Structures

Readings: HtDP, sections 6, 7.

• Avoid 6.2, 6.6, 6.7, 7.4.

Topics:

• Compound data [2–3]
• Example: posn structures [4–11]
• Defining & stepping structures [12–17]
• Data definition and analysis [18–27]
• Mixed data [28–36]
• Lists vs. structures [37–43]
Compound data

*Recall:* 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:

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

A name and salary always go together in this application.
Structures

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

*Key Point:* Structures permit the “bundling” of several related 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

There are functions to \textit{create} a structure (called \textbf{constructors}) and to \textit{produce components} of a structure (called \textbf{selectors}).

- The \textit{constructor} function \texttt{make-posn}, has the contract
  
  \texttt{;; make-posn: Num Num → Posn}

- \textit{selector} functions \texttt{posn-x} and \texttt{posn-y}, have the contracts
  
  \texttt{;; posn-x: Posn → Num}

  \texttt{;; posn-y: Posn → Num}

\textbf{Analogy:} The constructor function is similar to \textbf{cons} while the selector functions are similar to \textbf{first} and \textbf{rest}. 
Example: posn structures

(define mypoint (make-posn 8 1))

(posn-x mypoint) ⇒ 8
(posn-y mypoint) ⇒ 1

Possible uses of pos-n:

• coordinates of a point on a two-dimensional plane
• positions on a screen or in a window
• a geographical position
**Structures as values**

*Key Point:* An expression such as `(make-posn 8 1)` is considered a value.

That is, 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: distance in 2D

\[ \text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]
Example: distance in 2D

Functions can take posn’s as arguments.

;; (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 can produce posns

Functions can produce posn’s.

;; (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: scaling

Task: Multiply each component by the scale factor.

;; (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?

\[
\text{(distance (make-posn 'Iron 'Man)}
\hspace{1cm}
\text{(make-posn 'Tony 'Stark))}
\]

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

Key Point: If posn wasn’t built in, we could define it using the Racket function

```
(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.
Defining structures

The expression `(define-struct posn (x y))` creates functions that can be used to process `posn`'s.

- **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.
Semantics of structures

The *substitution rule* for the \(i\)th selector is:

\[
(s\text{-}name\text{-}fnamei \ (\text{make-sname} \ v1 \ldots vi \ldots vn)) \Rightarrow vi.
\]

Finally, the *substitution rules* for the new predicate are:

\[
(s\text{-}name? \ (\text{make-sname} \ v1 \ldots vn)) \Rightarrow \text{true}
\]

\[
(s\text{-}name? \ V) \Rightarrow \text{false} \text{ for } V \text{ a value of any other type.}
\]

In these rules, we again 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

The following code

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

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

*Key Point:* 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. I.e. it “unpacks” the data.

```scheme
;; 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))
Example: **update-genre**

;; update-genre: SongInfo Sym → SongInfo

\[
\text{(define (update-genre oldinfo newgenre)}
\text{(make-songinfo}
\text{(songinfo-performer oldinfo)}
\text{(songinfo-title oldinfo)}
\text{newgenre}
\text{(songinfo-length oldinfo))})
\]

*Key Point:* We could easily have done this without a template, but using it 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

*Key Point:* 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.
Design recipe for compound data

*Key Point:* 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?`.

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

*Key Point:* We can use these predicates to check aspects of contracts and to deal with data of mixed type.

Example: multimedia files
Example: multimedia files

First provide a data definition.

\[
\text{(define-struct movieinfo (director title genre duration))}
\]

;; 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”.

Next provide a template...
The template for mminfo

*Key Point: 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
mminfo example

(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")
mminfo example

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

Key Point: 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 or the following notation can be used

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

can be used.
Checked functions

**Key Point:** We can write a *safe version* of `make-posn`.

```scheme
;; 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”.

---

CS 135 Fall 2019 10: Structures
Mixed data and object-oriented programming

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

Recall: in M08 we wrote a Payroll data definition 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)) . . . )]
    ... (first (rest (first pr))) . . .
    ... (payroll-template (rest pr)) . . . ]))

*Key Point:* We recognized that two helper functions would make our code more readable, namely `name` and `amount`. 
;;; (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 improve readability using 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 containing 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)
Why use lists containing lists?
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?

*If we use lists, a single function,* name-list, *will produce a list of names for both a Payroll or a TaxOwed.*

*If we use structures, we would require two different 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 *more 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.
Module 10 Summary

Structures

1. **Structures** are used to store related data together. [3]
2. A **constructor** is a function used to create a structure. [4]
3. A **selector** is a function used to access the individual components of a structure. [4]
4. A **structure** is a value (i.e. it won’t be further simplified). [6]
5. When there is no operation defined for a particular type of value (e.g. Sym or Int), it causes a run-time error. [11]
6. The special form **define-struct** is used to create new types of structures. [12]
Module 10 Summary

7. The parts of the structure are called fields. [12]

8. The special form `define-struct` creates a constructor, selectors and a predicate for the newly defined structure. [13]

9. **Key Idea:** the form of a function often mirrors the form of the data so we create **templates** as a framework for a function in which we fill in the specifics. [20]

10. The template for a function, consumes a structure and selects every field in the structure. [21]

11. Use **data definitions** to specify the types that comprise a structure. [26]
Module 10 Summary

12. Create a Data Definition and Template for each new structure type. [26]

13. Then follow the Design Recipe for each function. [27]

14. For mixed data, use `cond` and predicates that check types when processing the data. [28, 30]

15. Use `anyof` to list the mixed types explicitly in a contract. [33]

16. You may use predicates and `error` to check a function’s argument types. [34]
Module 10 Summary

Lists vs. Structures

18. When we use *lists instead of structures* we can often reuse functions on related types. [41]

19. When we use *structures instead of lists* we can avoid some programming errors, make our code more readable, and can adapt the design recipe to give more guidance for writing functions. [43]