211lab01b

Lab 01b: Structures

Originally developed by Dr. Greiner

- Lab 01b: Structures
  - Design Recipe Reminder
  - Structured Data
  - Defining the Data:
  - Examples of the Data
  - Functions Using Structures
  - Combining Cases and Structures
  - Stepping back: Recipe redux

Instructions for students & labbies: Students should read and use DrScheme on the exercises at their own pace, while labbies wander among the students, answering questions, bringing the more important ones to the lab's attention.

Design Recipe Reminder

Let's quickly recall the design recipe (so far). We'll fill in some new steps during this lab, so we'll start with some funny numbering.

1. **Data Definition**
2. **Data Examples**
3. (not yet fully introduced)
4. **Function contract, purpose, header**
5. **Function examples**
6. **Function body**
7. If data has cases, use a cond with one branch per case.
8. **Function testing**

Structured Data

You've been hired by the university to help with this year's campus directory. **Before writing any code at all, your first question is: how will you represent any individual on campus?** We decide that the aspects we want our program to represent will be their name, phone, building, and room.

Let's think about what types of data we could use.

- Numbers?
  - Naw; I don't think of people as numbers 1, 2, 3, ..., so my program *should not* encode them like that either.
- Booleans?
  - Right out: (If nothing else, it's impossible by a counting argument: you can't represent thousands of people with only two possible values!)
- Symbols?
  - That's certainly possible, to have 'WalidTaha, 'JohnGreiner, 'DavidLeebron. It would mean we'd hand-craft functions such as;

    ```
    ; person->phone: symbol -> number
    ; Given a person's name, return their phone number.
    ;
    (define (person->phone a-name)
      ...)
    ```

    Similarly, you'd have functions to map from a name to a room, etc.

    But, upon reflection, it doesn't really capture how think about everyone on campus: Dr. Leebron is one person, which captures many different attributes. By not reflecting our thinking, this approach is also hard to maintain: next year when new students arrive, you'd have to encode the new info by changing several functions in several different places. Bugs are introduced if you forget any of those places. Keep in mind that in large programs, these functions might be spread over multiple files and maintained by different programmers. It's very easy to miss a bug!
- Structures!
  - We saw in class, we can make a new type of data, composed out of several smaller data. This reflects how we think about them; it also means that making a new person will be done all at once, rather than changing a bunch of functions.

Defining the Data:

```
;
; A person is:
;   (make-person symbol nat symbol int)
; where name is the person's name,
;       phone is their phone number (without hyphens and with area code),
;       building is the name of their on-campus office/college building,
; and   room is the number of their on-campus office/college room.
(define-struct person (name phone building room))
```

We're expanding the language by introducing a new data type. In addition to numbers, Booleans, and symbols, DrScheme now knows about persons! **We have not yet created any instances of this type** -- that is, we haven't yet defined Dr. Taha, etc.

Examples of the Data

Fortunately, the comments above, which explained the meaning of the different *fields* (or attributes), also shows us how to create particular instances of persons:

```
(make-person 'WalidTaha 7133485718 'DuncanHall 3103)
```

Or even better, let's give the whole structure a good name:

```
(define WalidTaha
   (make-person 'WalidTaha 7133485718 'DuncanHall 3103))
(define ThatOther210Instructor
   (make-person 'JohnGreiner 7133483838 'DuncanHall 3118))
```

Exercise: Creating structured data

Go ahead and create

```
person
```

s
representing yourself and somebody else.

The result of `make-person` is a particular structure. Think of it as a chest of drawers (the fields), with drawers labeled name, phone, etc. Within each drawer is the value of that field (`'WalidTaha, 7133485718,` etc).

Note that `make-person` is a function which was created automatically, after DrScheme evaluated your `define-struct` . `make-person` is called a *const ructor*.

Question

What is the contract for

```
make-person
```

?

In addition to the constructor `make-person` , DrScheme automatically creates a few other functions upon seeing your `(define-struct person (name ...))` . It creates four *selectors* (or *accessors*), one for each of the specified fields. One is

```
person-name : person -> symbol
```

Question

What are the other selectors and their contracts?

It also creates one predicate that tests whether something is a `person` . This will be useful in the second part of today's lab.

```
person? : any -> Boolean
```

Exercise: Using the predicate

Apply

```
person?
```

to various data, including the example

```
person
```

s you previously created and some other kinds of data.

Functions Using Structures

Note how much we've accomplished, and we *haven't yet even thought about what functions to write!*. Fortunately, once we have a clear data definition, writing the functions is often the easy part.

Exercises

1. Develop a function

   ```
   has-campus-phone?
   ```

   which takes in (our representation of) a person,
   and returns whether or not their listed phone number
   is a campus phone.
   All campus phone numbers look like 713-348-xxxx.

   If this exercise was no trouble,
   feel free to skip the next one.
2. Develop a function

   ```
   has-a-computer-science-office?
   ```

   .
   The computer science department offices are those on the
   3rd floor of Duncan Hall.
   (Well, that's not quite accurate, but it's close enough.)
3. Develop

   ```
   person=?
   ```

   , the equality function on
   the type. Note that Scheme doesn't create that for us.
   (There are some subtle reasons why it doesn't.
   We'll get into those reasons later in the course, but
   you might ponder about this a bit on your own.)
4. Develop a function

   ```
   move-person
   ```

   that
   takes in a person, building, and room.
   It returns a new person structure containing this new location.
   We'll assume you get to keep your old phone number when
   moving.

Combining Cases and Structures

Last lab we worked with user-defined types that were cases of atomic data. Now let's increase our complexity a bit and consider cases of compound data.

We've now been asked to revise our code above. Our boss has realized that, lo and behold, staff and students aren't alike. In particular, staff are paid a salary, and students can live off-campus. How should we model this?

We could use one kind of structure with this new information. For staff, we never use the off-campus information, while for students we never use the salary information. But that's not how we think about the problem! (Also, it tends to lead to bugs when we accidentally use the "unused" information.)

Instead, we want two different kinds of structures, each with their own information, but still consider both kinds to be

`person`

s.

```
;
; A person is one of:
; -  (make-staff symbol nat nat symbol int)
;   where name is the person's name,
;         salary is the person's salary (in dollars),
;         phone is their phone number (without hyphens and with area code),
;         building is the name of their office building,
;   and   room is the number of their office.
; -  (make-student symbol Boolean nat symbol int)
; where name is the person's name,
;       oc? is whether the person live off-campus or not,
;       phone is their phone number (without hyphens and with area code),
;       building is the name of their college ,
; and   room is the number of their on-campus college room.
;       (We'll assume all off-campus students have a designated
;        room to crash in, as is unofficially common.)
(define-struct staff (name salary phone building room))
(define-struct student (name oc? phone building room))
```

Any function on this type should have the following structure:

```
; person-func : person -> ...
(define (person-func a-person)
   (cond
      [(staff? a-person)
       (...(staff-name a-person)...
        ...(staff-salary a-person)...
        ...(staff-phone a-person)...
        ...(staff-building a-person)...
        ...(staff-room a-person)...)]
      [(student? a-person)
       (...(student-name a-person)...
        ...(student-oc? a-person)...
        ...(student-phone a-person)...
        ...(student-building a-person)...
        ...(student-room a-person)...)]))
```

This is the template for the person type. It states that any function with a person as input should, first, use the selectors to test which of the two cases is relevant, and second, for each case, use some or all of the selectors of that case. Within each case, the details will depend on the specific function we want to write, but writing out the selectors like this reminds us of the pieces we can use. **We can and should write the a type's template before writing any function on that type.** We can then copy-and-paste the template as a start in writing specific functions.

Exercises: Encore

Following the design recipe,
rewrite the following functions using
the new version of

```
person
```

.
Don't forget to create new example data!

1. Develop a function

   ```
   has-campus-phone?
   ```

   which takes in (our representation of) a person,
   and returns whether or not their listed phone number
   is a campus phone.
   All campus phone numbers look like 713-348-xxxx.
2. Develop a function

   ```
   has-a-computer-science-office?
   ```

   .
   The computer science department offices are those on the
   3rd floor of Duncan Hall.
   (Well, that's not quite accurate, but it's close enough.)
3. Develop a function

   ```
   move-person
   ```

   that
   takes in a person, building, and room.
   It returns a new person structure containing this new location.
   We'll assume you get to keep your old phone number when
   moving.

In your assignments, provide each template only once, regardless of how many functions may be based on that template. Copying the same template over and over should not only help your thinking, but also save time by reducing repetitive typing.

Stepping back: Recipe redux

What was involved in the preceding examples? We realized that when approaching a problem, the first thing we do, is decide how our program will represent data, in the problem we're modeling. Moreover, we've seen two common situations:

- The data is of the form "x, y, **or** z". Any particular function dealing with that data will use a cond, with one branch for each type of data.
- The data is of the form "x, y, **and** z". We use structures to encapsulate these three data into one conceptual datum. Any particular function dealing with that data will use the structure's selectors.

Our code mirrors our data, which in turn mirrors how we conceive of the problem. This is no coincidence. More than half the work of writing a good program lies in choosing good data definitions. We'll see how the shape of the data often guides you to a bug-free solution.

We'll close by updating our design recipe.

1. **Data definition:** How, exactly, do you represent the problem in terms of Scheme data?
2. **Data examples.**
3. *new* **Template.**
4. If your data has various cases, include a `cond` -- one branch per case of data.
   b. If your data is a structure, your code will be using the various selectors for that structure. You can write these down before thinking about this particular function, like a good craftsperson inspecting what tools are available to complete the task with, before launching in blindly.
5. **Function contract, purpose, header.**
6. **Function examples:** Use your previously-written data examples.
7. *revised* **Function body.** Copy-and-paste the template for the function's structure, and edit the details.
8. **Function testing:** Use your previously-written examples.