Structures

Readings: HtDP, sections 6, 7.

- Avoid 6.2, 6.6, 6.7, 7.4.

Topics:

- Compound data
- Example: posn structures
- Defining & stepping structures
- Data definition and analysis
- Mixed data
- Lists vs. structures

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.

The teaching languages provide a general mechanism called **structures**.

They permit the “bundling” of several values into one.

In many situations, data is naturally grouped, and most programming languages provide some mechanism to do this.

There is also one predefined structure, posn, to provide an example.

Example: posn structures

- **constructor** function make-posn, with contract

  ;; make-posn: Num Num $\longrightarrow$ Posn

- **selector** functions posn-x and posn-y, with contracts

  ;; posn-x: Posn $\longrightarrow$ Num

  ;; posn-y: Posn $\longrightarrow$ Num

The constructor function is similar to cons while the selector functions are similar to first and rest.

Example:

```
(define mypoint (make-posn 8 1))
(posn-x mypoint) ⇒ 8
(posn-y mypoint) ⇒ 1
```

Possible uses:

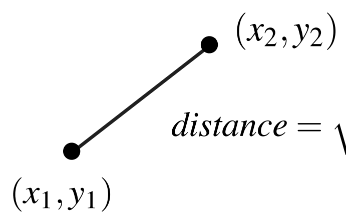
- coordinates of a point on a two-dimensional plane
- positions on a screen or in a window
- a geographical position

An expression such as (make-posn 8 1) is considered a value.

This expression will not be rewritten by the Stepper or our semantic rules.

The expression (make-posn (+ 4 4) (− 3 2)) would be rewritten to (eventually) yield (make-posn 8 1).

Example: point-to-point distance



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

```
;; (distance posn1 posn2) computes 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))))))
```

Functions that produce posns

```
;; (point-on-line slope intercept x) finds the point
;; on the line with given slope and intercept that has
;; the given x-coordinate
;; point-on-line: Num Num Num → Posn
;; Example:
(check-expect (point-on-line 3 7 2)
              (make-posn 2 13))

(define (point-on-line slope intercept x)
  (make-posn x (+ (∗ x slope) intercept)))
```

Another example

```
;; (scale v factor) scales vector v by the given factor
;; scale: Posn Num → Posn
;; Example:
(check-expect (scale (make-posn 3 4) 0.5)
              (make-posn 1.5 2))

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

Misusing posns

What is the result of evaluating the following expression?

```
(distance (make-posn 'Iron 'Man)
          (make-posn 'Tony 'Stark))
```

This causes a run-time error, but at a surprising point.

Defining structures

If posn wasn't built in, we could define it:

```
(define-struct posn (x y))
```

The arguments to the define-struct special form are:

- a structure name (e.g. posn), and
- a list of field names in parentheses.

Doing this once creates a number of functions that can be used many times.

The expression (define-struct posn (x y)) creates:

- **Constructor:** make-posn
- **Selectors:** posn-x, posn-y
- **Predicate:** posn?

  The posn? predicate tests if its argument is a posn.

Stepping with structures

The special form

(define-struct sname (fname1 . . . fnamen))

defines the structure type sname and automatically defines the following primitive functions:

- **Constructor:** make-sname
- **Selectors:** sname-fname1 . . . sname-fnamen
- **Predicate:** sname?

Sname may be used in contracts.

The substitution rule for the ith selector is:

(sname-fnamei (make-sname v1 . . . vi . . . vn)) ⇒ vi.

Finally, the substitution rules for the new predicate are:

(sname? (make-sname v1 . . . vn)) ⇒ true

(sname? V) ⇒ false for V a value of any other type.

In these rules, we use a pattern ellipsis.

An example using posns

```
(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)))) ⇒
```

```
(make-posn 2 (∗ 0.5 (posn-y (make-posn 4 2)))) ⇒
(make-posn 2 (∗ 0.5 2)) ⇒
(make-posn 2 1)
```

Data definition and analysis

Suppose we want to represent information associated with songs.

- The name of the performer
- The title of the song
- The genre of the music (rap, country, etc.)
- The length of the song

The data definition on the next slide will give a name to each field and associate a type of data with it.

Structure and data defs for SongInfo

```
(define-struct songinfo (performer title genre length))
;; An SongInfo is a (make-songinfo Str Str Sym Nat)
```

This creates the following functions:

- constructor make-songinfo,
- selectors songinfo-performer, songinfo-title, songinfo-genre, songinfo-length, and
- type predicate songinfo?.

Templates and data-directed design

As we noted earlier, one of the main ideas of the HtDP textbook is that the form of a program often mirrors the form of the data.

We make use of that for structures as well. Recall:

- A template is a general framework within which we fill in specifics.
- We create a template once for each new form of data, and then apply it many times in writing functions that consume that type of data.
- A template is derived from a data definition.

Templates for compound data

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.

```
;; songinfo-template: SongInfo → Any
(define (songinfo-template info)
  (... (songinfo-performer info) ...
       (songinfo-title info) ...
       (songinfo-genre info) ...
       (songinfo-length info) ...))
```

Example: update-genre

```
;; (update-genre oldinfo newgenre) produces a new SongInfo
;;     with the same information as oldinfo, except the genre
;;     is replaced by newgenre
;; update-genre: SongInfo Sym → SongInfo
;; Example:
(check-expect
  (update-genre
    (make-songinfo "C.O.C." "Eye For An Eye" 'Folk 78)
    'Punk)
  (make-songinfo "C.O.C." "Eye For An Eye" 'Punk 78))
```

```
;; update-genre: SongInfo Sym → SongInfo

(define (update-genre oldinfo newgenre)
  (make-songinfo
    (songinfo-performer oldinfo)
    (songinfo-title oldinfo)
    newgenre
    (songinfo-length oldinfo)))
```

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

Stepping update-genre

```
(define mysong (make-songinfo "U2" "Twilight" 'Rap 262))
(update-genre mysong 'Rock)
⇒ (update-genre
      (make-songinfo "U2" "Twilight" 'Rap 262) 'Rock)
⇒ (make-songinfo
      (songinfo-performer (make-songinfo "U2" "Twilight" 'Rap 262))
      (songinfo-title (make-songinfo "U2" "Twilight" 'Rap 262))
      'Rock
      (songinfo-length (make-songinfo "U2" "Twilight" 'Rap 262)))
```

Stepping an example (cont.)

```
⇒ (make-songinfo
     "U2"
     (songinfo-title (make-songinfo "U2" "Twilight" 'Rap 262))
     'Rock
     (songinfo-length (make-songinfo "U2" "Twilight" 'Rap 262)))
⇒ (make-songinfo
     "U2" "Twilight" 'Rock
     (songinfo-length (make-songinfo "U2" "Twilight" 'Rap 262)))
⇒ (make-songinfo "U2" "Twilight" 'Rock 262)
```

Design recipe for compound data

Do this *once per new structure type*:

Data Analysis and Definition: Define any new structures needed, based on problem description. Write data definitions for the new structures.

Template: Created once for each structure type, used for functions that consume that type.

Do the usual design recipe *for every function*:

Purpose: Same as before.

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

Examples: Same as before.

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

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

Mixed data

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

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

We can use these to check aspects of contracts, and to deal with data of mixed type.

Example: multimedia files

```
(define-struct movieinfo (director title genre duration))
;; A MovieInfo is a (make-movieinfo Str Str Sym Num )
;;
;; An MmInfo is one of:
;; ⋆ a SongInfo
;; ⋆ a MovieInfo
```

Here “mm” is an abbreviation for “multimedia”.

The template for mminfo

The template for mixed data is a cond with each type of data, and if the data is a structure, we apply the template for structures.

```
;; mminfo-template: MmInfo → Any
(define (mminfo-template info)
  (cond [(songinfo? info)
          (. . . (songinfo-performer info) . . .
                 (songinfo-title info) . . . )]; two more fields
         [(movieinfo? info)
          (. . . (movieinfo-director info) . . . )])); three more fields
```

```
(define favsong (make-songinfo "Beck" "Tropicalia"
                                'Alternative 185))
(define favmovie (make-movieinfo "Orson Welles" "Citizen Kane"
                                  'Classic 119))

;; (mminfo-artist info) produces performer/director name from info
;; mminfo-artist: MmInfo → Str
;; Examples:
(check-expect (mminfo-artist favsong) "Beck")
(check-expect (mminfo-artist favmovie) "Orson Welles")
```

```
(define (mminfo-artist info)
  (cond [(songinfo? info) (songinfo-performer info)]
        [(movieinfo? info)(movieinfo-director info)]))
```

The point of the design recipe and the template design:

- to make sure that one understands the type of data being consumed and produced by the function
- to take advantage of common patterns in code

anyof types

Unlike SongInfo and MovieInfo, there is no define-struct expression associated with MmInfo.

For the contract

```
;; mminfo-artist: MmInfo → Str
```

to make sense, the data definition for MmInfo must be included as a comment in the program.

Another option is to use the notation

```
;; mminfo-artist: (anyof SongInfo MovieInfo) → Str
```

Checked functions

We can write a safe version of make-posn.

```
;; safe-make-posn: Num Num → Posn
(define (safe-make-posn x y)
  (cond [(and (number? x) (number? y)) (make-posn x y)]
        [else (error "numerical arguments required")]))
```

The application (safe-make-posn 'Tony 'Stark) produces the error message “numerical arguments required”.

We were able to form the MmInfo type because of Racket’s dynamic typing.

Statically typed languages need to offer some alternative method of dealing with mixed data.

In later CS courses, you will see how the object-oriented features of inheritance and polymorphism gain some of this flexibility, and handle some of the checking we have seen in a more automatic fashion.

Lists versus structures

In M08 we wrote a Payroll data defintion for processing tax withholdings:

```
;; A Payroll is one of:
;; * empty
;; * (cons (list Str Num) Payroll)
```

Example payroll:

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

We also developed a corresponding template:

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
        [(cons? pr) (. . . (first (first pr)) . . .
                     . . . (first (rest (first pr))) . . .
                     . . . (payroll-template (rest pr)) . . . )]))
```

We recognized that some short helper functions will make our code more readable.

```
;; (name lst) produces the first item from lst – the name.
(define (name lst) (first lst))
;; (amount lst) produces the second item from lst – the amount.
(define (amount lst) (first (rest lst)))

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) . . . ]
        [(cons? pr) (. . . (name (first pr)) . . .
                     . . . (amount (first pr)) . . .
                     . . . (payroll-template (rest pr)) . . . )]))
```

Payroll with structures

We can accomplish the same goals with structures.

```
(define-struct payroll (name amount))
;; A Payroll is a (make-payroll Str Num)

(define (payroll-template pr)
   (cond [(empty? pr) . . . ]
         [(cons? pr) (. . . (payroll-name (first pr)) . . .
                      . . . (payroll-amount (first pr)). . .
                      . . . (payroll-template (rest pr)))]))
```

When should each of these two approaches be used?

Why use lists containings lists?

Recall that the result of our payroll program was a list of taxes owed.

Except for the name, the data definition is exactly the same as Payroll.

```
;; A TaxOwed is one of:
;; * empty
;; * (cons (list Str Num) TaxOwed)
```

We could write a single function to extract the names from either:

```
;; (name tax-rec) extracts the first item (the name) from tax-rec
(define (name tax-rec) (first tax-rec))

;; (list-names lst) produces a list of names from the payroll or ...
;; list-names: (anyof Payroll TaxOwed) → (listof Str)
(check-expect (list-names payroll) (list "Asha" "Joseph" "Sami"))
(define (list-names lst)
  (cond [(empty? lst) empty]
        [(cons? lst) (cons (name (first lst))
                           (list-names (rest lst)))]))
```

Why use lists containing lists?

The name-list function will produce a list of names from either a Payroll or a TaxOwed.

If we used structures, we would require two different (but very similar) functions or extra complexity in the same function to distinguish which structure selector to use.

We will exploit this ability to reuse code written to use “generic” lists when we discuss abstract list functions later in the course.

Why use structures?

Structure is often present in a computational task, or can be defined to help handle a complex situation.

Using structures helps avoid some programming errors (e.g., accidentally extracting a list of salaries instead of names).

Structures automatically create the selector functions we needed to make the list-based code readable.

Our design recipes can be adapted to give guidance in writing functions using complicated structures.

Structures are provided in all mainstream programming languages.

Goals of this module

You should understand the use of posns.

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 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 the similar uses of structures and fixed-size lists, and be able to write functions that consume either type of data.