Structures

Readings: None

Topics:

- Introducing compound data
- Formalities: Syntax and semantics; templates
- Example
- Mixed data
- Lists vs. structures
- Quote notation

Compound data

We have used short, fixed-length, lists for data that seems to always belong together. For example, in M08 we had a "payroll" with names and salaries:

```
(list (list "Asha" 50000)
      (list "Joseph" 100000)
      (list "Sami" 10000))
```

A name and salary always go together in this application.

Other kinds of data that always go together include:

- Student (name, program, courses)
- Point (x coordinate, y cooridinate)
- Book (author, title, number of pages)

> Example: Student

We could represent a student with a short list containing their name, program, and a list of courses.

If we were to use such a student list often, we might want to put more care into it:

- Some helper functions to extract the name, program, and courses
- A predicate to see if a given value represented a student
- Error messages if we gave it another kind of list

> Example: Student (1/3)

```
;; A Std (student) is a (make-std Str Str (listof Str))


;; A large "random" value to check for legit student values
(define STD-TAG "std_391249569284455218")

;; (make-std name prog classes) makes a new student structure
;;    containing the name, program and classes for the student.
;; make-std: Str Str (listof Str) → Std
(define (make-std name prog classes)
  (list STD-TAG name prog classes))

;; A sample student for testing
(define Juan (make-std "Juan" "CS" (list "CS 135" "MATH 137")))
```


> Example: Student (2/3)

```
;; (std? v) returns true if v is a Std and false otherwise.
;; std?:  Any → Bool
(check-expect (std? Juan) true)
(check-expect (std? (list "Juan" "CS" (list "CS 135" "MATH 137"))) false)
(check-expect (std? "Juan") false)

(define (std? s)
  (and (cons? s)
       (= (length s) 4)
       (string=? (first s) STD-TAG)))
```


> Example: Student (3/3)

```
;; (std-name s) extracts the name field from student s; error
;;    if s is not a student
;; std-name: Std → Str
(check-expect (std-name Juan) "Juan")
(check-error (std-name (list "Juan"))
             "std-name: expects a std, given (list \"Juan\")")

(define (std-name s)
  (cond [(std? s) (second s)]
        [else (error "std-name: expects a std, given " s)]))
```

`std-prog` and `std-classes` are nearly identical to `std-name`.

Racket support for structures

A Racket **structure definition** creates all of the above in only one line:

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))
```

Great News!

define-struct is a special form that (given the line above) automatically creates functions identical to the functions on the previous slides.

The second line is the structure's **data definition**. Whenever you use **define-struct**, add a data definition to give the expected types.

Given the data definition, Std may be used in contracts.

Functions created:

- make-std
- std?
- std-name
- std-prog
- std-classes

> Example: add-class

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))

;; (add-class s class) adds a new class to the student s.
;; add-class: Std Str → Std
(check-expect (add-class (make-std "Jo" "CS" (list "MATH 137")) "CS 135")
              (make-std "Jo" "CS" (list "CS 135" "MATH 137")))

(define (add-class s class)
  (make-std (std-name s)
            (std-prog s)
            (cons class (std-classes s))))
```

(make-std n p c) is considered a value (as long as n, p, and c are values) and will not be simplified.

Exercise 1

1. Create a structure 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.

Syntax and semantics

The special form
`(define-struct sname (fname_1 ... fname_n))`
defines the structure type `sname` with **fields** `fname_1` to `fname_n`. It also automatically defines the following primitive functions:

- **Constructor:** `make-sname`
- **Selectors:** `sname-fname_1 ... sname-fname_n`
- **Predicate:** `sname?`

`Sname` (note the capitalization) may be used in contracts.

Substitution rules

`(make-sname v_1 ... v_n)` is a value.

The substitution rule for the *i*th selector is:

`(sname-fname_i (make-sname v_1 ... v_i ... v_n))` ⇒ `v_i`.

Finally, the substitution rules for the new predicate are:

`(sname? (make-sname v_1 ... v_n))` ⇒ `true`
`(sname? V)` ⇒ `false` for `V` a value of any other type.

Structure templates

The template function for a structure simply selects all its fields, in the same order as listed in the `define-struct`. For example,

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))

;; std-template: Std → Any
(define (std-template s)
  (    ... (std-name s)
       ... (std-prog s)
       ... (std-classes s) ... ))
```

The above (structure definition, data definition, and template function) are only required *once per file*.

Example: Classlists

Define a “class list” that contains students enrolled in a course.

Develop functions that:

- Produce the names of the students in the class list.
- Add a new student to the classlist, preserving alphabetical order.
- Verify that all the students in a classlist have the class in their list of classes.

> Classlists (1/4)

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))

;; std-template: Std → Any
(define (std-template s)
  (    ... (std-name s) ... (std-prog s) ... (std-classes s)))

;; A Classlist is a (listof Std)

;; Sample students for testing
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))
```

These definitions are only done **once**, no matter how many functions use them.

> Classlists (2/4)

```
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))


;; (class-names clst) produces a list of the student names in clst.
;; class-names: Classlist → (listof Str)
(check-expect (class-names (list aj jo di))
              (list "AJ" "Jo" "Di"))

(define (class-names clst)
  (cond [(empty? clst) empty]
        [(cons? clst) (cons (std-name (first clst))
                            (class-names (rest clst)))]))
```

> Classlists (3/4)

```
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))

;; (add-std s clst) produces a new classlist compose
;;   all the students in clst.  Maintain alphabetica
;; add-std: Std Classlist → Classlist
;;   reqires: the classlist is in alphabetical order
(check-expect (add-std di (list aj jo))
              (list aj di jo))

(define (add-std s clst)
  (cond [(empty? clst) (list s)]
        [(string<? (std-name s) (std-name (first clst))) (cons s clst)]
        [else (cons (first clst) (add-std s (rest clst)))]))
```

No need to repeat the structure and data definitions.

> Classlists (4/4)

```
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))

;; (all-enrolled? class clst) produces true iff each student in clst has
;; class in their list of classes
;; all-enrolled?:  Str Classlist → Bool
(check-expect (all-enrolled? "CS 135" (list aj jo di)) true)
(check-expect (all-enrolled? "MATH 137" (list aj jo di)) false)

(define (all-enrolled? class clst)
  (cond [(empty? clst) true]
        [else (and (member? class (std-classes (first clst)))
                   (all-enrolled? class (rest clst)))]))
```


Exercise 2

Given the following data definition,

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

Exercise 3

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`.

Mixed data

Racket provides predicates such as `number?` and `symbol?` to identify data types.

`define-struct` also produces a predicate that tests whether its argument is that type of structure (e.g. `std?`).

We can use these to check aspects of contracts and to write functions that consume **mixed data** – data of several (probably related) types.

Example: A university has (undergraduate) students as well as graduate students. Graduate students are like other students except that they also have a supervisor. Some courses may have both kinds of students.

> Data definitions

```
(define-struct ustd (name prog classes))
;; A UStd (undergraduate student) is a (make-ustd Str Str (listof Str))

(define-struct gstd (name prog supervisor classes))
;; A GStd (graduate student) is a (make-gstd Str Str Str (listof Str))

;; A Student is one of:
;; * a UStd
;; * a GStd

;; A Classlist is a (listof Student)
```

There is no structure definition for mixed data. There is, however, a data definition that describes the data and gives a name that can be used in contracts.

> Template function

The template function for mixed data will determine the type of the data and then include a template for that type.

```
(define (student-template s)
  (cond [(ustd? s) (... (ustd-name s)...
                        (ustd-prog s) ...
                        (ustd-classes s)...)]
        [(gstd? s) (... (gstd-name s)...
                        (gstd-prog s)...
                        (gstd-supervisor s)...
                        (gstd-classes s)...)]))
```


> Example: Update program

```
;; (update-prog std prog) updates the student's program ...
;; update-prog:  Student Str → Student
(check-expect (update-prog (make-ustd "Jo" "Math" empty) "CS")
              (make-ustd "Jo" "CS" empty))
(check-expect (update-prog (make-gstd "Di" "CS" "Ian" empty) "Arts")
              (make-gstd "Di" "Arts" "Ian" empty))

(define (update-prog std prog)
  (cond [(ustd? std) (make-ustd (ustd-name std)
                                prog
                                (ustd-classes std))]
        [(gstd? std) (make-gstd (gstd-name std)
                                prog
                                (gstd-supervisor std)
                                (gstd-classes std))]))
```


> Example: Filter by program

```
;; (filter-prog prog cl) produces a classlist consisting of only
;;    the students in cl who are in the program prog.
;; filter-prog: Str Classlist → Classlist
(define (filter-prog prog cl)
  (cond [(empty? cl) empty]
        [(in-prog? prog (first cl))
         (cons (first cl) (filter-prog prog (rest cl)))]
        [else (filter-prog prog (rest cl))]))

;; (in-prog? prog s) produces true iff student s is in program prog.
(define (in-prog? prog s)
  (string=? prog (cond [(ustd? s) (ustd-prog s)]
                       [(gstd? s) (gstd-prog s)])))
```

anyof types

Unlike UStd and GStd, the Student and Classlist types do **not** have a structure definition (i.e. **define-struct**).

For a contracts like

```
;; update-prog:  Student Str → Student
```

and

```
;; filter-prog: Str Classlist → Classlist
```

to make sense, we need to have the data definitions for Student and Classlist included as a comment in the program.

An alternative to Student would be to use

```
;; update-prog:  (anyof UStd GStd) Str → (anyof UStd GStd)
```


Checked functions

Constructor functions do not check that their arguments have the correct type. We can use type predicates to make a type-safe version.

```
(define-struct ustd (name prog classes))
;; A UStd (undergraduate student) is a (make-ustd Str Str (listof Str))

(define (safe-make-ustd name prog classes)
  (cond [(and (string? name) (> (string-length name) 0)
              (string? prog) (> (string-length prog) 0)
              (list? classes)) (make-ustd name prog classes)]
        [else (error "Invalid argument types")]))

(check-error (safe-make-ustd "Jo" 123 empty) "Invalid argument types")
(check-error (safe-make-ustd "Jo" "CS" 'Sym) "Invalid argument types")
(check-expect (safe-make-ustd "J" "C" empty) (make-ustd "J" "C" empty))
```


Structures vs. lists

We don’t have to use structures. We could construct a class list with simple lists:

```
(define cs135/s (list
                 (make-ustd "AJ" "CS" (list "CS 488" "CS 449"))
                 (make-gstd "Jo" "CS" "Ian" (list "CS 688" "CS 749"))
                 (make-ustd "Di" "Math" (list "CS 488" "PMATH 330"))))

(define cs135/l (list
                 (list "AJ" "CS" (list "CS 488" "CS 449"))
                 (list "Jo" "CS" "Ian" (list "CS 688" "CS 749"))
                 (list "Di" "Math" (list "CS 488" "PMATH 330"))))
```

What are the advantages and disadvantages?

Structures vs. lists

Structures:

- help avoid some programming errors (e.g. extracting the wrong field)
- provide meaningful names that are easier to read and understand.
- automatically generate significant code.

Lists:

- make it possible to write "generic" functions that operate on several types of data (e.g. `(first s)` will extract the name for both undergraduates and graduates; with structures you need to use **`cond`** first).
- can be expressed more compactly than structures.

Quoting

The previous slide mentioned expressing lists more compactly. In the next module we'll have good use for both structures and a compact notation for lists: **quote notation**.

"**Quoting**" is an extension of how we expressed symbols.

`(cons 'red (cons 'blue (cons 'green empty)))` and `(list 'red 'blue 'green)` can be written as
`'(red blue green)`.

Quoted lists can be nested:
`'(red (blue green))` is the same as `(list 'red (list 'blue 'green))`.

Quoting

Strings and numbers can be used in quoted lists because quoted numbers evaluate to numbers and quoted strings evaluate to strings. That is `'5 => 5` and `'"Hello!" => "Hello!"`.

Therefore, `(list 5 4 3 2)` can be written `'(5 4 3 2)`.

What is `'()` ?

Exercise 4

Complete `count-sheep`.

```
;; (count-sheep L) return the number of 'sheep in L.
;; count-sheep: (listof Any) → Nat
;; Example:
(check-expect (count-sheep '(6 sheep ram 3.14 sheep ox)) 2)
```

Goals of this module

- You should be able to write code to define a structure and to use the functions that are defined when you do so.
- You should understand the data definitions we have used and be able to write your own.
- You should be able to write a structure definition's template and to expand it into the body of a particular function that consumes that type of structure.
- You should understand the use of type predicates and be able to use them to work with mixed data.
- You should understand the similar uses of structures and fixed-size lists and be able to write functions that consume either type of data.
- You should be able to convert back and forth between lists built with `cons`, `list`, and quote notation.