Module 8: Other Data Structures

Module 8, Section 1: Association lists and the need for Structures
Suppose I want to keep track of students by ID number, to store name and program. For each student I could make a list containing the data I want to store. I could then make a list, and put item $n$ in the $n$th position in my list, using our n-th-item function to get the $n$th student.

```
(define students
  (list (list "Al Gore" "government")
       (list "Barack Obama" "law")
       (list "Bill Gates" "appliedmath")
       (list "Conan O'Brien" "history")))
```

But then "Al Gore" has to be student 0, "Barack Obama" has to be student 1, "Bill Gates" has to be student 2.... I could put in some special value for ID numbers I don’t want to use. If I did this I would have millions of empty elements just to store thousands of student records!
A better way: make a list, where each item in the list is itself a list, containing two items: a key (ID number) and a value (information about students)

\[
\text{\textbf{A LStudent is a (list Str Str)}}
\]

\[
\text{\textbf{A LEntry is a (list Nat LStudent)}}
\]

\[
\text{\textbf{A LDict is a (listof LEntry)}}
\]

\[
\begin{align*}
\text{(define student-lldict} & \text{(list (list 6938 (list "Al Gore" "government"))}) \\
& \text{(list 7334 (list "Bill Gates" "appliedmath"))}) \\
& \text{(list 8535 (list "Conan O'Brien" "history"))}) \\
& \text{(list 8838 (list "Barack Obama" "law")))}
\end{align*}
\]

**Exercise**

Write a function (find-lldict key dict) that consumes a Nat and a LDict. The function returns the value in dict associated with the key. You may assume key appears exactly once in dict.

\[
\begin{align*}
\text{(check-expect (find-lldict 8838 student-lldict) (list "Barack Obama" "law"))}
\end{align*}
\]
Naming fields with Structures

Consider the data definition:

```plaintext
;; A LStudent is a (list Str Str)
;; A LEntry is a (list Nat LStudent)
```

Now we need to remember that if \( v \) is a LEntry, \((\text{first } v)\) returns the key, and \((\text{second } v)\) returns the associated value.

We also need to remember that if \( s \) is a LStudent:

- \((\text{first } s)\) returns the name
- \((\text{second } s)\) returns the program.

This is true, but inconvenient.

To help keep track of our data, we will now introduce structures, a new data type where the different parts have names.
It often comes up that we wish to join several pieces of data to form a single “package”. We can then write function that consume and return such packages.

A few examples:

A complex number

\[ z = a + bi \]

is built of a real part \( a \) and an imaginary part \( bi \).

A record in a student database might include the student’s name, ID number, and program.

```plaintext
{ 
  name: "James Bond"
  ID: 007
  program: "pure-math"
}
```

A labelled rooted binary tree has a label, left-child and right-child.
A Posn (short for Position) is a built-in structure that has two fields containing numbers intended to represent \( x \) and \( y \) coordinates. The computer knows these are called \( x \) and \( y \).

You can create a Posn using the constructor function, \texttt{make-posn}. Its contract is

\[
\text{;; make-posn: Num Num \rightarrow Posn}
\]

For example,

\[
\text{(define my-posn (make-posn 4 3))}
\]

Note here we are storing \textit{two things}, namely the \( x \) and \( y \) coordinates, in \textit{one} value. This one value is a Posn.
A Built-in structure: Posn

If you ask for the value of a Posn, it appears to just copy whatever you said.

```
(define my-posn (make-posn 4 3))
my-posn => (make-posn 4 3)
```

This is just like the quotation marks on a Str:

```
(define my-str "foo")
my-str => "foo"
```

**Exercise**

Create a constant somewhere that stores a Posn where the coordinates are (7, 2).

```
somewhere => (make-posn 7 2)
```
Selectors

With a Str, we have special functions which get a part of the value:

\[(\text{substring } "\text{foobar}" \text{ 0 3}) \Rightarrow "\text{foo}"\]

In a somewhat similar way, with a Posn, we have two selector functions. Each selector returns the field which has the name of the selector:

\[(\text{posn-x (make-posn 4 3)) \Rightarrow 4} \]
\[(\text{posn-y (make-posn 4 3)) \Rightarrow 3} \]

Note: these selectors are called posn-x and posn-y because the value is a Posn, and the fields are named x and y. Every structure has only the fields which are defined on it.

Exercise

Use posn-x and posn-y on your constant somewhere.
Ensure you understand the result.
One last function: the **type predicate**.

- `(posn? 42) => #false`
- `(posn? "oak") => #false`
- `(posn? my-posn) => #true`

The type predicate returns **#true** if its argument is some object of that type.

**Ex.** Find at least two values for which `posn?` returns **#true**, and two for which it returns **#false**.
Any time we create a structure we should create a data definition, and a template that goes along with it.

A template is derived from a **data definition**. When we create a new form of data, create the template. Use the template in writing functions to consume that type of data.

**Data definition:**

```
(define-struct sstudent
  (name id program))

;; a SStudent is a
;; (make-sstudent Str Nat Str)
;; Requires:
;; name is the student’s name
;; id is 8 digits long
;; program is sometimes fun.
```

**Template:**

```
;; my-sstudent-template: SStudent -> Any
(define (my-sstudent-template s)
  ...\(\text{sstudent-name s}\)...\n  ...\(\text{sstudent-id s}\)...\n  ...\(\text{sstudent-program s}\)...))
```

The template lists all the selectors, but does nothing. To write a function, replace the dots with code, and remove unused selectors.
Distance between two Posn

\[
((\text{posn-x } p2), (\text{posn-y } p2))
\]

\[
((\text{posn-x } p1), (\text{posn-y } p1))
\]

\[
;; (\text{distance } p1 \ p2) \text{ return the distance between } p1 \text{ and } p2.
\]

\[
;; \text{Posn Posn } \rightarrow \text{Num}
\]

\[
\text{Examples:}
\]

\[
(\text{check-expect \ (distance} \ (\text{make-posn 0 7)} \ (\text{make-posn 0 0)}) \ 7)
\]

\[
(\text{check-expect \ (distance} \ (\text{make-posn 5 6)} \ (\text{make-posn 2 2)}) \ 5)
\]

The template for the built-in Posn structure is as follows:

\[
;; \text{my-posn-template: Posn } \rightarrow \text{Any}
\]

\[
(\text{define \ (my-posn-template} \ p) \n\quad (\ldots(\text{posn-x} \ p)\ldots \n\quad \ldots(\text{posn-y} \ p)\ldots))
\]

Using the template, complete the function distance.
Returning a Posn

A function may return a \texttt{Posn} just like any other value. It needs to create it using \texttt{make-posn}.

\begin{verbatim}
;; (offset-a-little x y) return the point which is
;; moved over 3 and up 3 from (x, y).
;; offset-a-little: Num Num -> Posn
;; Example:
(check-expect (offset-a-little 5 7) (make-posn 8 10))

(define (offset-a-little x y)
  (make-posn (+ x 3) (+ y 3)))
\end{verbatim}

Exercise

Write a function \texttt{vector2D+} that consumes two \texttt{Posn} and does \textit{vector addition}. (That is, the new \texttt{x} is the sum of the \texttt{x} values, and the new \texttt{y} is the sum of the \texttt{y} values.)

\begin{verbatim}
;; (vector2D+ v1 v2) return the vector sum of v1 and v2.
;; vector2D+: Posn Posn -> Posn
;; Example:
(check-expect (vector2D+ (make-posn 2 3) (make-posn 5 8)) (make-posn 7 11))
\end{verbatim}
Custom structures

We can define a custom structure using the `define-struct` special form:

```
(define-struct struct-name (field0 field1 ... fieldn))
```

For example, suppose we are building a store inventory system, and for each item we need to store its description, price, and number available.

```
(define-struct inventory (desc price available))
```

This one line of code automatically creates several functions:

- **Constructor** `make-inventory` allows us to create values of this type
- **Predicate** `inventory?` lets us determine if a value is of this type
- **Selectors** are created, one for each field.
  - In this example, `inventory-desc`, `inventory-price` and `inventory-available`.
Creating and reading from custom structures

Once we have created the new data type:

```scheme
(define-struct inventory (desc price available))
```

...we can create a new `Inventory` value using the constructor, and store it in a constant:

```scheme
(define lentils (make-inventory "dry lentils" 2.49 42))
```

...and we can extract the values using the selector functions:

- `inventory-desc lentils` ; => "dry lentils"
- `inventory-price lentils` ; => 2.49
- `inventory-available lentils` ; => 42

Exercise

1. Create a structure data type called `book`, with fields `title`, `author`, and `year`.
2. Use the constructor to create a constant of this type.
3. Use the selector functions to extract the individual values from the constant.
(define-struct inventory (desc price available))

This `define-struct` determines the names of the fields, but it does not tell us what the fields mean. So we need to document these; this is done in a comment called a data definition:

;;; an Inventory is a (make-inventory Str Num Nat)
;;; Requires:
;;; desc describes what the item is
;;; price is the cost in dollars of one item
;;; available is the number of items in stock

The data definition tells us:

- the **type** of each field, in a line resembling a contract.
- the **meaning** of each field, in a Requires section.
Consuming custom structures

(define-struct inventory (desc price available))
;; an Inventory is a (make-inventory Str Num Nat)
;; Requires:
;;  desc describes what the item is
;;  price is the cost in dollars of one item
;;  available is the number of items in stock

Reminder: from this structure, if thing is an Inventory, you can access the fields using
(inventory-desc thing), (inventory-price thing) and (inventory-available thing).

Exercise

Complete the function total-value that consumes an Inventory and returns the amount of
money we would get if we sell out of item.

;; (total-value item) return cost of all our item.
;; total-value: Inventory -> Num
;; Example:
(check-expect (total-value (make-inventory "rice" 5.50 6)) 33.00)
Returning custom structures

(define-struct inventory (desc price available))

;; an Inventory is a (make-inventory Str Num Nat)
;; Requires:
;;   desc describes what the item is
;;   price is the cost in dollars of one item
;;   available is the number of items in stock

Reminder: from this structure, if thing is an Inventory, you can access the fields using (inventory-desc thing), (inventory-price thing) and (inventory-available thing).

To create an Inventory, use the make-inventory function.

Write a function (raise-price dollars item) that consumes a Num and a Inventory and returns the Inventory that results from increasing the price of item by dollars.

;; (raise-price dollars item) return item with price increased by dollars.
;; raise-price: Num Inventory -> Inventory
;; Example:
(check-expect (raise-price 0.49 (make-inventory "rice" 5.50 6))
 (make-inventory "rice" 5.99 6))
What is the result of evaluating the following expression?

```
(define (distance a b)
  (sqrt (+ (- (posn-x a) (posn-x b))
           (- (posn-y a) (posn-y b))))
)
```

```
(define pt1 (make-posn "Math135" "CS115"))
(define pt2 (make-posn "Red" #true))

(distance pt1 pt2)
```

This causes a run-time error, but not at `make-posn`.

Inside `distance`, there it attempts to compute `(- "Math 135" "Red")`, which is nonsense. The system does not enforce contracts. If your contract says `Int`, but you give it a `Str`, problems will probably occur.

Carefully watch your contracts, and be sure your code follows them!
Contract Errors

;;; (scale pt factor) return pt
;;; scaled by factor.
;;; scale: Posn Num -> Posn

(define (scale pt factor)
  (make-posn (* factor (posn-x pt))
              (* factor (posn-y pt))))

(scale 2 "George")
=>
(make-posn (* "George" (posn-x 2))
            (* "George" (posn-y 2)))

Contract errors will often manifest when we can’t simplify an expression. In this case, we can’t use posn-x on 2.
Suppose I have a complicated structure:

```scheme
(define-struct household (me sally fish cat thing1 thing2))
;; a Household is is a (make-household Nat Nat Nat Nat Nat Nat)
;; Requires:
;; me is my age
;; sally is Sally's age
;; cat is the cat's age, etc.
```

and I want a structure with one field changed. Do I really have to do all this work?!?

```scheme
;; update-cat: Household -> Household
(define (update-cat house newcat)
  (make-household
   (household-me house)
   (household-sally house)
   (household-fish house)
   newcat
   (household-thing1 house)
   (household-thing2 house)))
```

...Yes. Racket structures, as defined in this course, are clumsy. But don’t get put off structures! They are very useful, and much easier to use in every other language I know. In many languages you would just say house.cat = newcat
We have added two new things to our syntax.

1. The special form `(define-struct sname (field1 ... fieldn))` defines the structure type and creates:
   - a `constructor` function `make-sname`
   - a `predicate` function `sname?`
   - `n` selectors, one for each field, named `sname-field1...`

2. A value has additional possibilities. In addition to begin a `Nat`, `Int`, `Num`, `Str`, `Bool`, `(list ...), or (listof ...), it may be of the form:
   `(make-sname v1...vn)`
Additions to the Design Recipe for Structures

1. Place your **structure definitions** and **data definitions** right at the top of the file, just after the file header.

```scheme
(define-struct inventory (desc price available))
;; an Inventory is a (make-inventory Str Num Nat)
;; Requires:
;; desc describes what the item is
;; price is the cost in dollars of one item
;; available is the number of items in stock
```

2. Write a **template**, with a generic name and generic contract.

```scheme
(define (my-inventory-template item)
  (...(inventory-desc item)...
  ... (inventory-price item)...
  ... (inventory-available item) ...))
```

The rest of the design recipe is essentially unchanged, except now you have the custom type (e.g. **Inventory**) which you added.
Back on slide we used the following data definitions:

```scheme
;; A LStudent is a (list Str Str)
;; A LEntry is a (list Nat LStudent)
;; A LDict is a (listof LEntry)

(define student-lldict
  (list (list 6938 (list "Al Gore" "government"))
        (list 7334 (list "Bill Gates" "appliedmath"))
        (list 8535 (list "Conan O'Brien" "history"))
        (list 8838 (list "Barack Obama" "law"))))
```

This made it hard to keep track of which bits of data were which.

For example, I can get the first student from student-dict using (first student-dict). But why should I get that student’s name with (first (second (first student-dict)))?

This is a perfect place to use structures.
We can use a structure to store each student. Then to store an association list of students, we have:

```scheme
(define-struct student (name program))
;; a Student is a (make-student Str Str)
;; A LDict is a (listof (list Nat Student))
(define student-ldict
  (list (list 6938 (make-student "Al Gore" "government"))
        (list 7334 (make-student "Bill Gates" "appliedmath"))
        (list 8535 (make-student "Conan O'Brien" "history"))
        (list 8838 (make-student "Barack Obama" "law"))))
```

Suppose we have some student:

```scheme
(define test-student (first student-dict))
```

- `(first test-student) => 6938`
- `(second test-student) => (make-student "Al Gore" "government")`

...It works, but we still need to use `first` and `second` to get the key and value.
We can take it another level. Use another structure to keep track of keys and values.

```
(define-struct asc (key val))
;; An Asc is a (make-asc Any Any)
;; a Dict (dictionary) is a (listof Asc)
```

```
(define student-dict
  (list
    (make-asc 6938 (make-student "Al Gore" "government"))
    (make-asc 7334 (make-student "Bill Gates" "appliedmath"))
    (make-asc 8838 (make-student "Barack Obama" "law"))))
```

Extracting some test data from this list:
```
(define test-student (first student-dict))
```
```
(define test-student (first student-dict))
(asc-key test-student) => 6938
(asc-val test-student) => (make-student "Al Gore" "government")
Dictionaries: retrieval

(define-struct asc (key val))
;; An Asc is a (make-asc Any Any)
;; a Dict (dictionary) is a (listof Asc)

(define student-dict
  (list (make-asc 6938 (make-student "Al Gore" "government"))
        (make-asc 7334 (make-student "Bill Gates" "appliedmath"))
        (make-asc 8838 (make-student "Barack Obama" "law")))))

Reminder: if a is an Asc:
  (asc-key a) returns the key
  (asc-value a) returns the associated value.

---

Exercise

Complete dict-find. You may assume key appears at most once in dict.

;; (dict-find d key) return value associated with key in d.
;; If key is not in d, return #false.
;; dict-find: Dict Nat -> Any
;; Examples:
(check-expect (dict-find student-dict 7334)
  (make-student "Bill Gates" "appliedmath"))
(check-expect (dict-find student-dict 9999) #false)
Complete dict-add.

;;; (dict-add d k v) return a new dictionary containing all values in d,
;;; and new value (make-asc k v). Keep data sorted by key.
;;; If key is already in d, replace its value.
;;; dict-add: Dict Nat Any -> Dict
;;; Example:
(check-expect
 (dict-add student-dict 7587
  (make-student "George W Bush" "business"))
(list (make-asc 6938 (make-student "Al Gore" "government"))
 (make-asc 7334 (make-student "Bill Gates" "appliedmath"))
 (make-asc 7587 (make-student "George W Bush" "business"))
 (make-asc 8838 (make-student "Barack Obama" "law"))))
Module Summary

- Be comfortable with the following terms: structure, field, constructor, selector, type predicate, structure definition.
- Be able to write functions that consume and return structures, include Posn and custom data structures.
- Be able to create structure and data definitions for a new structure, determining an appropriate type for each field.
- Understand what functions are created by define-struct, and be able to use them.
- Be able to write the template associated with a structure definition, and to expand it into the body of a particular function that consumes that type of structure.
- Understand the use of type predicates and be able to write code that handles mixed data.

Further Reading: How to Design Programs Section 5