Module 8: Compound data: structures

Readings: Sections 6 and 7 of HtDP.

- Sections 6.2, 6.6, 6.7, 7.4 are optional readings; they use the obsolete `draw.ss` teachpack.
- The teachpacks `image.ss` and `world.ss` are more useful.
- Note that none of these particular teachpacks will be used on assignments or exams.

Recall types defined using fixed-length lists

In Module 05, we defined a new type Student using a list of length 4, and stored information about a student's name, assignment grade, as well as midterm and final exam grades.

We also defined a function to construct a Student.

```
;; A Student is a (list Str Num Num Num)
;; requires: All numbers are the range 0 to 100, inclusive.

;; make-student: Str Num Num Num Num → Student
(define (make-student name assts mid final)
    (list name assts mid final))
```

We defined functions that consume a Student and select the values of the individual fields:

```
(define (student-name s) (first s))
(define (student-assts s) (second s))
(define (student-mid s) (third s))
(define (student-final s) (fourth s))
```

We also developed a template for a Student, which is a partial function for the general shape of a function that consumes a Student.

```
;; student-template: Student → Any
(define (student-template s)
    (. . . (student-name s) . . . (student-assts s)
     . . . (student-mid s) . . . (student-final s) . . . ))
```

In this module, we will introduce a different way to define new types in Racket which is generally more readable than lists: structures.

Compound data

A **structure** is an alternate way of “bundling” several pieces of data together to form a single “package”.

As with our fixed-length list types, we can

- create functions that consume and/or produce structures, and
- define our own structures, automatically getting (“for free”) functions that create structures and functions that extract data from structures.

A new version of Student

We will use a structure definition in combination with a data definition when defining a new structure type.

```
(define-struct student (name assts mid final))
;; A Student is a (make-student Str Num Num Num)
;; requires:
;; * 0 ≤ assts, mid, final ≤ 100
```

As a result of the define-struct definition, Racket provides the following:

- a constructor function make-student
- selector functions student-name, student-assts, student-mid, student-final
- a type predicate student? that consumes Any and produces true only for values created using make-student

Again, unlike our fixed length list types, we do **not** have to define these additional functions - Racket does it for us!

Using the Student structure

```
(define alan (make-student "Alan Turing" 75 80 91))

(student-name alan) ⇒ "Alan Turing"

(+ (student-mid alan) (student-final alan)) ⇒ 171

(student? alan) ⇒ true

(student? (list "A" 100 100 100)) ⇒ false
```

We can write functions that consume a Student or a (listof Student).

We should still develop a template for functions that consume Student values. In this case, our previous template still works.

```
;; exams-passed: Student → Bool
(define (exams-passed? s)
   (and (>= (student-mid s) 50) (>= (student-final s) 50)))

;; name-list: (listof Student) → (listof Str)
(define (name-list sl)
  (cond [(empty? sl) empty]
        [else (cons (student-name (first sl)) (name-list (rest sl)))]))
```

A new type

Suppose we want to design a program for a card game such as poker or cribbage. Before writing any functions, we have to decide on how to represent data.

For each card, we have a suit (one of hearts, diamonds, spades, and clubs) and a rank (for simplicity, we will consider ranks as integers between 1 and 13). We can create a new structure with two fields using the following **structure definition**.

```
(define-struct card (rank suit))
```

Using the Card type

Once we have defined our new type, we can:

- Create new values using the **constructor** function make-card

```
(define h5 (make-card 5 'hearts))
```

- Retrieve values of the individual fields using the **selector** functions card-rank and card-suit

(card-rank h5) $\Rightarrow$ 5

(card-suit h5) $\Rightarrow$ 'hearts

We can also

- Check if a value is of type Card using the **type predicate** function card?

(card? h5) $\Rightarrow$ true

(card? "5 of hearts") $\Rightarrow$ false

Once the new structure card has been defined, the functions make-card, card-rank, card-suit, card? are created by Racket. We do not have to write them ourselves.

We have grouped all the data for a single card into one value, and we can still retrieve the individual pieces of information.

More information about Card

The structure definition of Card does not provide all the information we need to use the new type properly. We will still need a **data definition** to provide additional information about the types of the different field values.

```
(define-struct card (rank suit))
;; A Card is a
;;     (make-card Nat (anyof 'hearts 'diamonds 'spades 'clubs))
;; requires
;;      rank is between 1 and 13, inclusive
```

Functions using Card values

```
;; (pair? c1 c2) produces true if c1 and c2 have the same rank,
;;    and false otherwise
;; pair?: Card Card → Bool
(define (pair? c1 c2) (= (card-rank c1) (card-rank c2)))

;; (one-better c) produces a Card, with the same suit as c, but
;;    whose rank is one more than c (to a maximum of 13)
;; one-better: Card → Card
(define (one-better c)
  (make-card (min 13 (+ 1 (card-rank c))) (card-suit c)))
```

Posn structures

A Posn (short for Position) is a built-in structure that has two **fields** containing numbers intended to represent x and y coordinates. We might want to use a Posn to represent coordinates of a point on a 2-D plane, positions on a screen, or a geographical position.

This structure definition is built-in. We'll use the following data definition.

```
;; A Posn is a (make-posn Num Num)
```

Built-in functions for Posn

```
;; make-posn: Num Num → Posn
;; posn-x: Posn → Num
;; posn-y: Posn → Num
;; posn?: Any → Bool
```

Examples of use

```
(define myposn (make-posn 8 1))
(posn-x myposn) ⇒ 8
(posn-y myposn) ⇒ 1
(posn? myposn) ⇒ true
```

Simplifying expressions

For any values a and b

(posn-x (make-posn a b)) $\Rightarrow$ a

(posn-y (make-posn a b)) $\Rightarrow$ b

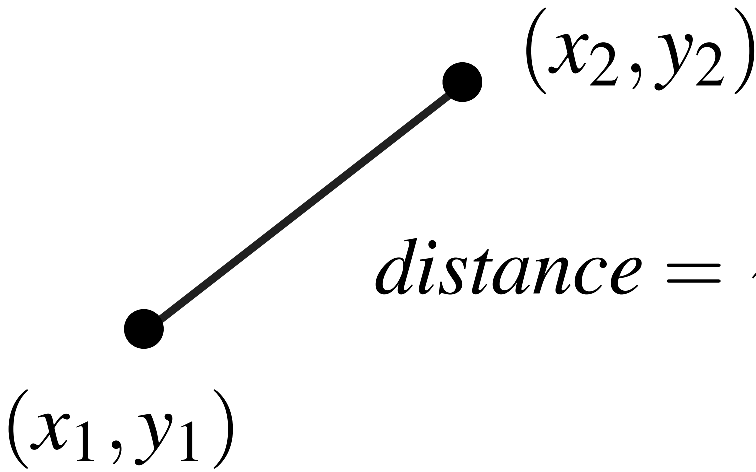
The make-posn you type is a function application.

The make-posn DrRacket displays indicates that the value is of type posn.

(make-posn (+ 4 4) (− 2 1)) yields (make-posn 8 1), which cannot be further simplified.

Similar behaviours apply to all structres we define.

Example: point-to-point distance



$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The function distance

```
;; (distance posn1 posn2) produces the Euclidean distance
;;    between posn1 and posn2.
;; distance: Posn Posn → Num
;; example:
(check-expect (distance (make-posn 1 1) (make-posn 4 5)) 5)

(define (distance posn1 posn2)
  (sqrt (+ (sqr (− (posn-x posn2) (posn-x posn1)))
           (sqr (− (posn-y posn2) (posn-y posn1))))))
```

Function that produces a Posn

```
;; (scale point factor) produces the Posn that results when the fields
;;    of point are multiplied by factor
;; scale: Posn Num → Posn
;; Examples:
(check-expect (scale (make-posn 3 4) 0.5) (make-posn 1.5 2))
(check-expect (scale (make-posn 1 2) 1) (make-posn 1 2))

(define (scale point factor)
  (make-posn (* factor (posn-x point))
             (* factor (posn-y point))))
```

When we have a function that consumes a number and produces a number, we do not change the number we consume.

Instead, we make a new number.

The function scale consumes a Posn and produces a new Posn.

It doesn't change the old one.

Instead, it uses make-posn to make a new Posn.

Structure definitions for new structures

Structure definition: code defining a structure, and resulting in constructor, selector, and type predicate functions.

```
(define-struct sname (field1 field2 field3))
```

Writing this once creates functions that can be used many times:

- **Constructor**: make-sname
- **Selectors**: sname-field1, sname-field2, sname-field3
- **Predicate**: sname?

Design recipe for compound data

Do this *once per new structure type:*

Data analysis and design: Define any new structures needed for the problem. Write structure and data definitions for each new type (include right after the file header).

Template: Create one template for each new type defined, and use for each function that consumes that type. Use a generic name for the template function and include a generic contract.

Do the usual design recipe steps for *every function:*

Purpose: Same as before.

Contract and requirements: Can use both built-in data types and defined structure names.

Examples: Same as before.

Function Definition: To write the body, expand the template based on the examples.

Tests: Same as before. Be sure to capture all cases.

Structure for Movie information

Suppose we want to represent information associated with movies, that is:

- the name of the director
- the title of the movie
- the duration of the movie
- the genre of the movie (sci-fi, drama, comedy, etc.)

```
(define-struct movieinfo (director title duration genre))
;; A MovieInfo is a (make-movieinfo Str Str Nat Sym)
;; where:
;;      duration is the length of the movie in minutes,
```

Note: We will omit the field-by-field descriptions when the field name explains the role of the field.

The structure definition gives us:

- Constructor make-movieinfo
- Selectors movieinfo-director, movieinfo-title, movieinfo-duration, and movieinfo-genre
- Predicate movieinfo?

```
(define m-movie
  (make-movieinfo "Lang" "M" 99 'Crime))
(movieinfo-duration m-movie) ⇒ 99
(movieinfo? 6) ⇒ false
```

A template for MovieInfo

The template for a function that consumes a structure selects every field in the structure, though a specific function may not use all the selectors.

```
;; movieinfo-template: MovieInfo → Any
(define (movieinfo-template info)
  (... (movieinfo-director info)...
       (movieinfo-title info)...
       (movieinfo-duration info)...
       (movieinfo-genre info)...))
```

An example

```
;; (correct-genre oldinfo newg) produces a new MovieInfo
;;    formed from oldinfo, correcting genre to newg.
;; correct-genre: MovieInfo Str → MovieInfo
;; example:
(check-expect
  (correct-genre
      (make-movieinfo "Lang" "M" 99 'Comedy)
        'Crime)
      (make-movieinfo "Lang" "M" 99 'Crime))
```

The function correct-genre

We use the parts of the template that we need, and add a new parameter.

```
(define (correct-genre oldinfo newg)
  (make-movieinfo (movieinfo-director oldinfo)
                  (movieinfo-title oldinfo)
                  (movieinfo-duration oldinfo)
                  newg))
```

We could have done this without a template, but the use of a template pays off when designing more complicated functions.

Additions to syntax for structures

The special form (define-struct sname (field1 . . . fieldn)) defines the structure type sname and automatically defines a constructor function, selector functions, and a type predicate.

Our definition of a **value** is now extended to include (make-sname v1 . . . vn) for values v1 through vn.

The selector and type predicate functions simplify as follows:

(sname-fieldi (make-sname v1 . . . vi . . . vn)) $\Rightarrow$ vi

(sname? (make-sname v1 . . . vn)) $\Rightarrow$ true

(sname? V) $\Rightarrow$ false for a value V of any other type.

An example using posns

Recall the definition of the function scale :

```
(define (scale point factor)
  (make-posn (∗ factor (posn-x point))
             (∗ factor (posn-y point))))
```

Then we can make the following substitutions:

```
(define myposn (make-posn 4 2))
(scale myposn 0.5)
⇒ (scale (make-posn 4 2) 0.5)
⇒ (make-posn
    (∗ 0.5 (posn-x (make-posn 4 2)))
    (∗ 0.5 (posn-y (make-posn 4 2))))
⇒ (make-posn
    (∗ 0.5 4)
    (∗ 0.5 (posn-y (make-posn 4 2))))
```

$\Rightarrow$ (make-posn 2 (∗ 0.5 (posn-y (make-posn 4 2))))

$\Rightarrow$ (make-posn 2 (∗ 0.5 2))

$\Rightarrow$ (make-posn 2 1)

Since (make-posn 2 1) is a value, no further substitutions are needed.

Another example

```
(define mymovie (make-movieinfo "Reiner" "Princess Bride" 98 'War))

(correct-genre mymovie 'Fantasy)

⇒ (correct-genre

(make-movieinfo "Reiner" "Princess Bride" 98 'War) 'Fantasy)

⇒ (make-movieinfo

(movieinfo-director (make-movieinfo "Reiner" "Princess Bride" 98 'War))

(movieinfo-title (make-movieinfo "Reiner" "Princess Bride" 98 'War))

(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 'War))

'Fantasy)
```

```
⇒ (make-movieinfo
"Reiner"
(movieinfo-title (make-movieinfo "Reiner" "Princess Bride" 98 'War))
(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 'War))
'Fantasy)
⇒ (make-movieinfo
"Reiner"
"Princess Bride"
(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 'War))
'Fantasy)
⇒ (make-movieinfo "Reiner" "Princess Bride" 98 'Fantasy)
```

Design recipe example

Suppose we wish to create a function total-length that consumes information about a TV series, and produces the total length (in minutes) of all episodes of the series.

Data analysis and design.

```
(define-struct tvseries (title eps len-per))
;; A TVSeries is a (make-tvseries Str Nat Nat)
;; where
;;       title is the name of the series
;;       eps is the total number of episodes
;;       len-per is the average length (in minutes) for one episode
```

The structure definition gives us:

- Constructor make-tvseries
- Selectors tvseries-title, tvseries-eps, and tvseries-len-per
- Predicate tvseries?

The data definition tells us:

- types required by make-tvseries
- types produced by tvseries-title, tvseries-eps, and tvseries-len-per

Templates for TVSeries

We can form a template for use in any function that consumes a single TVSeries:

```
;; tvseries-template: TVSeries → Any
(define (tvseries-template show)
  (... (tvseries-title show) ...
       (tvseries-eps show) ...
       (tvseries-len-per show) ...))
```

You might find it convenient to use constant definitions to create some data for use in examples and tests.

```
(define murdoch (make-tvseries "Murdoch Mysteries" 168 42))
(define friends (make-tvseries "Friends" 236 22))
(define fawlty (make-tvseries "Fawlty Towers" 12 30))
```

Mixed data and structures

Consider writing functions that use a streaming video file (movie or tv series), which does not require any new structure definitions.

```
(define-struct movieinfo (director title duration genre))
;; A MovieInfo is a (make-movieinfo Str Nat Str)
(define-struct tvseries (title eps len-per))
;; A TVSeries is a (make-tvseries Str Nat Nat)

;; A StreamingVideo is one of:
;; * a MovieInfo or
;; * a TVSeries.
```

The template for StreamingVideo

```
;; StreamingVideo-template: StreamingVideo → Any
(define (streamingvideo-template info)
  (cond [(movieinfo? info)
         (. . . (movieinfo-director info) . . .
                (movieinfo-title info) . . .
                (movieinfo-duration info) . . .
                (movieinfo-genre info) . . . ) ]
         [else
         (. . . (tvseries-title info) . . .
                (tvseries-eps info) . . .
                (tvseries-len-per info) . . . ) ]))
```

An example: StreamingVideo

```
;; (streamingvideo-title info) produces title of info
;; streamingvideo-title: StreamingVideo → Str
;; Examples:
(check-expect (streamingvideo-title
      (make-movieinfo "Lang" "M" 99 'Crime)) "M")
(check-expect (streamingvideo-title
      (make-tvseries "Friends" 236 22)) "Friends")
(define (streamingvideo-title info) . . . )
```

The definition of streamingvideo-title

```
(define (streamingvideo-title info)
  (cond
    [(movieinfo? info) (movieinfo-title info)]
    [else (tvseries-title info)]))
```

A nested structure

```
(define-struct doublefeature (first second start-hour))
;; A DoubleFeature is a
;; (make-doublefeature MovieInfo MovieInfo Nat),
;; requires:
;;       start-hour is between 0 and 23, inclusive
```

An example of a DoubleFeature is

```
(define classic-movies
  (make-doublefeature
    (make-movieinfo "Welles" "Citizen Kane" 119 'Drama)
    (make-movieinfo "Kurosawa" "Rashomon" 88 'Mystery)
    20))
```

- Develop the function template.
- What is the title of the first movie?
- Do the two movies have the same genre?
- What is the total duration for both movies?

Structures containing lists

Suppose we store the name of a server along with a list of tips collected.

How might we store the information?

```
(define-struct server (name tips))
;; A Server is a (make-server Str (listof Num))
;; requires:
;;       numbers in tips are non-negative
```

We form templates for a server and for a list of numbers.

```
(define (server-template s)
  (local
      [(define (listof-num-template lon) . . . )]
      (. . . (server-name s) . . .
             (listof-num-template (server-tips s)) . . . ))
```

Note: We may choose to use abstract list functions or the basic list template for the helper function.

The function big-tips consumes a server s and a number smallest and produces the server formed from s by removing tips smaller than smallest.

```
(define (big-tips s smallest)
  (make-server (server-name s)
        (filter (lambda (tip) (<= smallest tip))
                (server-tips s))))
```

Lists of structures

Dealing with lists of structures will not be much different from dealing with other lists.

Depending on the problem, we may choose to use recursion or abstract list functions to process the list.

```
;; A (listof Student) is one of:
;; * empty
;; * (cons Student (listof Student))
```

```
(define myslist
  (list (make-student "Virginia Woolf" 100 100 100)
        (make-student "Alan Turing" 90 80 40)
        (make-student "Anonymous" 30 55 10) ))
```

What are the values of these expressions?

- (first myslist)
- (rest myslist)
- (student-mid (first (rest myslist)))

The template for (listof Student)

If using recusion, we just modify the basic list template and recognize that each item in the list is a Student.

```
;; listof-student-template: (listof Student) → Any
(define (listof-student-template slist)
  (cond
    [(empty? slist) ...]
      [else (... (student-template (first slist))
              ... (listof-student-template (rest slist))...)]))
```

Computing final grades

Suppose we wish to determine final grades for students based on their marks in each course component (20% for assignments, 30% for the midterm, and 50% for the final exam).

How should we store the information we produce?

```
(define-struct grade (name mark))
;; A Grade is a (make-grade Str Num),
;; requires:
;;     0 <= mark <= 100
```

The function compute-grades

```
;; (compute-grades slist) produces a grade list from slist.
;; compute-grades: (listof Student) → (listof Grade)
;; Examples:
(check-expect (compute-grades empty) empty)
(check-expect (compute-grades myslist)
  (list (make-grade "Virginia Woolf" 100)
        (make-grade "Alan Turing" 62)
        (make-grade "Anonymous" 27.5)))
```

```
(define (compute-grades slist)
  (local
    [(define assts-weight .20)
     (define mid-weight .30)
     (define final-weight .50)
     (define (final-grade astudent)
           (make-grade
                (student-name astudent)
                (+ (* assts-weight (student-assts astudent))
                     (* mid-weight (student-mid astudent))
          (* final-weight (student-final astudent)))))]
    (map final-grade slist)))
```

Goals of this module

You should be comfortable with these terms: structure, field, constructor, selector, type predicate, dynamic typing, static typing, data definition, structure definition, template.

You should be able to write functions that consume and produce structures, including Posns.

You should be able to create structure and data definitions for a new structure, determining an appropriate type for each field.

You should know what functions are defined by a structure definition, and how to use them.

You should 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.

You should understand the use of type predicates and be able to write code that handles mixed data.

You should understand how to process lists of structures.