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:

(define mypoint (make-posn 8 1))

(posn-x mypoint) $\Rightarrow$ 8
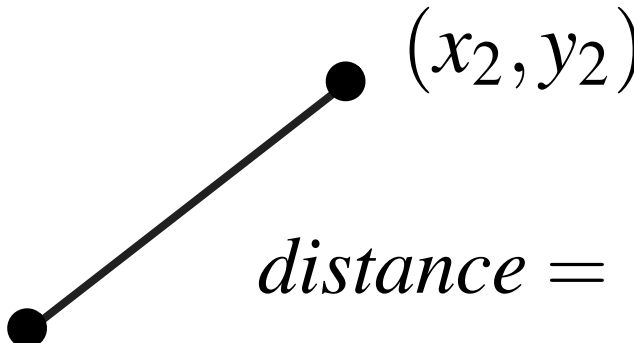
(posn-y mypoint) $\Rightarrow$ 1

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 (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

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

Point (x_2, y_2) and point (x_1, y_1).

```
;; (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 which produce posns

;; (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?

(distance (make-posn 'Iron 'Man)

  (make-posn 'Tony '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 also human-readable comments not enforced by the machine.

We can also create functions which check their inputs to catch errors of this type 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 (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.

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

(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)) $\Rightarrow$ vi.

Finally, the substitution rules for the new predicate are:

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

(sname? V) $\Rightarrow$ 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) $\Rightarrow$

(scale (make-posn 4 2) 0.5) $\Rightarrow$

(make-posn

  (∗ 0.5 (posn-x (make-posn 4 2)))

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

(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)

Data definition and analysis

Suppose we want to represent information associated with downloaded MP3 files.

- The name of the performer

- The title of the song

- The length 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 Mp3Info

(define-struct mp3info (performer title length genre))

;; An Mp3Info is a (make-mp3info Str Str Num Sym)

This creates the following functions:

- constructor make-mp3info,

- selectors mp3info-performer, mp3info-title, mp3info-length, mp3info-genre, and

- type predicate mp3info?.

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.

```
;; my-mp3info-fn: Mp3Info → Any
(define (my-mp3info-fn info)
    (… (mp3info-performer info) …
        (mp3info-title info) …
        (mp3info-length info) …
        (mp3info-genre info) …))
```

An example

```
;; (correct-performer oldinfo newname) corrects the name
;;    in oldinfo to be newname
;; correct-performer: Mp3Info Str → Mp3Info
;; Example:
(check-expect
  (correct-performer
    (make-mp3info "Hannah Montana" "Bite This" 80 'Punk)
            "Anonymous Doner Kebab")
  (make-mp3info "Anonymous Doner Kebab" "Bite This"
            80 'Punk))
```

```
;; correct-performer: Mp3Info Str → Mp3Info

(define (correct-performer oldinfo newname)
  (make-mp3info
    newname
    (mp3info-title oldinfo)
    (mp3info-length oldinfo)
    (mp3info-genre oldinfo)))
```

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

Stepping the example

(define mymp3 (make-mp3info "Avril Lavigne" "Boy" 180 'Rock))

(correct-performer mymp3 "U2") ⟹

(correct-performer

  (make-mp3info "Avril Lavigne" "Boy" 180 'Rock) "U2") ⟹

(make-mp3info

  "U2"

  (mp3info-title (make-mp3info "Avril Lavigne" "Boy" 180 'Rock))

  (mp3info-length (make-mp3info "Avril Lavigne" "Boy" 180 'Rock))

  (mp3info-genre (make-mp3info "Avril Lavigne" "Boy" 180 'Rock)))

⟹ (make-mp3info "U2" "Boy" 180 'Rock); 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 *for every function*:

Purpose: Same as before.

Contract: Can use both atomic 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.

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 input 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 duration genre))
;; A MovieInfo is a (make-movieinfo Str Str Num Sym )
;;
;; A MmInfo is one of:
;; ⋆ an Mp3Info
;; ⋆ 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.

```
;; my-mminfo-fn: MmInfo → Any
(define (my-mminfo-fn info)
  (cond [(mp3info? info)
          (... (mp3info-performer info) ...
               (mp3info-title info) ... )]; two more fields
        [(movieinfo? info)
          (... (movieinfo-director info) ... )])); three more fields
```

```
(define favsong (make-mp3info "Beck" "Tropicalia"
                              185 'Alternative))
(define favmovie (make-movieinfo "Orson Welles" "Citizen Kane"
                                 119 'Classic))


;; (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 [(mp3info? info) (mp3info-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 Mp3Info 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 Mp3Info 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.

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.