Module 6: Working with recursion

Readings: HtDP, sections 11, 12, 13 (Intermezzo 2)

We can extend the idea of a self-referential definition to defining the natural numbers, which leads to the use of recursion in order to write functions that consume numbers.

Natural numbers

A **natural number** is either

- 0 or
- 1 plus a *natural number*.

The analogy to the self-referential definition of lists can be made clearer by defining a "1 plus" function:

```
(define (add1 n) (+ 1 n))
(add1 0) ⇒ 1
(add1 (add1 0)) ⇒ 2
(add1 (add1 (add1 0))) ⇒ 3
```

For lists:

- empty is the base case,
- empty? distinguishes between the base case and the recursive case,
- cons makes a list that is one step farther from the base case,
- rest makes the list formed by “undoing” the cons, and
- first is what you take away from a list to get the rest.

For natural numbers:

- 0 is the base case,
- zero? distinguishes between the base case and the recursive case,
- add1 makes a number that is one step farther from the base case, and
- sub1 makes the number formed by "undoing" the add1 (for (define (sub1 n) (− n 1))).
- What plays the role of first?

```
;; A list is either
;;      empty or
;;      (cons f r), where f is a value and r is a list.
```

To find a template, we used a cond for the two cases.

We broke up the nonempty list using

- the selector first to extract f,
- the selector rest to extract r, and
- an application of the function on r.

```
;; A natural number is either
;;      0 or
;;      (add1 k), where k is a natural number.
```

To find a template for a natural number n, we will use a cond for the two cases.

We will break up the non-zero case using

- the function sub1 to extract k and
- an application of the function on k (that is, (sub1 n)).

Comparing the templates

```
;; The list template
(define (my-list-fun alist)
  (cond
    [(empty? alist) ...]
    [else ... (first alist) ... (my-list-fun (rest alist)) ...]))
;; The natural number template
(define (my-nat-fun n)
  (cond
    [(zero? n) ...]
    [else ... (my-nat-fun (sub1 n)) ...]))
```

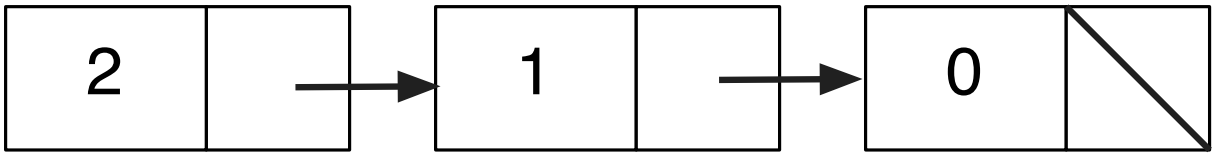
Example: producing a decreasing list

countdown consumes a natural number n and produces a decreasing list of all natural numbers less than or equal to n.

Use the data definition to derive examples.

(countdown 0) $\Rightarrow$ (cons 0 empty)

(countdown 2) $\Rightarrow$ (cons 2 (cons 1 (cons 0 empty)))



Developing countdown

Using the natural number template:

```
(define (countdown n)
  (cond
    [(zero? n) ...]
    [else ... (countdown (sub1 n)) ...]))
```

If n is 0, we produce the list containing 0.

If n is nonzero, we cons n onto the countdown list for n-1.

```
;; countdown: nat → (listof nat)
;; Produces a decreasing list of nats starting at n.
;; Examples: (countdown 0) ⇒ (cons 0 empty)
;; (countdown 2) ⇒ (cons 2 (cons 1 (cons 0 empty)))
(define (countdown n)
  (cond
    [(zero? n) (cons 0 empty)]
    [else (cons n (countdown (sub1 n)))]))
```

Condensed trace of countdown

(countdown 2)

$\Rightarrow$ (cons 2 (countdown (sub1 2)))

$\Rightarrow$ (cons 2 (countdown 1))

$\Rightarrow$ (cons 2 (cons 1 (countdown (sub1 1))))

$\Rightarrow$ (cons 2 (cons 1 (countdown 0)))

$\Rightarrow$ (cons 2 (cons 1 (cons 0 empty)))

If the function countdown is applied to a negative argument, it will not terminate.

The following variation is a little more robust.

It can handle negative arguments more gracefully.

```
(define (countdown n)
  (cond
    [(<= n 0) (cons 0 empty)]
    [else (cons n (countdown (sub1 n)))]))
```

Subintervals of the natural numbers

If we change the base case test from (zero? n) to (= n 7), we can stop the countdown at 7.

This corresponds to the following data definition:

A **natural number greater than or equal to 7** is either

- 7 or
- one plus a *natural number greater than or equal to 7*.

Recall the notation [. . .] to denote a subset in a contract, such as nat[>= 7].

Template for a natural number greater than or equal to 7

```
;; my-downto-7-fun: nat[>=7] → any
(define (my-downto-7-fun n)
  (cond
    [(<= n 7) ...]
    [else ... (my-downto-7-fun (sub1 n))]))
```

```
;; countdown-to-7: nat[>= 7] → (listof nat[>= 7])
;; Produces a decreasing list of nats starting with n
;; and ending with 7.
;; Examples: (countdown-to-7 7) ⇒ (cons 7 empty)
;; (countdown-to-7 9) ⇒ (cons 9 (cons 8 (cons 7 empty)))
(define (countdown-to-7 n)
  (cond
    [(<= n 7) (cons 7 empty)]
    [else (cons n (countdown-to-7 (sub1 n)))]))
```

Again, making the base case be (<= n 7) is more robust.

We can generalize both countdown and countdown-to-7 by providing the base value (e.g. 0 or 7) as a parameter b.

This corresponds to the following definition:

An **integer greater than or equal to** b is either

- b or
- 1 plus *an integer greater than or equal to* b.

The parameter b (for "base") has to "go along for the ride" in the recursion.

Template for an integer greater than or equal to a base

```
;; my-downto-fun: int int → any
;; n is greater than or equal to b
(define (my-downto-fun n b)
  (cond
    [(= n b) ... b ...]
    [else ... (my-downto-fun (sub1 n) b)]))
```

The template my-nat-fun is a special case where b is zero.

Since we know the value zero, it doesn't need to be a parameter.

The function countdown-to

```
;; countdown-to: int int → (listof int)[nonempty]
;; Produces a decreasing list of ints starting with n
;; and ending with b; n is greater than or equal to b.
;; Examples: (countdown-to 3 3) ⇒ (cons 3 empty)
;; (countdown-to 4 2) ⇒ (cons 4 (cons 3 (cons 2 empty)))
(define (countdown-to n b)
  (cond
    [(= n b) (cons b empty)]
    [else (cons n (countdown-to (sub1 n) b))]))
```

Condensed trace of countdown-to

(countdown-to 4 2)

⇒ (cons 4 (countdown-to (sub1 4) 2))

⇒ (cons 4 (countdown-to 3 2))

⇒ (cons 4 (cons 3 (countdown-to (sub1 3) 2)))

⇒ (cons 4 (cons 3 (countdown-to 2 2)))

⇒ (cons 4 (cons 3 (cons 2 empty)))

What is the result of (countdown-to 1 −2)?

Of (countdown-to −4 −2)?

Some useful notation

The symbol $\mathbb{Z}$ is often used to denote the integers.

We can add subscripts to define subsets of the integers.

For example, $\mathbb{Z}_{\geq 0}$ defines the non-negative integers, also known as the natural numbers.

$\mathbb{Z}_{\geq b}$ defines the integers greater than or equal to b.

Our previous definition:

An **integer greater than or equal to** b is either

- b or
- 1 plus *an integer greater than or equal to* b.

Rewritten:

An **integer in** $\mathbb{Z}_{\geq b}$ is either

- b or
- $1 + k$, where k is an *integer in* $\mathbb{Z}_{\geq b}$.

Going the other way

What if we want an increasing count?

Consider the non-positive integers $\mathbb{Z}_{\leq 0}$.

An **integer in** $\mathbb{Z}_{\leq 0}$ is either

- 0 or
- $k - 1$, where k is an *integer in* $\mathbb{Z}_{\leq 0}$.

Examples: -1 is $(0 - 1)$, -2 is $((-1) - 1)$.

```
(define (my-nonpos-fun n)
  (cond
    [(zero? n) ...]
    [else ... (my-nonpos-fun (add1 n)) ...]))
```

We can use this to develop a function to produce lists such as (cons −2 (cons −1 (cons 0 empty))).

```
;; countup: int[≤ 0] → (listof int[≤ 0])
;; Produces an increasing list of ints from n up to zero.
;; Examples: (countup 0) ⇒ (cons 0 empty)
;; (countup -2) ⇒ (cons -2 (cons -1 (cons 0 empty)))
(define (countup n)
  (cond
    [(zero? n) (cons 0 empty)]
    [else (cons n (countup (add1 n)))]))
```

As before, we can generalize this to counting up to b.

An **integer in** $\mathbb{Z}_{\leq b}$ is either

- b or
- $k - 1$, where k is an *integer in* $\mathbb{Z}_{\leq b}$.

For $b = 14$, 14 is an integer in $\mathbb{Z}_{\leq 14}$.

13 is an integer in $\mathbb{Z}_{\leq 14}$ because it is $14 - 1$.

12 is because it is $13 - 1$, and so on.

In other words, 12 is (sub1 (sub1 14)).

An integer n in $\mathbb{Z}_{\leq b}$ is either

- b or
- of the form $k - 1$ for k in $\mathbb{Z}_{\leq b}$.

To extract k from $n = k - 1$, we use (add1 n).

```
;; my-upto-fun: int int → any
;; n is less than or equal to b.
(define (my-upto-fun n b)
  (cond
    [(= n b) ... b ...]
    [else ... (my-upto-fun (add1 n) b)]))
```

```
;; countup-to: int int → (listof int)
;; Produces an increasing list of ints starting with n
;; and ending with b; n is less than or equal to b.
;; Examples: (countup-to 5 5) ⇒ (cons 5 empty)
;; (countup-to 6 8) ⇒ (cons 6 (cons 7 (cons 8 empty)))
(define (countup-to n b)
  (cond
    [(= n b) (cons b empty)]
    [else (cons n (countup-to (add1 n) b))]))
```

Condensed trace of countup-to

(countup-to 6 8)

⇒ (cons 6 (countup-to (add1 6) 8))

⇒ (cons 6 (countup-to 7 8))

⇒ (cons 6 (cons 7 (countup-to (add1 7) 8)))

⇒ (cons 6 (cons 7 (countup-to 8 8)))

⇒ (cons 6 (cons 7 (cons 8 empty)))

Many imperative programming languages offer several language constructs to do repetition:

for i = 1 to 10 do { ... }

Scheme offers one construct – recursion – that is flexible enough to handle these situations and more.

We will soon see how to use Scheme's abstraction capabilities to handle many common uses of recursion.

When you are learning to use recursion with integers, sometimes you will “get it backwards” and use the countdown pattern when you should be using the countup pattern, or vice-versa.

Avoid using the built-in list function reverse to fix your error. It cannot always save a computation done in the wrong order.

Do not use reverse in your labs, assignments, or exams. You will lose many marks.

Instead, learn to fix your mistake by starting with the right template.

Example: factorial

Suppose we wish to compute $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$, or (factorial n), for $n \geq 0$ ($0!$ is defined to be 1).

To choose between the templates, we consider what information each gives us.

Counting down: (factorial (sub1 n))

Counting up: (factorial (add1 n))

Filling in the template

```
(define (my-downto-fun n b)
  (cond
    [(= n b) ... b ...]
    [else ... (my-downto-fun (sub1 n) b)]))

(define (factorial n)
  (cond
    [(zero? n) 1]
    [else (* n (factorial (sub1 n)))]))
```

Example: largest square

Suppose we wish to find the largest square less than or equal to n.

How do we check if a particular number is a square?

What range of numbers should we check?

In what order should we check them?

A number is a square if its square root is an integer.

```
;; is-square?: nat[>0] → boolean
;; Produces true if n is a square and false otherwise.
;; Examples: (is-square? 9) ⇒ true
;; (is-square? 10) ⇒ false
(define (is-square? n)
   (integer? (sqrt n)))
;; Tests for is-square?
(check-expect (is-square? 9) true)
(check-expect (is-square? 10) false)
```

Example: marble multiples

Suppose we wish to find the smallest number of marbles needed so that they can be divided evenly by a group of size n1, n2, or n3.

A number is a *good* number if it is divisible by n1, n2, and n3.

How do we check if a particular number is good?

What range of numbers should we check?

In what order should we check them?

Helper function: multiple? determines if a number is a multiple of a factor.

```
(define (multiple? number factor)
  (zero? (remainder number factor)))
```

Helper function: is-good? determines if a number is good.

```
(define (is-good? m n1 n2 n3)
  (and (multiple? m n1) (multiple? m n2) (multiple? m n3)))
```

The smallest number that could be good is the maximum of n1, n2, and n3, for example 12, 3, and 4.

The product of n1, n2, and n3 will be good, so we don't need to check any higher numbers.

Should we count up or count down to find the smallest number with a property?

```
;; range-marbles: nat[>0] nat[>0] nat[>0] nat[>0] → nat[>0]
;; Produces the smallest number that is divided by all of n1, n2,
;; and n3 in the range from m to the product of n1, n2, and n3.
;; Examples: (range-marbles 1 1 1 1) ⇒ 1
;; (range-marbles 12 6 3 2) ⇒ 12
;; (range-marbles 10 6 3 2) ⇒ 12
(define (range-marbles m n1 n2 n3)
  (cond
    [(is-good? m n1 n2 n3) m]
    [else (range-marbles (add1 m) n1 n2 n3)]))
```

We now have four parameters instead of three.

How do we change the number of parameters back to three?

```
;; fewest-marbles: nat[>0] nat[>0] nat[>0] → nat[>0]
;; Produces the smallest number that is divided by all of
;; n1, n2, and n3.
;; Examples: (fewest-marbles 1 1 1) ⇒ 1
;; (fewest-marbles 6 3 2) ⇒ 6
(define (fewest-marbles n1 n2 n3)
  (range-marbles (max n1 n2 n3) n1 n2 n3))
```

Even numbers

An **even natural number** is either

- 0 or
- 2 plus an *even natural number*.

```
(define (my-even-fun n)
  (cond
    [(zero? n) ...]
    [else ... (my-even-fun (- n 2)) ...]))
```

Integer powers of ten

An **integer power of ten** is either

- 1 or
- 10 times an *integer power of ten*.

```
(define (my-power-ten-fun n)
  (cond
    [(= 1 n) ...]
    [else ... (my-power-ten-fun (/ n 10)) ...]))
```

More complicated situations

Sometimes a recursive function will use a helper function that itself is recursive.

Sorting a list of numbers provides a good example.

In CS 115 and CS 116, we will see several different sorting algorithms.

We will sort from highest number to lowest.

(cons 9 (cons 5 (cons 3 empty)))

A list is sorted if no number is followed by a larger number.

Filling in the list template

```
;; sort: (listof num) → (listof num)
;; Produces a list like alon sorted in descending order.
(define (sort alon)
  (cond
    [(empty? alon) ...]
    [ else ... (first alon) ... (sort (rest alon)) ...]))
```

If the list alon is empty, so is the result.

Otherwise, the template suggests doing something with the first element of the list, and the sorted version of the rest.

```
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [ else (insert (first alon) (sort (rest alon)))]))
```

insert is a recursive auxiliary function which consumes a number and a sorted list, and adds the number to the sorted list.

Tracing sort

(sort (cons 2 (cons 4 (cons 3 empty))))

$\Rightarrow$ (insert 2 (sort (cons 4 (cons 3 empty))))

$\Rightarrow$ (insert 2 (insert 4 (sort (cons 3 empty))))

$\Rightarrow$ (insert 2 (insert 4 (insert 3 (sort empty))))

$\Rightarrow$ (insert 2 (insert 4 (insert 3 empty)))

$\Rightarrow$ (insert 2 (insert 4 (cons 3 empty)))

$\Rightarrow$ (insert 2 (cons 4 (cons 3 empty)))

$\Rightarrow$ (cons 4 (cons 3 (cons 2 empty)))

The auxiliary function insert

We again use the list template for insert.

```
;; insert: num (listof num) → (listof num)
;; Produces the sorted (in decreasing order) list formed by
;; adding the number n to the sorted list alon.
(define (insert n alon)
  (cond
    [(empty? alon) ...]
    [ else ... (first alon) ... (insert n (rest alon)) ...]))
```

```
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [ else (cond
             [(>= n (first alon)) (cons n alon)]
             [ else (cons (first alon) (insert n (rest alon)))])]))
```

Tracing insert

(insert 3 (cons 5 (cons 4 (cons 2 empty))))

$\Rightarrow$ (cons 5 (insert 3 (cons 4 (cons 2 empty))))

$\Rightarrow$ (cons 5 (cons 4 (insert 3 (cons 2 empty))))

$\Rightarrow$ (cons 5 (cons 4 (cons 3 (cons 2 empty))))

This is known as **insertion sort**.

List abbreviations

Now that we understand lists, we can abbreviate them.

The expression

(cons exp1 (cons exp2 (. . . (cons expn empty). . .)))

can be abbreviated as

(list exp1 exp2 . . . expn)

The result of the trace we did on the last slide can be expressed as (list 5 4 3 2).

Beginning Student Scheme with List Abbrevations 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, and use list to construct long lists.

There will still remain situations when using cons is the best choice.

Quoting lists

If the expressions consist of just values, the list abbreviation can be further abbreviated using the quote notation we used for symbols.

(cons 'red (cons 'blue (cons 'green empty))) can be written '(red blue green).

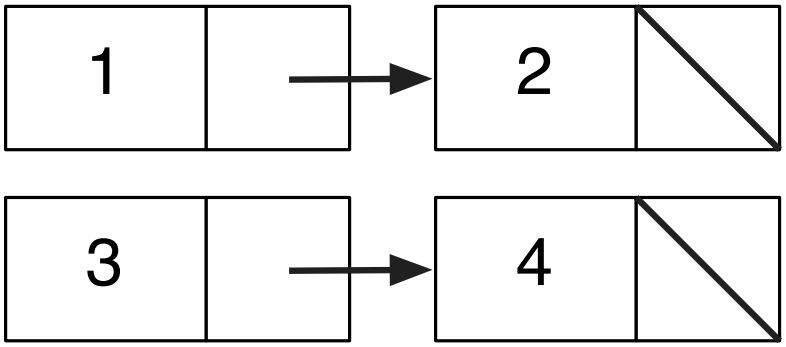
(list 5 4 3 2) can be written '(5 4 3 2), because quoted numbers evaluate to numbers; that is, '1 is the same as 1.

What is '()?

Lists containing lists

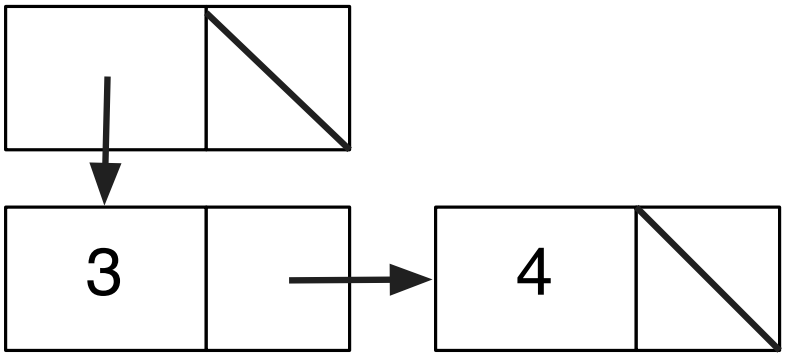
Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

Here are two different two-element lists.




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

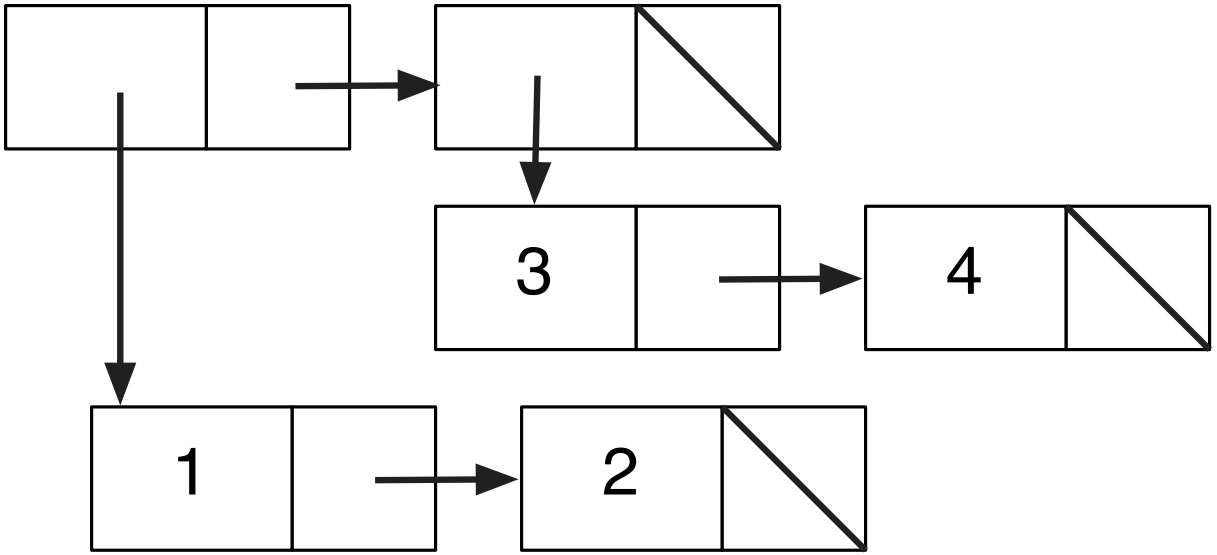
Here is a one-element list whose single element is one of the two-element lists we saw above.




```
(cons (cons 3 (cons 4 empty))
      empty)
```

We can create a two-element list by consing the other list onto this one-element list.

We can create a two-element list, each of whose elements is itself a two-element list.




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

Expressing the list

We have several ways of expressing this list in Scheme:

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

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

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

The abbreviations are generally more expressive.

Lists versus structures

Since lists can contain lists, they offer an alternative to using structures.

For example, we defined a list of students as a list of four-field structures.

We could have instead defined an **slist** as a list of four-element lists.

```
(list (list "Virginia Woolf" 100 100 100)
      (list "Alan Turing" 90 80 40)
      (list "Anonymous" 30 55 10))
```

We can use a data definition to be precise about lists containing four-element lists.

```
;; An slist is either
;;      empty or
;;      (cons (list name assts mid final) sl), where
;;           name is a string,
;;           assts is a number between 0 and 100,
;;           mid is a number between 0 and 100,
;;           final is a number between 0 and 100, and
;;           sl is an slist.
```

This leads to a template for slists.

```
(define (my-slist-fun sl)
  (cond
    [(empty? sl) ...]
    [else ... (first (first sl)) ; name of first
          ... (second (first sl)) ; assts of first
          ... (third (first sl)) ; mid of first
          ... (fourth (first sl)) ; final of first
          ... (my-slist-fun (rest sl))...]))
```

Example: a function name-list which consumes an slist and produces the corresponding list of names.

```
(define (name-list sl)
  (cond
    [(empty? sl) empty]
    [else (cons (first (first sl)) ; name of first
                (name-list (rest sl)))]))
```

This code is less readable, because it uses only lists, instead of structures, and so is more generic-looking.

We can fix this with a few definitions.

```
(define (name x) (first x))
(define (assts x) (second x))
(define (mid x) (third x))
(define (final x) (fourth x))
(define (name-list sl)
  (cond
    [(empty? sl) empty]
    [else (cons (name (first sl))
                (name-list (rest sl)))]))
```

Dictionaries

You know dictionaries as books in which you look up a word and get a definition or a translation.

More generally, a dictionary contains a number of **keys**, each with an associated **value**.

Our task is to store the set of (key,value) pairs to support the operations *lookup*, *add*, and *remove*.

Association

```
;; As association (as) is
;;     (list k v), where
;;         k is a number (the key),
;;         v is a string (the value).
```

Association lists

```
;; An association list (al) is either
;;     empty or
;;     (cons a alst), where
;;         a is an association, and
;;         alst is an association list.
```

Without these data definitions, we might write a contract as

```
(listof (listof (union num string))[len=2])
```

and give more details in the purpose.

```
;; lookup-al: num al → (union string false)
;; Produces the value associated with k, or false if k not present.
(define (lookup-al k alst)
  (cond
    [(empty? alst) false]
    [(equal? k (first (first alst))) (second (first alst))]
    [else (lookup-al k (rest alst))]))
```

We will leave the add and remove functions as exercises.

This solution is simple enough that it is often used for small dictionaries.

For a large dictionary, association lists are inefficient in the case where the key is not present and the whole list must be searched.

Keeping the list in sorted order might improve some searches, but there is still a case where the whole list is searched.

In Module 8 we will see how to avoid this.

Why use lists containing lists?

If we define a **glist** as a list of two-element lists, each sublist holding name and grade, we could reuse the name-list function to produce a list of names from a glist.

Our original structure-based definitions of lists of students and grades require two different (but very similar) functions.

We will exploit this ability to reuse code written to use “generic” lists when we discuss abstract list functions later in the course.

Why use structures?

Structure is often present in a computational task, or can be defined to help handle a complex situation.

Using structures helps avoid some programming errors (e.g., accidentally extracting a list of grades instead of names).

Our design recipes can be adapted to give guidance in writing functions using complicated structures.

Most mainstream programming languages provide structures for use in larger programming tasks.

Different kinds of lists

When we introduced lists in Module 5, the items they contained were not lists. These were **flat lists**.

We have just seen **lists of lists** in our example of lists containing two-element flat lists.

In Module 9, we will see **nested lists**, in which lists may contain lists that contain lists, and so on to an arbitrary depth.

Goals of this module

You should understand the recursive definition of a natural number, and how it leads to a template for recursive functions that consume natural numbers.

You should understand how subsets of the integers greater than or equal to some bound b, or less than or equal to such a bound, can be defined recursively, and how this leads to a template for recursive functions that "count down" or "count up". You should be able to write such functions.

You should understand the principle of insertion sort, and how the functions involved can be created using the design recipe.

You should be able to use list abbreviations and quote notation for lists where appropriate.

You should be able to construct and work with lists that contain lists.

You should understand the similar uses of structures and fixed-size lists, and be able to write functions that consume either data.

You should be able to use association lists to implement dictionaries.