Records in Scheme
CSC/MAT 208: Discrete structures
Department of Computer Science
Department of Mathematics and Statistics
Grinnell College

Record type definitions

A record in Scheme is a data structure that encloses some fixed number of values, each of which is given a name (called the 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 amateur astronomers will 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
  (fields name distance apparent-magnitude 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 that takes as many arguments as the record has fields and returns a record containing those arguments in its fields. The name of the constructor is formed by prefixing make- to the name of the record type. (In the example, the constructor would be make-star, and it would take four arguments, since star records have four fields.)

- 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 formed by postfixing a question mark to the name of the record type. (In the example, the type predicate would be 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 name of an accessor procedure is formed by connecting the name of the record type and the name of the field with a hyphen. (In the example, there would be four accessors, and their names would be 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? What are the names of those procedures? What is the arity of each one (that is, how many arguments does each one expect to receive)?

**Exercise 1:** Write a record type definition for a structure that keeps track of several items of information about a disk partition: the number of bytes in each sector, the number of file allocation tables, the total number of sectors, the number of sectors in each cluster, the number of sectors in each file allocation table, the number of sectors in each track, the number of "reserved" sectors, the number of "hidden" sectors, the number of root-directory entries, the number of read/write heads available in the drive, and a numerical descriptor identifying the type of medium used in the disk.
Exercise 2: Is it possible to define a record type that has no fields? Could this be useful in any way?

Mutable fields

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 replacing the name of the field inside the record type definition with a two-element list in which the symbol mutable is the first element and the name of the field is the second.

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
  (fields id-number
    name
    image
    charisma
    intelligence
    (mutable agility)
    (mutable strength)
    (mutable location)
    (mutable possessions)))
```

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, 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 (short of total annihilation) 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 a name consisting of the record type name, a hyphen, the field name, another hyphen, and the postfix set!. 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.

Controlling procedure names

In many cases, the default names that define-record-type supplies for the constructors, type predicates, accessors, and mutators that it defines are entirely suitable, convenient, and readable. Sometimes, however, you may want to provide different names, overriding the defaults. Scheme’s record type definitions provide a mechanism for doing this.

Specifically:

- If you replace the record type name with a list of three identifiers, the first identifier becomes the name of the record type, the second the name of the constructor, and the third the name of the type predicate.

- If you replace the name of an immutable field, inside the fields list, with a list of three symbols in which the first element is the symbol immutable, then the second element becomes the name of the field and the third becomes the name of the accessor procedure.

- If you replace (mutable xxx), where xxx is the name of a mutable field inside the fields list, with a list of four symbols in which the first element is the symbol mutable, then the second element becomes the name of the field, the third becomes the name of the accessor procedure, and the fourth becomes the name of the mutator procedure.

For instance, if Scheme did not already have a built-in pair data type, we could define a functionally equivalent record type called pair, with fields
named `left` and `right`, and still preserve the traditional names for the constructor, type predicate, field accessors, and field mutators, thus:

```scheme
(define-record-type (pair cons pair?)
  (fields (mutable left car set-car!)
            (mutable right cdr set-cdr!)))
```

**Exercise 3:** What would be the names of the various procedures associated with the `pair` type if we just defined it as

```scheme
(define-record-type pair
  (fields (mutable left) (mutable right)))
```

**Exercise 4:** Rewrite the definition of the `avatar` record type so that the mutator procedure for each mutable field begins with `update` instead of ending with `set!`, but keeping the exclamation point at the end. (That is, the mutator for the `strength` field should be `update-avatar-strength!`, and so on.)