Module 4: Compound data: structures

Readings: Sections 6 and 7 of HtDP.

- Sections 6.2, 6.6, 6.7, 7.4, and 10.3 are optional reading; they use the obsolete `draw.ss` teachpack.
- The teachpacks `image.ss` and `world.ss` are more useful.

Compound data

Data may naturally be joined, but a function can produce only a single item.

A **structure** is a way of “bundling” several pieces of data together to form a single “package”.

We can

- create functions that consume and/or produce structures, and
- define our own structures, automatically getting (“for free”) functions that create structures and functions that extract data from structures.

Posn structures

A posn is a built-in structure that has two **fields** containing numbers intended to represent x and y coordinates.

The **constructor** function make-posn, has contract

;; make-posn: num num $\longrightarrow$ posn

Each **selector** function consumes a posn and produces a number.

There is one per number, posn-x and posn-y.

The function posn? is a **type predicate**.

```
(define myposn (make-posn 8 1))
(posn-x myposn) ⇒ 8
(posn-y myposn) ⇒ 1
(posn? myposn) ⇒ true
```

Substitution rules: for any values a and b

(posn-x (make-posn a b)) $\Rightarrow$ a

(posn-y (make-posn a b)) $\Rightarrow$ b

The make-posn you type is a function application.

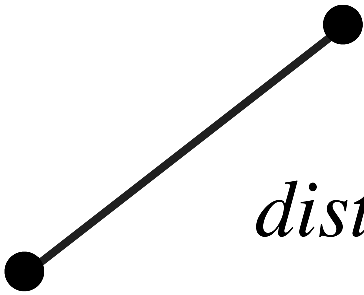
The make-posn DrRacket displays is a marker to show that the value is a posn.

(make-posn (+ 4 4) (− 2 1)) simplifies to (make-posn 8 1)

(make-posn 8 1) cannot be simplified.

Example: point-to-point distance

(x_2, y_2)



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

(x_1, y_1)

The function distance

```
(define first-point (make-posn 1 1))
(define second-point (make-posn 4 5))
;; distance: posn posn → num[≥0]
;; Produces the Euclidean distance between posn1 and posn2.
;; Examples: (distance first-point second-point) ⇒ 5
(define (distance posn1 posn2)
  (sqrt (+ (sqr (− (posn-x posn1) (posn-x posn2)))
           (sqr (− (posn-y posn1) (posn-y posn2))))))
;; Test for distance
(check-expect (distance first-point second-point) 5)
```

When we have a function that consumes a number and produces a number, we do not change the number we consume.

Instead, we make a new number.

The function scale consumes a posn and produces a new posn.

It doesn't change the old one.

Instead, it uses make-posn to make a new posn.

Functions that produce posns

```
;; scale: posn num → posn
;; Produces the posn of point scaled by factor.
;; Examples: (scale (make-posn 3 4) 0.5) ⇒ (make-posn 1.5 2)
;; (scale (make-posn 1 2) 1) ⇒ (make-posn 1 2)
(define (scale point factor)
  (make-posn (∗ factor (posn-x point))
             (∗ factor (posn-y point))))
;; Tests for scale
(check-expect (scale (make-posn 3 4) 0.5) (make-posn 1.5 2))
(check-expect (scale (make-posn 1 2) 1) (make-posn 1 2))
```

```
(define point1 (make-posn 'Hannah 'Montana))
(define point2 (make-posn 'Miley 'Cyrus))
(distance point1 point2)
```

Dynamic typing: the type of a value bound to an identifier is determined by the program as it is run,

e.g. `(define x (check-divide n))`

Static typing: constants and what functions consume and produce have pre-determined types,

e.g. `real distance(Posn posn1, posn2)`

Scheme uses dynamic typing.

Pros and cons of dynamic typing

Pros:

- No need to write down pre-determined types.
- Flexible: the same definitions can be used for various types (e.g. lists in Module 5, functions in Module 10).

Cons:

- Contracts are not enforced by the computer.
- Type errors are caught only at run time.

Dealing with dynamic typing

Approach 1: don't trust anybody

```
;; safe-make-posn: num num → posn
(define (safe-make-posn xcoord ycoord)
  (cond
    [(and (number? xcoord) (number? ycoord))
     (make-posn xcoord ycoord)]
    [else (error 'safe-make-posn "numerical arguments required")]))
```

Approach 2: data definitions

Data definitions

Data definition: a comment specifying a data type; for structures, include name of structure, number and types of fields.

```
;; A posn is a structure (make-posn xcoord ycoord), where
;;       xcoord is a number and
;;       ycoord is a number.
```

Using posn in a contract now means fields contain numbers.

Any type defined by a data definition can be used in a contract.

Structure definitions

Structure definition: code defining a structure, and resulting in constructor, selector, and type predicate functions.

```
(define-struct sname (field1 field2 field3))
```

Writing this once creates functions that can be used many times:

- **Constructor**: make-sname
- **Selectors**: sname-field1, sname-field2, sname-field3
- **Predicate**: sname?

For a new structure we define, we get the functions “for free” by creating a structure definition.

For posn these functions are already built in, so a structure definition isn’t needed.

Create a data definition for each structure definition.

Put them first, before defined constants and defined functions.

Design recipe modifications

Data analysis and design: design a data representation that is appropriate for the information handled in our function.

Later: more choices for data representations.

Now: determine which structures are needed and define them.

Include

- a structure definition (code) and
- a data definition (comment).

Structures for MP3 files

Suppose we want to represent information associated with downloaded MP3 files, that is:

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

```
(define-struct mp3info (performer title length genre))
;; An mp3info is a structure (make-mp3info p t l g) where
;;      p is a string (name of performer),
;;      t is a string (name of song),
;;      l is a nat (length in seconds), and
;;      g is a symbol (genre of music).
```

The structure definition gives us:

- Constructor make-mp3info
- Selectors mp3info-performer, mp3info-title, mp3info-length, and mp3info-genre
- Predicate mp3info?

```
(define song
  (make-mp3info "Kelly Clarkson" "Bite This" 80 'Punk))
(mp3info-length song) ⇒ 80
(mp3info? 6) ⇒ false
```

```
;; correct-performer: mp3info string → mp3info
;; Produces mp3info formed from oldinfo, correcting
;; performer to newname.
;; Example: (correct-performer
;;                 (make-mp3info "Kelly Clarkson" "Bite This" 80 'Punk)
;;                 "Anonymous Doner Kebab")⇒
;; (make-mp3info "Anonymous Doner Kebab" "Bite This" 80 'Punk)
(define (correct-performer oldinfo newname)...)
```

More design recipe modifications: templates

Key idea: the form of a program often mirrors the form of the data consumed.

A **template** is a general framework within which we fill in specifics.

A template is derived from a data definition.

We create a template once for each type of data, and then apply it many times in writing functions that consume that type.

Templates for structures use selectors on each field, even though a specific function might not use all the selectors.

A template for mp3info

This template can be used for any function that consumes an mp3info structure.

```
;; my-mp3info-fun: mp3info → any
(define (my-mp3info-fun info)
  ... (mp3info-performer info) ...
  ... (mp3info-title info) ...
  ... (mp3info-length info) ...
  ... (mp3info-genre info) ... )
```

Note how it mirrors the data definition.

Using templates to create functions

- Choose a template and examples that fit the type(s) of data the function consumes.
- For each example, figure out the values for each part of the template.
- Figure out how to use the values to obtain the value produced by the function.
- Different examples may lead to different cases.
- Different cases may use different parts of the template.
- If a part of a template isn't used, it can be omitted.
- New parameters can be added as needed.

The function correct-performer

We use the parts of the template that we need, and add a new parameter.

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

Templates are crucial when designing complicated functions.

Using the design recipe for compound data

Step 1: Data analysis and design.

- Define any new structures needed for the problem.
- Structure and data definitions appear before the function.

Step 2: Template.

- Create based on the data definition.
- Use for each function that consumes that type.
- Blank template does not need to appear in comments.

Step 3: Contract and function header.

- Number and order of parameters should match.
- Atomic data types and compound types are allowed.

Steps 4 and 5: Purpose and Examples (unchanged).

Step 6: Body.

- Use the template that matches the data consumed.
- Use the examples to help you fill in the blanks.

Step 7: Tests (unchanged).

Design recipe example

Suppose we wish to create a function card-colour that consumes a card and produces the symbol ’red or ’black indicating the colour of the suit of a playing card.

Step 1: Data analysis and design.

```
(define-struct card (value suit))
;; A card is a structure (make-card v s), where
;;       v is an integer in the range from 1 to 10 and
;;       s is a symbol from the set 'hearts, 'diamonds, 'spades, and 'clubs
```

The structure definition gives us:

- Constructor make-card
- Selectors card-value and card-suit
- Predicate card?

Step 2: Template.

Templates for cards

We can form a template for use in any function that consumes a single card:

```
(define (my-card-fun acard)
... (card-value acard) ...
... (card-suit acard) ...)
```

You might find it convenient to use constant definitions to create some data for use in examples and tests.

```
(define tenofhearts (make-card 10 'hearts))
(define oneofdiamonds (make-card 1 'diamonds))
(define threeofspades (make-card 3 'spades))
(define fourofclubs (make-card 4 'clubs))
```

Mixed data and structures

mminfo-artist consumes a multimedia file and produces either the performer of an MP3 or the director of a movie.

```
(define-struct movieinfo (director title duration genre))
;; A movieinfo is a structure (make-movieinfo di t du g), where
;;       di is a string (director),
;;       t is a string (title),
;;       du is a number (duration in minutes), and
;;       g is a symbol (genre or type).

;; mminfo-artist: (union mp3info movieinfo) → string
```

Defining a new data type mminfo

To use mminfo in a contract, we need a data definition.

There is no structure definition for mminfo.

It is defined in terms of mp3info and movieinfo, which are structures defined before the function.

```
;; An mminfo is either
;;     an mp3info or
;;     a movieinfo.
```

```
;; mminfo-artist: mminfo → string
;; Produces performer/director name from info
;; Examples:
;; (mminfo-artist
;;     (make-mp3info “Beck” “Tropicalia” 185 ’Alternative))
;; ⇒ “Beck”
;; (mminfo-artist
;;     (make-movieinfo “Orson Welles” “Citizen Kane” 119 ’Drama))
;; ⇒ “Orson Welles”
```

The template for mminfo

The template for mixed data is a cond with one question for each type of data.

We use type predicates in our questions.

If the data is a structure, we apply the template for structures.

```
(define (my-mminfo-fun info)
  (cond [(mp3info? info)
         . . . (mp3info-performer info) . . .
         . . . (mp3info-title info) . . .
         . . . (mp3info-length info) . . .
         . . . (mp3info-genre info) . . . ]
         [(movieinfo? info)
         . . . (movieinfo-director info) . . .
         . . . (movieinfo-title info) . . .
         . . . (movieinfo-duration info) . . .
         . . . (movieinfo-genre info) . . . ]))
```

The definition of mminfo-artist

```
(define (mminfo-artist info)
  (cond
    [(mp3info? info) (mp3info-performer info)]
    [(movieinfo? info) (movieinfo-director info)]))
```

Reasons for 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

Additions to syntax for structures

The special form (define-struct sname (field1 . . . fieldn)) defines the structure type sname and automatically defines the following primitive functions:

- **Constructor:** make-sname
- **Selectors:** sname-field1 . . . sname-fieldn
- **Predicate:** sname?

A **value** is a number, a symbol, a character, a string, a boolean, or is of the form (make-sname v1 . . . vn) for values v1 through vn.

Additions to semantics for structures

The substitution rule for the ith selector is:

(sname-fieldi (make-sname v1 . . . vi . . . vn)) $\Rightarrow$ vi

The substitution rules for the type predicate are:

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

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

An example using posns

Recall the definition of the function scale :

```
(define (scale point factor)
  (make-posn (* factor (posn-x point))
             (* factor (posn-y point))))
```

Then we can make the following substitutions:

```
(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))))
```

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

$\Rightarrow$ (make-posn 2 ($*$ 0.5 2))

$\Rightarrow$ (make-posn 2 1)

Since (make-posn 2 1) is a value, no further substitutions are needed.

Another example

```
(define mymp3 (make-mp3info "Avril Lavigne" "One " 276 'Rock))
(correct-performer mymp3 "U2")
⇒ (correct-performer
  (make-mp3info "Avril Lavigne" "One " 276 'Rock) "U2")
⇒ (make-mp3info
  "U2"
  (mp3info-title (make-mp3info "Avril Lavigne" "One " 276 'Rock))
  (mp3info-length (make-mp3info "Avril Lavigne" "One " 276 'Rock))
  (mp3info-genre (make-mp3info "Avril Lavigne" "One " 276 'Rock)))
```

```
⇒ (make-mp3info
   "U2" "One"
   (mp3info-length (make-mp3info "Avril Lavigne" "One " 276 'Rock))
   (mp3info-genre (make-mp3info "Avril Lavigne" "One " 276 'Rock)))
⇒ (make-mp3info
   "U2" "One" 276
   (mp3info-genre (make-mp3info "Avril Lavigne" "One " 276 'Rock)))
⇒ (make-mp3info "U2" "One " 276 'Rock)
```

A nested structure

```
(define-struct DoubleFeature (first-movie second-movie start-hour))
;; A DoubleFeature is a structure (make-DoubleFeature f s st), where
;;       f is a movieinfo (first movie of DoubleFeature),
;;       s is a movieinfo (second movie of DoubleFeature), and
;;       st is an int between 1 and 24 (starting hour of first movie)
```

An example of a DoubleFeature is

```
(define classic-movies
  (make-DoubleFeature
    (make-movieinfo "Orson Welles" "Citizen Kane" 119 'Drama)
    (make-movieinfo "Akira Kurosawa" "Rashomon" 88 'Mystery)
    20))
```

- What is the name of the first movie?
- Do the two movies have the same genre?
- What time does the second movie end?

Design recipe for compound data

1. **Data analysis and design**. Put structure and data definitions first.
2. **Template**. Create based on the data definition.
3. **Contract** and **function header**. Compound data can be used.
4. **Purpose**.
5. **Examples**.
6. **Body**. Use the template for the data consumed.
7. **Tests**.
8. Run the program.

Goals of this module

You should be comfortable with these terms: structure, field, constructor, selector, type predicate, dynamic typing, static typing, data definition, structure definition, template.

You should be able to write functions that consume and produce structures, including posns.

You should be able to create structure and data definitions for a new structure, determining an appropriate type for each field.

You should know what functions are defined by a structure definition, and how to use them.

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.