Structures

- Compound data
- Defining & stepping structures
- Mixed data
- Lists vs. structures
Structures

- Readings: HtDP, sections 6, 7.
- Avoid 6.2, 6.6, 6.7, 7.4.
Compound data
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 -> Posn`

Selector functions `posn-x` and `posn-y`, with contracts
`;; posn-x: Posn -> Num`
`;; posn-y: Posn -> Num`

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

(define my-point (make-posn 8 1))
(posn-x my-point) => 8
(posn-y my-point) => 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

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

;; (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`

```scheme
;; (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 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 & stepping structures
Defining structures

If posn was not 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.
Defining structures

The expression \(\text{define-struct posn (x y)}\) creates:
- Constructor: \text{make-posn}
- Selectors: \text{posn-x, posn-y}
- Predicate: \text{posn?}

The \text{posn?} predicate tests if its argument is a \text{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 \( i \)th selector is:
\[
(sname\-fnamei (make\-sname v1 \ldots vi \ldots vn)) \Rightarrow vi
\]

Finally, the substitution rules for the new predicate are:
\[
(sname? (make\-sname v1 \ldots vn)) \Rightarrow true
\]
\[
(sname? V) \Rightarrow false \text{ for } V \text{ a value of any other type.}
\]

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

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

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

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

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

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

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

=> (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 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))

;; An Songinfo is a (make-songinfo Str Str Str Sym)

This creates the following functions:

- constructor make-songinfo,
- selectors songinfo-performer, songinfo-title, songinfo-genre, 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)))
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 "Edguy" "Mysteria" 'Metal) 'Power-Metal)
 (make-songinfo "Edguy" "Mysteria" 'Power-Metal)
;; update-genre: SongInfo Sym -> SongInfo
(define (update-genre oldinfo newgenre)
  (make-songinfo (songinfo-performer oldinfo)
                  (songinfo-title oldinfo)
                  newgenre))

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

(\texttt{define mysong}
 (\texttt{make-songinfo "Triple Concept" "Tonetwister" 'Trance}))

(\texttt{update-genre mysong 'Hardstyle})
=> (\texttt{update-genre}
 (\texttt{make-songinfo "Triple Concept" "Tonetwister" 'Trance})
 'Hardstyle)

=> (\texttt{make-songinfo}
 (\texttt{songinfo-performer}
 (\texttt{make-songinfo "Triple Concept" "Tonetwister" 'Trance}))
 (\texttt{songinfo-title}
 (\texttt{make-songinfo "Triple Concept" "Tonetwister" 'Trance})
 'Hardstyle)

=> (\texttt{make-songinfo}
 "Triple Concept"
 (\texttt{songinfo-title}
 (\texttt{make-songinfo "Triple Concept" Tonetwister" 'Trance})
 'Hardstyle)

=> (\texttt{make-songinfo "Triple Concept" "Tonetwister" 'Hardstyle})
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.