3 ADT Implementations

Last time, we defined an ADT via a signature and a specification. An *implementation* of ADT T (in a given programming language) is code in the language that implement the functions in the signature of T, in such a way that the code *satisfies* the equations in the specification. That is, the expressions that a specification equation says should be equal are actually equal when evaluated in the language of the implementation.

When we were describing ADTs, we made a point to say that we were not committing to a representation for the elements of the ADT. As we saw, we can still use the specification to do some amount of computation — in particular, predict the result of applying operations to various elements of the ADT.

Now, the job of the implementor of the ADT is to implement the spec, while the client of the ADT agrees to only rely on what's in the spec. There's a reason for that. In part, it allows one to *swap out* different implementations of the ADT, either because you have figured out a better way (more efficient perhaps) way to implement the ADT, or perhaps because on some machines or in some environments, an implementation has advantages that another does not. Irrespectively of the reason why, it just happens that there are many implementations to chose from sometimes, and as long as they all satisfy the same spec, and as long as the client of an ADT actually only relies on what's in the spec, then we can use any of the possible implementations without worrying about breaking code.

But how do we come up with implementation? When we implement an ADT, that is the time to address the question of how to *represent* the elements of the ADT. Different representations will lead to different implementations, and some of those implementations will have different properties — some will be more efficient in terms of time taken to compute results, or in memory needed to compute results, or even how much power is needed to compute results (which is important if you're working with a battery-based device like a mobile phone). If the implementations we use all implement the ADT and its spec, then we are covered in so far as functionality is concerned. So the only reasons to chose one implementation over another will have to do with something else than functionality.

It turns out that for many ADTs, there is a natural implementation that can be derived almost automatically from the signature of the ADT — but that implementation need not be the most efficient one. We will return to this point later, both in terms of defining this natural implementation and also address questions of efficiency.

Today, let's work with a simple language — Scheme — because it is simple, and you all know it. We look at a few examples of ADT implementations.

3.1 Implementing the Point ADT

Let's implement the POINT ADT from last lecture, with the following signature:

```
CREATORS
  cartesian : (Float, Float) -> Point
  polar :     (Float, Float) -> Point

OPERATIONS
  xCoord :              (Point) -> Float
  yCoord :              (Point) -> Float
  angleWithXAxis :      (Point) -> Float
  distanceFromOrigin :  (Point) -> Float
  distance :            (Point, Point) -> Float
  move :                (Point, Float, Float) -> Point
  add :                 (Point, Point) -> Point
  rotate :              (Point, Float) -> Point

  isEqual :             (Point, Point) -> Boolean
  isOrigin :            (Point) -> Boolean
```

(I will refer you to the lecture notes for last class for the specification.)

There are two straightforward possible representations for points. The easiest one is to simply represent a point as its cartesian coordinates, and when we construct a point in polar coordinates, we first convert it to cartesian coordinates.

Here is the data representation, where we use `define-struct` to define the representation of a point, as well as the creators:

```
(define-struct point (xpos ypos))

;; Creator (cartesian x-position y-position)
(define (cartesian x y)
  (make-point x y))

;; Creator (polar distance-from-origin angle-with-positive-x-axis)
;;    note that we convert from polar to cartesian immediately
(define (polar r theta)
  (if (< r 0)
      (error "(polar r theta): r must be >= 0")
      (cartesian (* r (cos theta))
                 (* r (sin theta)))))
```

We can now define the operations based on this representation:

```
(define (xCoord p)
  (point-xpos p))

(define (yCoord p)
  (point-ypos p))

(define (angleWithXAxis p)
  (atan (yCoord p) (xCoord p)))

(define (distanceFromOrigin p)
  (distance p (cartesian 0 0)))

(define (distance p q)
  (sqrt (+ (expt (- (xCoord p) (xCoord q)) 2)
           (expt (- (yCoord p) (yCoord q)) 2))))

(define (move p dx dy)
  (cartesian (+ (xCoord p) dx)
             (+ (yCoord p) dy)))

(define (add p q)
  (cartesian (+ (xCoord p) (xCoord q))
             (+ (yCoord p) (yCoord q))))

(define (rotate p theta)
  (let ((x (xCoord p))
        (y (yCoord p)))
    (cartesian (- (* (cos theta) x) (* (sin theta) y))
               (+ (* (sin theta) x) (* (cos theta) y)))))

(define (isEqual p q)
  (and (= (xCoord p) (xCoord q))
       (= (yCoord p) (yCoord q))))

(define (isOrigin p)
  (and (= (xCoord p) 0)
       (= (yCoord p) 0)))
```

I claim this is an implementation of the POINT ADT – this means checking that the equations of the specification hold. How do we check this? There are two ways of checking that an implementation satisfies a specification:

(1) Formal verification

(2) Testing

By formal verification here, I mean proving properties of programs the way you did it in *Logic and Computation*. Formal verification is very thorough, and can be used to establish that *no matter what the conditions under which the ADT is used*, the specification will hold. The downside is that formal verification is difficult. In fact, for us, here, we cannot really do it, because formally verifying programs that use floating point numbers is even more difficult than others.

The alternative to formal verification is, of course, testing. Testing is much easier than formal verification — you can do it in any language, for instance — but the downside is that you cannot test *everything*. Most functions can take one of an infinite number of possible values, and it is of course impossible to test infinitely many cases. So one often resorts to randomized testing, which is good, but may miss some cases that show a bug in your code.

When we have a specification, testing the equations in the specification is an obvious thing to do. In Racket (the implementation of Scheme you used in Fundies 1), where testing is done using `check-expect` and `check-within` (the latter for testing floating point numbers for equality), we can test the specification using:

```
;; testing a single random point

(let ((x (- (random 1000) 500))
      (y (- (random 1000) 500))
      (r (random 1000))
      (t (* 2 pi (random))))
  (check-within (xCoord (cartesian x y))
                x 0.0001)
  (check-within (xCoord (polar r t))
                (* r (cos t))  0.0001)
  (check-within (yCoord (cartesian x y))
                y 0.0001)
  (check-within (yCoord (polar r t))
                (* r (sin t))  0.0001)
  (check-within (angleWithXAxis (cartesian x y))
                (atan y x)  0.0001)
  (check-within (angleWithXAxis (polar r t))
                t 0.0001)
  (check-within (distanceFromOrigin (cartesian x y))
                (sqrt (+ (* x x) (* y y ))) 0.0001)
  (check-within (distanceFromOrigin (polar r t))
                r) 0.0001)
```

```
  ;; ...
  ;; ...
  ;; EXERCISE: write the rest of the tests
```

Here is some client code that exercises the ADT:

```
(define (main)
  (let ((point1 (polar 10 (/ pi 2)))
        (point2 (cartesian 50 -50))
        (point3 (polar 100 (/ pi 4))))
    (distance point1 (add (rotate (add point1 point2) (/ pi 2))
                          (rotate point3 (/ pi 8))))))
```

Running this codes gives the result

```
> (main)
153.79370506952165
```

The above representation for points has the advantage of being simple, at the cost of doing a lot of work in some cases. For instance, if we construct a point `(polar 10.0 (/ pi 4))` only to ask for `(distance-from-origin (polar 10.0 (/pi 4)))`, then the representation will take the point, convert it to cartesian coordinates, and then compute the distance from the origin of that point using the distance formula, doing a lot of work for extracting information that was available just from the arguments to the creators (since the distance from the origin, here, is clearly the `10.0` that was give as the first argument to `polar`)).

Let's try a different representation, which does not have that problem. The idea is to represent a point as either a cartesian position or a polar position, and have a flag in the representation that tells us if the point is in cartesian coordinates or polar coordinates. Again, we use a `define-struct` to define the representation of the point. Here is the data definition and the creators for this new implementation of the Point ADT:

```
(define-struct point (kind first second))
;;  kind is 'CARTESIAN or 'POLAR
;;   if kind is 'CARTESIAN, then first is x-pos and second is y-pos
;;   if kind is 'POLAR, then first is distance and second is angle


;; Creator (cartesian x-position y-position)
(define (cartesian x y)
  (make-point 'CARTESIAN x y))

;; Creator (polar distance-from-origin angle-with-positive-x-axis)
(define (polar r theta)
```

```
  (if (< r 0)
      (error "(polar r theta): r must be >= 0")
      (make-point 'POLAR r theta)))
```

We can now define the operations based on this new representation — we also define a few helper functions:

```
;; Helper functions

(define (polar-rep? p)
  (eq? (point-kind p) 'POLAR))

(define (cartesian-rep? p)
  (eq? (point-kind p) 'CARTESIAN))

(define (normalize angle)
  ;; takes an angle and normalizes it to between 0 and 2*pi
  (cond ((>= angle (* 2 pi)) (normalize (- angle (* 2 pi))))
        ((< angle 0) (normalize (+ angle (* 2 pi))))
        (else angle)))


;; Operations

(define (xCoord p)
  (if (cartesian-rep? p)
      (point-first p)
      (* (point-first p) (cos (point-second p)))))

(define (yCoord p)
  (if (cartesian-rep? p)
      (point-second p)
      (* (point-first p) (sin (point-second p)))))

(define (angleWithXAxis p)
  (if (cartesian-rep? p)
      (atan (point-second p) (point-first p))
      (point-second p)))

(define (distanceFromOrigin p)
  (if (cartesian-rep? p)
      (distance p (cartesian 0 0))
      (point-first p)))
```

```
(define (distance p q)
  (sqrt (+ (expt (- (xCoord p) (xCoord q)) 2)
           (expt (- (yCoord p) (yCoord q)) 2))))

(define (move p dx dy)
  (cartesian (+ (xCoord p) dx)
             (+ (yCoord p) dy)))

(define (add p q)
  (cartesian (+ (xCoord p) (xCoord q))
             (+ (yCoord p) (yCoord q))))

(define (rotate p theta)
  (if (cartesian-rep? p)
      (cartesian (- (* (cos theta) (point-first p))
                    (* (sin theta) (point-second p)))
                 (+ (* (sin theta) (point-first p))
                    (* (cos theta) (point-second p))))
      (polar (point-first p) (+ (point-second p) theta))))


;; BUGGY! Exercise -- spot and correct the bug!
(define (isEqual p q)
  (if (cartesian-rep? p)
      (and (= (point-first p) (point-first q))
           (= (point-second p) (point-second q)))
      (and (= (point-first p) (point-first q))
           (= (normalize (point-second p))
              (normalize (point-second q))))))

(define (isOrigin p)
  (if (cartesian-rep? p)
      (and (= (point-first p) 0)
           (= (point-second p) 0))
      (= (point-first p) 0)))
```

Note that my implementation of `isEqual` is buggy – it shouldn't pass the tests. I could correct it, of course, but instead I'll leave it to you to find the bug and to suggest a correction. Hint: it's a pretty big bug, and not one of those subtle ones!

Aside from this problem with `isEqual`, the rest of the implementation actually satisfies the specification, as testing would suggest. You can try the `main` function from earlier with this

code and it will still give you back the result `153.79370506952165` — which is what we expect from the specification anyways! (Try it!)

Note that the reason why our client (here, function `main`) works with both implementations is that the client only relies on the signature and specification of the `Point` ADT to work with points. Were we to write a *bad* client that somehow used information beyond that provided by the signature or specification, for instance, details of the representation, then we could break it by changing the implementation. For example, consider the following bad client function that uses details of the representation in the first implementation of points above:

```
;; a bad client for point2.rkt
(define (bad-main)
  (let ((point1 (polar 10 (/ pi 2)))
        (point2 (cartesian 50 -50))
        (point3 (polar 100 (/ pi 4))))
    ;; add the two points
    ;; rotate around the origin by 90 degrees
    ;; compute the distance between point1 and the new point
    (distance point1 (add (rotate (cartesian (+ (point-xpos point1)
                                                (point-xpos point2))
                                             (+ (point-ypos point1)
                                                (point-ypos point2)))
                                  (/ pi 2))
                          (rotate point3 (/ pi 8))))))
```

If we try this with the first implementation, we get a result — still `153.79370506952165`. But if we try it with the second implementation, it fails miserably: `(point-xpos point1)`, for example, complains of an error that there is no `xpos` field in structure `point`.

The client relied on something beyond the signature or the specification of the ADT — and since as we saw the client's job was to not rely on anything that is not in the signature or the specification, this failure is the client's fault. Part of the process in this course will be helping the client make sure they do not break the contract.

3.2 Implementing the List ADT

So let's look at another example of ADT implementation. Consider an ADT for lists of integers, the List ADT. You're all used to the ADT for lists that uses `empty` and `cons` as creators. Let's look at one that uses a variant of those, for variety's sake.

```
CREATORS
  empty :      () -> List
```

```
  singleton :  (Int) -> List
  merge :      (List,List) -> List

OPERATIONS:
  isEmpty : (List) -> Boolean
  first :   (List) -> Int
  rest :    (List) -> List
  length :  (List) -> Int

  isEqual : (List) -> Boolean
```

Here is the corresponding specification:

$$\texttt{isEmpty(empty())} = \mathit{true}$$

$$\texttt{isEmpty(singleton(}n\texttt{))} = \mathit{false}$$

$$\texttt{isEmpty(merge(}L\texttt{,}M\texttt{))} = \begin{cases} \mathit{true} & \text{if } \texttt{isEmpty(}L\texttt{)} = \texttt{isEmpty(}M\texttt{)} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{first(singleton(}n\texttt{))} = n$$

$$\texttt{first(merge(}L\texttt{,}M\texttt{))} = \begin{cases} \texttt{first(}M\texttt{)} & \text{if } \texttt{isEmpty(}L\texttt{)} = \mathit{true} \\ \texttt{first(}L\texttt{)} & \text{otherwise} \end{cases}$$

$$\texttt{rest(singleton(}n\texttt{))} = \texttt{empty()}$$

$$\texttt{rest(merge(}L\texttt{,}M\texttt{))} = \begin{cases} \texttt{rest(}M\texttt{)} & \text{if } \texttt{isEmpty(}L\texttt{)} = \mathit{true} \\ \texttt{merge(rest(}L\texttt{),}M\texttt{)} & \text{otherwise} \end{cases}$$

$$\texttt{length(empty())} = 0$$

$$\texttt{length(singleton(}n\texttt{))} = 1$$

$$\texttt{length(merge(}L\texttt{,}M\texttt{))} = \texttt{length(}L\texttt{)} + \texttt{length(}M\texttt{)}$$

$$\texttt{isEqual(}L\texttt{,}M\texttt{)} = \begin{cases} \mathit{true} & \text{if } \texttt{isEmpty(}L\texttt{)} = \texttt{isEmpty(}M\texttt{)} = \mathit{true} \\ \mathit{true} & \text{if } \texttt{isEmpty(}L\texttt{)} = \texttt{isEmpty(}M\texttt{)} = \mathit{false} \\ & \text{and } \texttt{first(}L\texttt{)} = \texttt{first(}M\texttt{)} \\ & \text{and } \texttt{isEqual(rest(}L\texttt{),rest(}M\texttt{))} = \mathit{true} \end{cases}$$

Notice that some of the equations are conditional — their result depends on conditions. Also, there are some combination of operations and creators that do not have a corresponding equation — `first(empty())`, for instance, is unaccounted for. If an operation does not have an equation, then this simply means that the implementation is free to do whatever it

wants for that operation when given a value equal to what that creators creates. Often, this means that the operation is undefined for that value, as in the example above, since taking the first element of an empty list is undefined. Often, an implementation will elect to report an error or throw an exception when that occurs.

As with the previous examples, we have two possible representations for the List ADT in Scheme. The first one, the simplest, is to simply cheat and use a Scheme list to represent an element of the List ADT (suitably wrapped in a structure, for clarity):

```
(define-struct list (content))

;; Creators

(define (empty)
  (make-list '()))

(define (singleton a)
  (make-list (cons a '())))

(define (merge L M)
  (define (app l1 l2)
    (if (null? l1)
        l2
        (cons (first l1)
              (app (rest l1) l2))))
  (make-list (app (list-content L)
                  (list-content M))))


;; Operations

(define (isEmpty L)
  (let ((l (list-content L)))
    (null? l)))

(define (first L)
  (let ((l (list-content L)))
    (if (null? l)
        (error "(first L): L is empty")
        (car l))))

(define (rest L)
  (let ((l (list-content L)))
```

```
    (if (null? l)
        (error "(rest L): L is empty")
        (make-list (cdr l)))))

(define (length L)
  (let ((l (list-content L)))
    (length l)))

(define (isEqual L M)
  (or (and (isEmpty L) (isEmpty M))
      (and (not (isEmpty L)) (not (isEmpty M))
           (= (first L) (first M))
           (isEqual (rest L) (rest M)))))
```

Here is some client code that exercises the ADT:

```
;; convert a List to a string
(define (to-string L)
   (if (isEmpty L)
      ""
      (string-append " "
                     (number->string (first L))
                     " "
                     (to-string (rest L)))))

(define (main)
  (let ((list1 (merge (merge (singleton 1)
                             (singleton 2))
                      (merge (singleton 3)
                             (singleton 4))))
        (list2 (merge (singleton 99)
                      (merge (singleton 98)
                             (singleton 97)))))
    (to-string (merge (singleton (+ (length list1) (length list2)))
                      (merge list1 list2)))))
```

Executing this client yields:

```
> (main)
" 7  1  2  3  4  99  98  97 "
```

As before we can develop tests for this implementation of the ADT. Here, we can also use formal verification. We can transliterate the above into ACL2, and use it to prove the required equations as *theorems*. Here is the code, for your perusal:

```
;; A List is a '(content) where content is an ACL2 list

(defun mk-list (l) (cons l nil))
(defun list-content (L) (car l))

;; needed for ACL2
(defun null? (L) (atom L))

;; Operations

(defun L-isEmpty (L)
  (let ((l (list-content L)))
    (null? l)))

(defun L-first (L)
  (let ((l (list-content L)))
    (car l)))

(defun L-rest (L)
  (let ((l (list-content L)))
    (mk-list (cdr l))))

;; easier to just redefine length for acl2 lists
(defun acl2len (L)
  (if (atom L)
    0
    (+ 1 (acl2len (cdr L)))))

(defun L-length (L)
  (let ((l (list-content L)))
    (acl2len l)))


; IS-EQUALS is left as an exercise

;; Creators

(defun L-empty ()
  (mk-list '()))

(defun L-singleton (a)
```

```
  (mk-list (cons a '())))

(defun app (l1 l2)
  (if (atom l1)
      l2
      (cons (car l1)
            (app (cdr l1) l2))))

(defthm app-atoms (= (atom (app l1 l2)) (and (atom l1) (atom l2))))

(defun L-merge (L M)
  (let ((l (list-content L))
    (m (list-content M)))
    (mk-list (app l m))))

(defthm lemma-empty
  (= (L-isEmpty L) (atom (list-content l))))

(defthm lemma-empty-merge
  (= (L-isEmpty (L-merge L M))
     (atom (app (list-content L) (list-content M)))))

(defthm lemma-length
  (= (L-length L) (acl2len (list-content L))))

(defthm lemma-length-cons
  (= (acl2len (cons x y)) (+ 1 (acl2len y))))

(defthm lemma-length-app
  (= (acl2len (app x y)) (+ (acl2len x) (acl2len y))))

(defthm isempty-1 (= (L-isEmpty (L-empty)) t))
(defthm isempty-2 (= (L-isEmpty (L-singleton n)) nil))
(defthm isempty-3 (= (L-isEmpty (L-merge L M))
                     (if (and (L-isEmpty L) (L-isEmpty M)) t nil)))

(defthm first-2 (= (L-first (L-singleton n)) n))
(defthm first-3 (= (L-first (L-merge L M))
           (if (L-isEmpty L) (L-first M) (L-first L))))

(defthm rest-2 (= (L-rest (L-singleton n)) (L-empty)))
(defthm rest-3 (= (L-rest (L-merge L M))
```

```
          (if (L-isEmpty L) (L-rest M) (L-merge (L-rest L) M))))

(defthm length-1 (= (L-length (L-empty)) 0))
(defthm length-2 (= (L-length (L-singleton n)) 1))
(defthm length-3 (= (L-length (L-merge L M))
                    (+ (L-length L) (L-length M))))
```

Note that I renamed the operations and creators so that they take `L-` as a prefix to avoid name clashes with existing operations.

Running the above code proves all the theorems, which establishes the correctness of the implementation — that is, that it correctly implements the specification of the ADT.

So that's one implementation of the LIST ADT, using the underlying lists of Scheme (and ACL2). Let's look at another implementation, one that does not use cheating quite so much. The idea is similar to the second implementation of the POINT ADT. Here, a list is an element of a structure with three fields, where the first field of the structure indicate the kind of list, that is, an empty list (in which case the other two fields are unused), a singleton list (in which case the second field is the element), and a merge of two lists (in which case the other two fields contain the lists to be merged). Here is the complete implementation:

```
;; Representation: a List is a struct where kind is one of
;;   <'EMPTY ? ?>
;;   <'SINGLETON a ?>
;;   <'MERGE L M>

(define-struct list (kind field1 field2) #:transparent)

;; Creators

(define (empty)
  (make-list 'EMPTY 0 0))

(define (singleton a)
  (make-list 'SINGLETON a 0))

(define (merge L M)
  (make-list 'MERGE L M))


;; Helper functions

(define (kind-empty? L)
  (eq? (list-kind L) 'EMPTY))
```

```
(define (kind-singleton? L)
  (eq? (list-kind L) 'SINGLETON))
(define (kind-merge? L)
  (eq? (list-kind L) 'MERGE))


;; Operations

(define (isEmpty L)
  (or (kind-empty? L)
      (and (kind-merge? L)
           (isEmpty (list-field1 L))
           (isEmpty (list-field2 L)))))

(define (first L)
  (cond [(kind-empty? L) (error "(first L): L is empty")]
        [(kind-singleton? L) (list-field1 L)]
        [(isEmpty (list-field1 L)) (first (list-field2 L))]
        [else (first (list-field1 L))]))

(define (rest L)
  (cond [(kind-empty? L) (error "(rest L): L is empty")]
        [(kind-singleton? L) (empty)]
        [(isEmpty (list-field1 L)) (rest (list-field2 L))]
        [else (merge (rest (list-field1 L)) (list-field2 L))]))

(define (length L)
  (cond [(kind-empty? L) 0]
        [(kind-singleton? L) 1]
        [else (+ (length (list-field1 L)) (length (list-field2 L)))]))

(define (isEqual L M)
  (or (and (isEmpty L) (isEmpty M))
      (and (not (isEmpty L)) (not (isEmpty M))
           (= (first L) (first M))
           (isEqual (rest L) (rest M)))))
```

We can check that this implementation satisfies the specification by testing it, or again by formal verification. We can also ensure that the client code we gave above works with this implementation (which we know it should after we confirm that the implementation satisfies the specification.) For completeness, here is the ACL2 code and theorem for this representation of lists:

```
(defun mk-list (kind field1 field2)
  (list kind field1 field2))
(defun list-kind (L)
  (car L))
(defun list-field1 (L)
  (cadr L))
(defun list-field2 (L)
  (caddr L))


;; Creators

(defun L-empty ()
  (mk-list 'EMPTY 0 0))

(defun L-singleton (a)
  (mk-list 'SINGLETON a 0))

(defun L-merge (L M)
  (mk-list 'MERGE L M))


;; Helper functions

(defun kind-empty? (L)
  (= (list-kind L) 'EMPTY))
(defun kind-singleton? (L)
  (= (list-kind L) 'SINGLETON))
(defun kind-merge? (L)
  (= (list-kind L) 'MERGE))


;; Operations

(defun L-isEmpty (L)
  (or (kind-empty? L)
      (and (kind-merge? L)
       (L-isEmpty (list-field1 L))
       (L-isEmpty (list-field2 L)))))

(defun L-first (L)
```

```
  (cond
   ((atom L) L)
   ((kind-empty? L) "(L-first L): L is empty")
   ((kind-singleton? L) (list-field1 L))
   ((L-isEmpty (list-field1 L)) (L-first (list-field2 L)))
   (t (L-first (list-field1 L)))))

(defun L-rest (L)
  (cond
   ((atom L) L)
   ((kind-empty? L) (L-empty)) ;; -- default
   ((kind-singleton? L) (L-empty))
   ((L-isEmpty (list-field1 L)) (L-rest (list-field2 L)))
   (t (L-merge (L-rest (list-field1 L)) (list-field2 L)))))

(defun L-length (L)
  (cond
   ((atom L) 0)
   ((kind-empty? L) 0)
   ((kind-singleton? L) 1)
   (t (+ (L-length (list-field1 L)) (L-length (list-field2 L))))))

; IS-EQUALS is left as an exercise

(defthm isempty-1 (= (L-isEmpty (L-empty)) t))
(defthm isempty-2 (= (L-isEmpty (L-singleton n)) nil))
(defthm isempty-3 (= (L-isEmpty (L-merge L M))
                     (if (and (L-isEmpty L) (L-isEmpty M)) t nil)))

(defthm first-2 (= (L-first (L-singleton n)) n))
(defthm first-3 (= (L-first (L-merge L M))
                   (if (L-isEmpty L) (L-first M) (L-first L))))

(defthm rest-2 (= (L-rest (L-singleton n)) (L-empty)))
(defthm rest-3 (= (L-rest (L-merge L M))
                  (if (L-isEmpty L) (L-rest M) (L-merge (L-rest L) M))))

(defthm length-1 (= (L-length (L-empty)) 0))
(defthm length-2 (= (L-length (L-singleton n)) 1))
(defthm length-3 (= (L-length (L-merge L M))
                    (+ (L-length L) (L-length M))))
```