Fundamentals of Computer Science I (CS151.01 2006F)

Records in Scheme

Summary: As we've noted in the past, computer science is the study of data types and algorithms. This semester, we have emphasized algorithm design and a few key data types (list, vectors, basic types). In this reading, we consider *records*, one of the key techniques for designing data types.

Contents:

- Introduction
- Designing Records
- Implementing Records in Scheme
  - An Example: Compounds
- Searching and Sorting Records

Introduction

In our exploration of Scheme, we've seen a number of *data structures* that allow us to organize data. A list is a dynamic data structure with a variable number of components. A vector is a data structure with a fixed number of components, each of which you can quickly access by number.

A *record* is a data structure with a fixed number of components, known as the the *fields* of the record. Each field has a name, and one uses that name to access that particular field of a record. Ideally, the structure should provide random access to each field; in other words, one should be able to inspect and possibly modify the contents of any field, independently, without taking the time to access any of the others.

For instance, an astronomical database that keeps track of information about various stars might assemble the data into records, one record for each star. The fields of such a record might include `name`, `right-ascension`, `declination`, `visual-magnitude`, `spectral-type`, and so on. Similarly, a library's card catalog includes a record for each book or other item that the library holds, with fields like `author`, `title`, `imprint`, `call-number`, and `checked-out?`.

Designing Records

Scheme does not provide any built-in record types, so programmers have to define their own. Fortunately, defining records is straightforward. To create a new record type `T` in Scheme, one should define a procedure to carry out each of the following operations:

- *Construct* and return a new record of type `T`, providing space for each field of the record and initializing any fields for which an initialization is possible and appropriate. The procedure that performs this operation is called a *constructor* for the type `T`.
- Given any record of type `T`, recover from it the value stored in a particular field. There should be one such procedure for each field that exists in records of type `T`. Such procedures are called *selectors*.
- Given any record of type `T`, store in a particular field of that record a new value `obj`. Procedures of

this sort are called *mutators*. Depending on the particular application, it may be a good idea to provide a mutator for each field of a record, or for some and not others, or for none. For example, it might make sense to provide a mutator for the `checked-out?` field of the record for a library book, but one would probably not want to provide a mutator for the `title` field.

There's an obvious analogy between this collection of procedures and the provision that Scheme makes for its built-in data structures (`vector`, `string`, and `pair`). In the case of `string`, for instance, Scheme supplies a constructor (`make-string`), a selector (`string-ref`), and a mutator (`string-set!`). The only difference is that the characters in a string are accessed by position number, so that it is sufficient to provide just one selector and just one mutator. The fields in a record are accessed by separate procedures, so that one must provide as many selector procedures as there are fields and as many mutator procedures as there are mutable fields.

In some cases it is a good idea to define some other primitives as well:

- A *type predicate* that determines, for any given Scheme value, whether it is a record of type `T`.
- An *equality tester* that determines, given two records of type `T`, where they should be counted as equal.
- A *copier* that takes any record of type `T` and returns another record of the same type, with the same contents.
- A *displayer* that takes any record of type `T` and outputs a description of it and its contents, through a given output port or, if none is given, the standard output port.
- A *reader* that can input a record of type `T`.

Implementing Records in Scheme

To implement a record type in Scheme, one has to select an existing type and figure out how to perform the record operations using only values of the existing type. There are various ways to do this. One might, for instance, make each record an *association list* -- a list of pairs, each pair having a field name as its car and the value stored in that field as its cdr. This can be implemented elegantly in Scheme, but it does not provide random access to the field values, since it is necessary to walk down the association list one pair at a time until one arrives at the pair corresponding to the field one wishes to examine.

One common and widely recommended approach that does provide random access is to use vectors as the implementation type. A record containing *n* fields can be identified with a vector of *n* + 1 elements -- the first being a symbol identifying the record type, and the subsequent ones being the values of the various fields.

An Example: Compounds

For instance, in the file `compound.scm` you'll find a vector implementation of a record type suitable for storing some of the chemical and physical properties of an inorganic compound, with fields for the name and chemical formula of the compound, its molecular weight, its melting and boiling points, and its usual color. Here are some examples of the use of the procedures defined in that file. We construct a new compound with `make-compound`; get its fields with `get-compound-name`, `get-compound-formula`, `get-compound-molecular-weight`, and so on and so forth; set is

fields with `set-compound-name!`, `set-compound-formula`, and so on and so forth; display it with `compound-display`; make a copy with `compound-copy`, and compare compounds with `compound=?`.

First, let's construct a compound and give it a name:

```
> (define sample
    (make-compound "gadolinium iodide"
                   "GdI3"
                   537.96
                   926
                   1340
                   'yellow))
```

We can print it back out immediately, just to confirm that the fields got filled in correctly.

```
> (compound-display sample)
#compound(name: gadolinium iodide
          formula: GdI3
          molecular-weight: 537.96
          melting-point: 926
          boiling-point: 1340
          color: yellow)
```

We can also examine a few of its fields separately.

```
> (get-compound-formula sample)
"GdI3"
> (get-compound-boiling-point sample)
1340
```

We can verify that it satisfies the type predicate.

```
> (compound? sample)
#t
```

We can even try to build a vector that looks like the representation of a compound, but without using the constructor. However, we will find that the result does not satisfy the type predicate.

```
> (define attempted-fake
    (vector (list 'compound)
            "potassium fluoride"
            "KF"
            58.10
            858
            1505
            'colorless))
> (compound? attempted-fake)
#f
```

We can make a copy of the compound and make sure that the copy resembles the original.

```
> (define clone (compound-copy sample))
> (compound? clone)
#t
> (get-compound-color clone)
yellow
```

We can even check that they are identical as compounds.

```
> (compound=? sample clone)
#t
```

We can modify a field of the original compound destructively and observe the effect.

```
> (set-compound-color! sample 'orange)
> (compound-display sample)
#compound(name: gadolinium iodide
          formula: GdI3
          molecular-weight: 537.96
          melting-point: 926
          boiling-point: 1340
          color: orange)
```

Now the original and the copy are no longer identical.

```
> (compound=? sample clone)
#f
```

Let's try to modify a compound to contain invalid data. As you can see, the invariants are preserved.

```
> (set-compound-melting-point! clone -12000)
set-compound-melting-point!: The melting point of a compound must be a real number greater than -273.15.
> (set-compound-color! clone -12000)
set-compound-color!: The color of a compound must be a Scheme symbol.
> (compound-display clone)
#compound(name: gadolinium iodide
         formula: GdI3
         molecular-weight: 537.96
         melting-point: 926
         boiling-point: 1340
         color: yellow)
```

Searching and Sorting Records

The sorting and searching methods that we have been studying are easily adapted to lists and vectors in which the elements are records, to be sorted according to the values in some field. For instance, suppose that we are given a vector of records of type `compound` and asked to arrange the records in order of ascending melting point. All we need to do is define an appropriate comparison procedure and specialize our preferred sorting algorithm to use it:

```
(define sort-by-melting-point
  (lambda (lst)
    (merge-sort lst (lambda (compound1 compound2)
                      (<= (compound-melting-point compound1)
                          (compound-melting-point compound2))))))
```

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/2.5/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.