Structures

Readings: HtDP, sections 6, 7.

- Avoid 6.2, 6.6, 6.7, 7.4.
- These use the obsolete `draw.ss` teachpack.
- The new `image.ss` and `world.ss` are more functional.
Compound data

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.
Posn structures

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

  ;; make-posn: Num Num → Posn

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

  ;; posn-x: Posn → Num
  ;; posn-y: Posn → Num
Example:

\[(\text{define mypoint (make-posn 8 1))}\]
\[(\text{posn-x mypoint}) \Rightarrow 8\]
\[(\text{posn-y mypoint}) \Rightarrow 1\]

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 \((\text{make-posn} \ 8 \ 1)\) is considered a value.

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

The expression \((\text{make-posn} \ (+ \ 4 \ 4) \ (- \ 3 \ 2))\) would be rewritten to (eventually) yield \((\text{make-posn} \ 8 \ 1)\).
Example: point-to-point distance

\[ \text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]
;; (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 point factor) scales point 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 point factor)
  (make-posn (* factor (posn-x point))
             (* factor (posn-y point)))))
Misusing posns

What is the result of evaluating the following expression?

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

This causes a run-time error, but at a surprising point.
Racket does not enforce contracts, which are just comments, and ignored by the machine.

Each value created during the running of a program has a type (integer, Boolean, etc.).

Types are associated with values, not with constants or parameters.

(define p 5)
(define q (mystery-fn 5))
This is known as **dynamic typing**.

Many other mainstream languages use a more restrictive approach known as **static typing**.

With static typing, the header of our `distance` function might look like this:

```
real distance(Posn posn1, Posn posn2)
```

Here the contract is part of the language.

A program containing the function application `distance(3, "test")` would be illegal.
Dynamic typing is a potential source of both flexibility (as we will see) and confusion.

To avoid making mistakes such as the one with `make-posn`, we can use **data definitions**:

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

Definitions like this are human-readable comments and not enforced by the machine.

We can also create functions that check their arguments to catch type errors more gracefully (examples soon).
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 \texttt{(define-struct posn (x y))} creates:

- **Constructor:** \texttt{make-posn}
- **Selectors:** \texttt{posn-x, posn-y}
- **Predicate:** \texttt{posn?}

The \texttt{posn?} predicate tests if its argument is a \texttt{posn}.
Structures were added to the teaching languages before they were added to standard Racket in 2007.

It is not hard to build structures using other Racket features.

We will see a few ways to do it in CS 135 and CS 136.

Later on, we will see how structures can be viewed as the basis for objects, the main way of handling data and of organizing larger programs in many languages.
Stepping with structures

The special form

\begin{verbatim}
(define-struct sname (fname1 ... fnamen))
\end{verbatim}

defines the structure type \texttt{sname} and automatically defines the following primitive functions:

- **Constructor:** \texttt{make-sname}
- **Selectors:** \texttt{sname-fname1} \ldots \texttt{sname-fnamen}
- **Predicate:** \texttt{sname}? 

\texttt{Sname} may be used in contracts.
The substitution rule for the $i$th selector is:

$$(\text{sname-fname}i \ (\text{make-sname } v1 \ldots vi \ldots vn)) \Rightarrow vi.$$  

Finally, the substitution rules for the new predicate are:

$$(\text{sname? } (\text{make-sname } v1 \ldots vn)) \Rightarrow \text{true}$$  

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

In these rules, we use a pattern ellipsis.
An example using posns

\[(\text{define myposn (make-posn 4 2))}\]

\[(\text{scale myposn 0.5)} \Rightarrow\]

\[(\text{scale (make-posn 4 2) 0.5)} \Rightarrow\]

\[(\text{make-posn}\]

\[(* 0.5 (\text{posn-x (make-posn 4 2))})\]

\[(* 0.5 (\text{posn-y (make-posn 4 2))})) \Rightarrow\]

\[(\text{make-posn}\]

\[(* 0.5 4)\]

\[(* 0.5 (\text{posn-y (make-posn 4 2))})) \Rightarrow\]
(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 Num)

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

One of the main ideas of the HtDP textbook is that the form of a program often mirrors the form of the data.

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) ...))
An example

;; (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 an example

(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)))) ⇒
(make-songinfo "U2" "Twilight" 'Rock 262); after three steps
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.
Design recipe for compound data

Do the usual design recipe \textit{for every function}:

\textbf{Purpose}: Same as before.

\textbf{Contract}: Can use both built-in data types and defined structure names.

\textbf{Examples}: Same as before.

\textbf{Definition}: To write the body, expand the template based on examples.

\textbf{Tests}: Same as before. Be sure to capture all cases.
Dealing with 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")
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

```scheme
;; 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

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

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”.
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.
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 (anyof ...) notation in contracts. We will use it in later lecture modules.