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.
- Note that none of these particular teachpacks will be used on assignments or exams.

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 (short for Position) is a built-in structure that has two **fields** containing numbers intended to represent x and y coordinates.

We might want to use a Posn to represent:

- coordinates of a point on a 2-D plane
- positions on a screen or in a window
- a geographical position

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

;; make-posn: Num Num $\rightarrow$ Posn

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

;; posn-x: Posn $\longrightarrow$ Num

;; posn-y: Posn $\longrightarrow$ Num

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

;; posn?: Any $\longrightarrow$ Bool

Examples

```
(define myposn (make-posn 8 1))
```

`(posn-x myposn)` $\Rightarrow$ 8

`(posn-y myposn)` $\Rightarrow$ 1

`(posn? myposn)` $\Rightarrow$ true

Substitution rules

For any values a and b

(posn-x (make-posn a b)) ⇒ a

(posn-y (make-posn a b)) ⇒ b

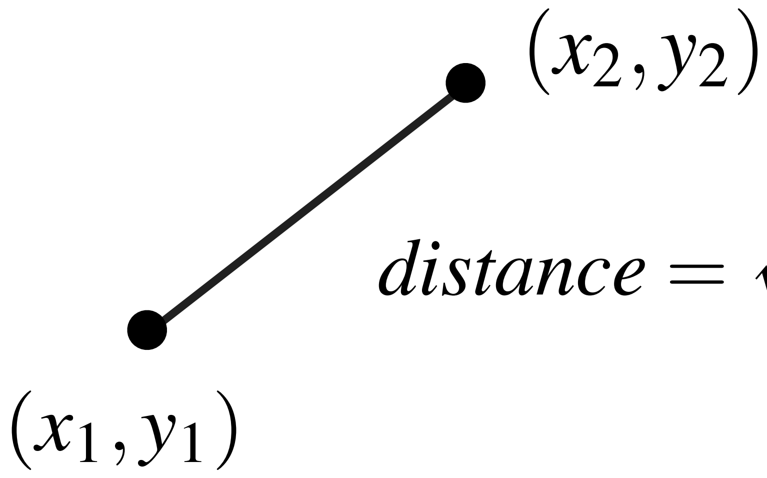
The make-posn you type is a function application.

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



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

The function **distance**

```
;; (distance posn1 posn2) produces 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 posn1) (posn-x posn2)))
           (sqr (− (posn-y posn1) (posn-y posn2))))))
```

Functions that produce posns

```
;; (scale point factor) produces the Posn that results when the fields
;;    of point are multiplied by factor
;; scale: Posn Num → Posn
;; Examples:
(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 (scale point factor)
  (make-posn (∗ factor (posn-x point))
             (∗ factor (posn-y point))))
```

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.

Misusing Posn

What is the result of evaluating the following expression?

```
(define point1 (make-posn "Math135" "CS115"))
(define point2 (make-posn 'Red true))
(distance point1 point2)
```

This causes a run-time error, but possibly not where you think.

Racket does not enforce contracts, which are just comments, and ignored by the machine.

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

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

```
(define p 5)
(define q (mystery-fn 5))
```

Racket uses dynamic typing

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)`

While Racket does not enforce contracts, we will always assume that contracts for our functions are followed. Never call a function with data that violates the contract and requirements.

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

Dynamic typing is a potential source of both flexibility 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)
```

Using Posn in a contract now means fields contain numbers.

Data definitions

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

Data definitions like this are also comments, and are not enforced by the computer. However, we will assume data definitions are always followed, unless explicitly told otherwise.

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).

and submit with your assignment.

Structure 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 (make-mp3info Str Str Nat Sym)
;; requires:
;;      performer is the name of performer of the song,
;;      title is the name of song,
;;      length is the length of the song in seconds,
;;      genre is the genre (type or category) of the song.
```

Note: If all the field names for a new structure are self-explanatory, we will often omit the field-by-field descriptions.

The structure definition gives us:

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

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

Templates and data-directed design

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

A **template** is a general framework which we will complete with specifics. It is the starting point for our implementation.

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

A template is derived from a data definition.

A template for Mp3Info

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-fun: Mp3Info → Any
(define (my-mp3info-fun info)
  ... (mp3info-performer info)...
  ... (mp3info-title info)...
  ... (mp3info-length info)...
  ... (mp3info-genre info)... )
```

An example

```
;; (correct-performer oldinfo newname) produces a new Mp3Info
;;    formed from oldinfo, correcting performer to newname.
;; correct-performer: Mp3Info Str → Mp3Info
;; example:
(check-expect
  (correct-performer
     (make-mp3info "LaLaLa Singer" "Bite This" 80 'Punk)
     "Anonymous Doner Kebab")
  (make-mp3info "Anonymous Doner Kebab" "Bite This" 80 'Punk))
```

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)))
```

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

Additions to syntax for structures

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

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

A **value** is a number, a symbol, 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 a value V 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))))
```

⇒ (make-posn 2 (∗ 0.5 (posn-y (make-posn 4 2))))

⇒ (make-posn 2 (∗ 0.5 2))

⇒ (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)
```

Using the design recipe for compound data

Step 1: Data analysis and design.

- Define any new structures needed for the problem.
- Write structure and data definitions for each new type, and include right after the file header.

Using the design recipe for compound data

Step 2: Template.

- Create one template for each new type defined, and use for each function that consumes that type.
- Use a generic name for the template function and include a generic contract.

Step 3: Purpose (unchanged).

Step 4: Contract and requirements.

- Atomic data types and compound types are allowed.
- Must use type names from the new data definitions.

Steps 5: Examples (unchanged).

Step 6: Function definition (header and 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 playing card and produces the symbol 'red or 'black indicating the colour of the suit of that card. For simplicity, we ignore face cards.

Data analysis and design.

```
(define-struct card (value suit))
;; A Card is a (make-card Nat Sym)
;; requires
;;       value is between 1 to 10, inclusive, and
;;       suit is one of the values 'hearts, 'diamonds, 'spades, and 'clubs.
```

The structure definition gives us:

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

The data definition tells us:

- types required by make-card
- types produced by card-value and card-suit

Templates for Cards

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

```
;; my-card-fun: Card → Any
(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

Consider writing functions that use a multimedia file (MP3 or movie).

```
(define-struct movieinfo (director title duration genre))
;; A MovieInfo is a (make-movieinfo Str Str Nat Sym)
;; requires:
;;      duration is a the length of the movie in minutes
;;
;; A MmInfo is one of:
;; * an Mp3Info or
;; * a MovieInfo.
```

Note that the MmInfo does not require a structure definition.

The template for Mminfo

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

```
;; my-mminfo-fun: Mminfo → Any
(define (my-mminfo-fun info)
  (cond [(mp3info? info) . . . ]
        [(movieinfo? info) . . . ]))
```

We use type predicates in our questions.

Next, expand the template to include more information about the structures.

```
;; my-mminfo-fun: Mminfo → Any
(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) . . . ]))
```

An example: mminfo-artist

```
;; (mminfo-artist info) produces performer/director name from info
;; mminfo-artist: MmInfo → Str
;; Examples:
(check-expect (mminfo-artist
    (make-mp3info "Beck" "Tropicalia" 185 'Alternative)) "Beck")
(check-expect (mminfo-artist
    (make-movieinfo "Orson Welles" "Citizen Kane" 119 'Drama))
    "Orson Welles")

(define (mminfo-artist 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 you understand the type of data being consumed and produced by the function
- to take advantage of common patterns in code

Reminder: anyof types

If a consumed or produced value for a function can be one of a restricted set of types, we will use the notation

(anyof type1 type2 . . . typeK)

For example, if we hadn't defined MmInfo as a new type, we could have written the contract for mminfo-artist as

```
;; mminfo-artist: (anyof Mp3Info MovieInfo) → Str
```

A nested structure

```
(define-struct doublefeature (first second start-hour))
;; A DoubleFeature is a
;; (make-doublefeature MovieInfo MovieInfo Nat),
;; requires:
;;      start-hour is between 0 and 23, inclusive, for the
;;      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))
```

- Develop the function template.
- What is the title of the first movie?
- Do the two movies have the same genre?
- What is the total duration for both movies?

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.