A record in Scheme is a data structure that encloses some fixed number of values, each of which is given a field name to indicate its status within the record. Programmers can define new types of values in Scheme (distinct from built-in types of values, such as numbers, Booleans, characters, procedures, pairs, vectors, and the null value) by providing record type definitions. A record type definition introduces a new type, giving it a name and identifying the fields that each value of this type will contain.

For instance, in a program that astronomers might use for keeping an inventory of interesting stars, the programmer might well define a record type for stars, with the fields of each record containing different pieces of information about a star. The record definition might look like this:

```
(define-record-type star
  (make-star name distance apparent-magnitude spectral-class)
  star?
  (name star-name)
  (distance star-distance)
  (apparent-magnitude star-apparent-magnitude)
  (spectral-class star-spectral-class))
```

This defines `star` as a record type. Each record of this type will have four fields, called `name`, `distance`, `apparent-magnitude`, and `spectral-class`. As in pairs, lists, and vectors, Scheme imposes no restriction on the type of value that can be stored in any of the fields.

A record type definition implicitly defines several procedures for operating on values of the newly defined type:

- A constructor, with the interface specified in the third subform of the record type definition. That subform is a list in which the first element is an identifier giving the name of the constructor and the remaining elements are identifiers to be used as parameters of the constructor. (In the example, the constructor is named `make-star`, and it takes four arguments, which will become the initial values of the record it constructs and returns.

- A type predicate that takes one argument, which can be any value, and returns `#t` if that argument belongs to the new record type and `#f` if it does not. The name of the type predicate is the fourth subform of the record type definition. (In the example, the type predicate is named `star?`.)

- For each field of the new record type, an accessor procedure that takes one argument, a record of the type, and returns the value stored in a particular field of that record. The fifth and subsequent subforms of the record type definition give the names of the fields and the corresponding accessor procedures. (In the example, there are four fields and hence four accessors, and the names of the accessors are `star-name`, `star-distance`, `star-apparent-magnitude`, and `star-spectral-class`.)

After the type definition, the programmer can begin using the implicitly defined procedures as if they had been introduced with `define`. However, it is neither necessary nor possible actually to write out definitions for the procedures that the record type definition generates.

If a record type definition occurs inside a library, the library can export any of the procedures that the record type definition implicitly defines, but it cannot export the record type itself. So, for
instance, a library containing the above definition of the star record type can export any or all of the procedures make-star, star?, star-name, star-distance, star-apparent-magnitude, and star-spectral-class, but cannot export the record type star. Similarly, application programs and other libraries can import such procedures, but cannot import the record type definitions that produce them.

Exercise 0: Write a record type definition for a record type called pixel, with three fields named x-coordinate, y-coordinate, and color. How many procedures are implicitly defined by your definition? How many arguments does each one expect to receive?

Exercise 1: Test your solution to the previous exercise by writing Scheme expressions to construct two pixels, one at coordinates (35, -17) with the symbol red as its color value and the other at coordinates (5, 12) with the symbol blue as its color value.

Exercise 2: Write a procedure distance that takes two pixels as arguments and computes and returns the straight-line Euclidean distance between them.

Records of the types defined above are immutable, in the sense that, once the record has been created, none of the values in its fields can be changed. The record type definition doesn’t even generate a procedure for updating a field of a record (in the way that, say, the vector-set! procedure updates a position within an array).

For some purposes, we may want to be able to update one or more of the fields of a record, replacing the value initially stored in that field when the record was created with a different value, while preserving the identity of the record as a container. We can arrange this by adding to the list that already specifies the name of the field and the name of the accessor procedure a third element, giving the name of the mutator for that field.

For example, here’s a record type definition that might appear in the code for a multi-player role-playing game:

```
(define-record-type avatar
  (make-avatar name image)
  avatar?
  (id-number avatar-id-number)
  (name avatar-name)
  (image avatar-image)
  (charisma avatar-charisma)
  (intelligence avatar-intelligence)
  (agility avatar-agility avatar-agility-set!)
  (strength avatar-strength avatar-strength-set!)
  (location avatar-location avatar-location-set!)
  (possessions avatar-possessions avatar-possessions-set!))
```

The constructor, make-avatar, would initialize all of these fields, supplying a unique, internally generated ID number, a name and image supplied by the player as parameters, and randomly generated levels of charisma, intelligence, agility, and strength, setting the location for each avatar to the place at which The Quest Begins, and issuing some minimal set of standard possessions to each player. In the course of the avatar’s misadventures, the contents of the agility and strength fields might change, and the location and possession fields would also naturally change, but the values in the other fields are fixed—presumably nothing that happens in the game can affect the avatar’s charisma and intelligence.

For each mutable field, the define-record-type expression will tacitly define not only an accessor but also a mutator procedure with the specified name. (For instance, the procedure for changing an avatar’s location would be avatar-location-set!.) Mutator procedures take two
arguments; the first is the record containing the field to be modified and the second is the new
value to be stored in that field, overwriting the old value. As the exclamation point indicates,
mutators are destructive and can lead to data loss. The value returned by a field mutator is
unspecified, since mutators are called solely for their side effects.

**Exercise 3:** Revise the record type definition for the `pixel` type so that the `color` field is
mutable, giving the mutator procedure the name `pixel-color-set!`.

**Exercise 4:** Test the revised record type definition by writing Scheme expressions to construct
two new pixels with the same coordinates and colors as before, and then to change the color of the
second to match the color of the first.