functions that do structural recursion on one list (possibly with additional parameters) and the template for functions that do structural recursion on two lists.
3. You should be able to create different Racket data representations for the same mathematical concept, and to informally discuss their relative efficiencies.

1. You should be able to write functions that consume and produce lists.
2. You should be able to use the template for functions that do structural recursion on one list (possibly with additional parameters) and the template for functions that do structural recursion on two lists.
3. You should be able to create different Racket data representations for the same mathematical concept, and to informally discuss their relative efficiencies.
4. You should be able to use list abbreviations in your programs, and to understand list values expressed using quote notation.