A record in Scheme is a data structure that comprises some fixed number of values, each of which is given a name (the field name) to indicate its status and situation within the record. Programmers can create values of new types in Scheme (distinct from built-in types of values, such as numbers, Booleans, characters, procedures, pairs, vectors, and the null value), specifying the new types in record type definitions. A record type definition introduces a new type, giving it a name and specifying the fields that a value of the record 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 type definition for the new type 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. The first subexpression of the define-record-type-expression must be an identifier, and this identifier will be the name of the new record type.

Each record of the star type will have four fields: 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 builds and returns a newly allocated instance of the record type. The second subexpression of the define-record-type-expression should be an S-expression containing one or more identifiers. The first identifier will be the name of the constructor procedure, and the remaining identifiers, if any, will be the names of the fields that the constructor is expected to initialize. These names will be parameters of the constructor, so that the appropriate values to be stored into the specified fields initially must be provided as arguments to the constructor procedure when it is invoked. (In the example, the constructor is make-star, and it takes four arguments, corresponding to the four fields of the records. In this case, the constructor is expected to initialize all four of the fields in the record that it builds before returning it.)

- A type predicate takes one argument, which can be any Scheme value, and returns #t if the type of that argument is the new record type and #f if it is not. The third subexpression of the define-record-type-expression must be an identifier that will be the name of the type predicate. (In the example, the type predicate is star?.)

- An accessor procedure takes one argument, a record of the type, and returns the value stored in a particular field of that record. There will be an accessor procedure for each field. Each of the subexpressions of the define-record-type-expression following the type-predicate identifier must be an S-expression containing either two or three identifiers. The first identifier will be the name of the field, and the second will be the accessor procedure for that
field. (In the example, the names of the accessor procedures are \texttt{star-name}, \texttt{star-distance}, \texttt{star-apparent-magnitude}, and \texttt{star-spectral-class}.)

- A \texttt{modifier} procedure takes two arguments, a record of the type and a value, and stores the value into a particular field of that record, destructively replacing the value previously stored in that field. Modifiers are invoked only for this side effect and the value that they return is unspecified. A modifier procedure for a particular field is defined if, and only if, the S-expression that specifies the name and accessor for the field also includes a third identifier, which will be the name of the modifier procedure. (In the example, none of the fields has a modifier procedure, so \texttt{star} records are “immutable” — once constructed, they can never be changed.)

After the type definition, the programmer can begin using the implicitly defined procedures as if they had been introduced with \texttt{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 \texttt{star} record type can export any or all of the procedures \texttt{make-star}, \texttt{star?}, \texttt{star-name}, \texttt{star-distance}, \texttt{star-apparent-magnitude}, and \texttt{star-spectral-class}, but cannot export the record type \texttt{star}. Similarly, application programs and other libraries can import such procedures, but cannot import the record type definitions that produce them.

There are conventional rules for deriving the names of constructors, type predicates, accessor procedures, and modifier procedures from the programmer-selected names for the type and for each field:

- The name of the constructor is formed by prefixing \texttt{make-} to the base name for the type.
- The name of the type predicate is formed by suffixing a question mark to the base name for the type.
- The name of each accessor procedure is formed by concatenating the base name for the type, a hyphen, and the name of the field to be selected.
- The name of each modifier procedure is formed by concatenating the prefix \texttt{set-}, the base name for the type, a hyphen, the name of the field to be modified, and an exclamation point. (For instance, if we had provided a modifier procedure for the \texttt{distance} field of a \texttt{star} record, we would have named it \texttt{set-star-distance!}.)

These are conventions rather than requirements, so they can be overridden when there is a good reason to do so. The most common such reason is that the constructor or type predicate that \texttt{define-record-type} builds fails to enforce preconditions or constraints that the programmer wants to impose. In such cases, one can give the preliminary constructor or type predicate that \texttt{define-record-type} builds some name like ‘\texttt{proto-make-star}’ and then define the real \texttt{make-star} procedure, with the additional preconditions, using \texttt{proto-make-star} as a helper procedure.