

# 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 readings; 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.

# A new type

Suppose we want to design a program for a card game such as poker or cribbage. Before writing any functions, we have to decide on how to represent data.

For each card, we have a suit (one of hearts, diamonds, spades, and clubs) and a rank (for simplicity, we will consider ranks as integers between 1 and 13). We can create a new structure with two fields using the following **structure definition**.

```
(define-struct card (rank suit))
```

# Using the **Card** type

Once we have defined our new type, we can:

- Create new values using the **constructor** function `make-card`

```
(define h5 (make-card 5 "hearts"))
```

- Retrieve values of the individual fields using the **selector** functions `card-rank` and `card-suit`

```
(card-rank h5) ⇒ 5
```

```
(card-suit h5) ⇒ "hearts"
```

We can also

- Check if a value is of type `Card` using the **type predicate** function `card?`

`(card? h5) ⇒ true`

`(card? "5 of hearts") ⇒ false`

Once the new structure `card` has been defined, the functions `make-card`, `card-rank`, `card-suit`, `card?` are created by Racket. We do not have to write them ourselves.

We have grouped all the data for a single card into one value, and we can still retrieve the individual pieces of information.

# More information about **Card**

The structure definition of **Card** does not provide all the information we need to use the new type properly. We will use a comment called a **data definition** to provide additional information about the types of the different field values.

```
(define-struct card (rank suit))
```

```
:: A Card is a
```

```
::   (make-card Nat (anyof "hearts" "diamonds" "spades" "clubs"))
```

```
:: requires
```

```
::   rank is between 1 and 13, inclusive
```

# Functions using **Card** values

:: (pair? c1 c2) produces true if c1 and c2 have the same rank,

:: and false otherwise

:: pair?: Card Card  $\rightarrow$  Bool

```
(define (pair? c1 c2) (= (card-rank c1) (card-rank c2)))
```

:: (one-better c) produces a Card, with the same suit as c, but

:: whose rank is one more than c (to a maximum of 13)

:: one-better: Card  $\rightarrow$  Card

```
(define (one-better c)
```

```
  (make-card (min 13 (+ 1 (card-rank c))) (card-suit c)))
```

# 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 a geographical position.

The structure definition is built-in. We'll use the following data definition.

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



# Built-in functions for **Posn**

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

:: posn-x: Posn  $\rightarrow$  Num

:: posn-y: Posn  $\rightarrow$  Num

:: posn?: Any  $\rightarrow$  Bool

## Examples of use

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

(posn-x myposn)  $\Rightarrow$  8

(posn-y myposn)  $\Rightarrow$  1

(posn? myposn)  $\Rightarrow$  true

# Substitution rules and simplified values

For any values  $a$  and  $b$

$(\text{posn-x } (\text{make-posn } a \ b)) \Rightarrow a$

$(\text{posn-y } (\text{make-posn } a \ b)) \Rightarrow b$

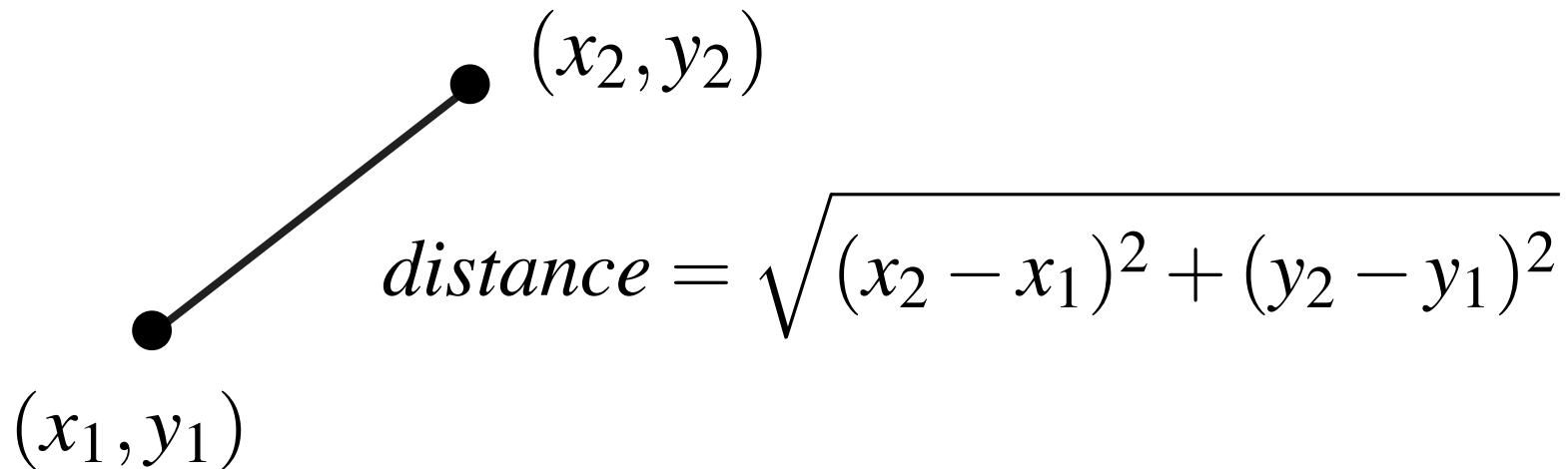
The `make-posn` you type is a function application.

The `make-posn` DrRacket displays indicates that the value is of type `posn`.

`(make-posn (+ 4 4) (- 2 1))` yields `(make-posn 8 1)`, which cannot be further simplified.

Similar rules apply to our newly defined `card` structure as well.

# Example: point-to-point distance



$(x_1, y_1)$

$(x_2, y_2)$

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

# The function **distance**

;; (distance posn1 posn2) produces the Euclidean distance

;; between posn1 and posn2.

;; distance: Posn Posn  $\rightarrow$  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))))))
```

# Function that produces a Posn

:: (scale point factor) produces the Posn that results when the fields

:: of point are multiplied by factor

:: scale: Posn Num  $\rightarrow$  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 or data definitions, 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. Writing a data definition for each new structure will help us avoid making mistakes with structures.

**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 for new structures

**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?`

# Design recipe modifications

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

Determine which structures are needed and define them. Include

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

Place the structure and data definitions immediately after the file header, before your constants and functions.

# Structure for Movie information

Suppose we want to represent information associated with movies, that is:

- the name of the director
- the title of the movie
- the duration of the movie
- the genre of the movie (sci-fi, drama, comedy, etc.)

```
(define-struct movieinfo (director title duration genre))  
;; A MovieInfo is a (make-movieinfo Str Str Nat Str)  
;; where:  
;;   director is the name of director of the movie,  
;;   title is the name of movie,  
;;   duration is the length of the movie in minutes,  
;;   genre is the genre (type or category) of the movie.
```

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-movieinfo`
- Selectors `movieinfo-director`, `movieinfo-title`, `movieinfo-duration`, and `movieinfo-genre`
- Predicate `movieinfo?`

```
(define et-movie
```

```
  (make-movieinfo "Spielberg" "ET" 115 "Sci-Fi"))
```

```
(movieinfo-duration et-movie) ⇒ 115
```

```
(movieinfo? 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 **MovieInfo**

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.

```
::; movieinfo-template: MovieInfo → Any
```

```
(define (movieinfo-template info)  
  (... (movieinfo-director info)...  
    (movieinfo-title info)...  
    (movieinfo-duration info)...  
    (movieinfo-genre info)...))
```

# An example

:: (correct-genre oldinfo newg) produces a new MovieInfo

:: formed from oldinfo, correcting genre to newg.

:: correct-genre: MovieInfo Str → MovieInfo

:: example:

(check-expect

  (correct-genre

    (make-movieinfo "Spielberg" "ET" 115 "Comedy")

    "Sci-Fi")

  (make-movieinfo "Spielberg" "ET" 115 "Sci-Fi"))

# 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-genre`

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

```
(define (correct-genre oldinfo newg)
  (make-movieinfo (movieinfo-director oldinfo)
                  (movieinfo-title oldinfo)
                  (movieinfo-duration oldinfo)
                  newg))
```

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 a constructor function, selector functions for each field, and a type predicate function.

A **value** is a number, 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  $i$ th selector is:

$(\text{sname-field}_i (\text{make-sname } v_1 \dots v_i \dots v_n)) \Rightarrow v_i$

The substitution rules for the type predicate are:

$(\text{sname? } (\text{make-sname } v_1 \dots v_n)) \Rightarrow \text{true}$

$(\text{sname? } V) \Rightarrow \text{false}$  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 mymovie (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
```

```
(correct-genre mymovie "Fantasy")
```

```
⇒ (correct-genre
```

```
(make-movieinfo "Reiner" "Princess Bride" 98 "War") "Fantasy")
```

```
⇒ (make-movieinfo
```

```
(movieinfo-director (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
```

```
(movieinfo-title (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
```

```
(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
```

```
"Fantasy")
```

⇒ (make-movieinfo

"Reiner"

(movieinfo-title (make-movieinfo "Reiner" "Princess Bride" 98 "War"))

(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 "War"))

"Fantasy")

⇒ (make-movieinfo

"Reiner"

"Princess Bride"

(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 "War"))

"Fantasy")

⇒ (make-movieinfo "Reiner" "Princess Bride" 98 "Fantasy")

# Design recipe for compound data

Do this *once per new structure type*:

**Data analysis and design:** Define any new structures needed for the problem. Write structure and data definitions for each new type (include right after the file header).

**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.

**Do the usual design recipe steps for *every function*:**

**Purpose:** Same as before.

**Contract and requirements:** Can use both built-in data types and defined structure names.

**Examples:** Same as before.

**Function Definition:** To write the body, expand the template based on the examples.

**Tests:** Same as before. Be sure to capture all cases.

# Design recipe example

Suppose we wish to create a function `total-length` that consumes information about a TV series, and produces the total length (in minutes) of all episodes of the series.

Data analysis and design.

```
(define-struct tvseries (title eps len-per))
```

```
:: A TVSeries is a (make-tvseries Str Nat Nat)
```

```
:: where
```

```
::   title is the name of the series
```

```
::   eps is the total number of episodes
```

```
::   len-per is the average length (in minutes) for one episode
```

The structure definition gives us:

- Constructor `make-tvseries`
- Selectors `tvseries-title`, `tvseries-eps`, and `tvseries-len-per`
- Predicate `tvseries?`

The data definition tells us:

- types required by `make-tvseries`
- types produced by `tvseries-title`, `tvseries-eps`, and `tvseries-len-per`



# Templates for TVSeries

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

```
:: tvseries-template: TVSeries → Any
```

```
(define (tvseries-template show)  
  (... (tvseries-title show) ...  
        (tvseries-eps show) ...  
        (tvseries-len-per show) ... ))
```

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

```
(define murdoch (make-tvseries "Murdoch Mysteries" 168 42))
```

```
(define friends (make-tvseries "Friends" 236 22))
```

```
(define fawlty (make-tvseries "Fawlty Towers" 12 30))
```

# Mixed data and structures

Consider writing functions that use a streaming video file (movie or tv series), which does not require any new structure definitions.

```
(define-struct movieinfo (director title duration genre))
```

```
:: A MovieInfo is a (make-movieinfo Str Nat Str) pause
```

```
(define-struct tvseries (title eps len-per))
```

```
:: A TVSeries is a (make-tvseries Str Nat Nat) pause
```

```
:: A StreamingVideo is one of:
```

```
:: * a MovieInfo or
```

```
:: * a TVSeries.
```

# The template for **StreamingVideo**

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

```
:: streamingvideo-template: StreamingVideo → Any
```

```
(define (streamingvideo-template info)
```

```
  (cond [(movieinfo? info) ... ]
```

```
        [else ... ]))
```

We use type predicates in our questions.

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

:: StreamingVideo-template: StreamingVideo → Any

```
(define (streamingvideo-template info)
  (cond [(movieinfo? info)
         (... (movieinfo-director info) ...
              (movieinfo-title info) ...
              (movieinfo-duration info) ...
              (movieinfo-genre info) ... ) ]
        [else
         (... (tvseries-title info) ...
              (tvseries-eps info) ...
              (tvseries-len-per info) ... ) ]]))
```

# An example: StreamingVideo

:: (streamingvideo-title info) produces title of info

:: streamingvideo-title: StreamingVideo → Str

:: Examples:

(check-expect (streamingvideo-title

  (make-movieinfo "Spielberg" "ET" 115 "Sci-Fi")) "ET")

(check-expect (streamingvideo-title

  (make-tvseries "Friends" 236 22)) "Friends")

(define (streamingvideo-title info) ...)

# The definition of streamingvideo-title

```
(define (streamingvideo-title info)
  (cond
    [(movieinfo? info) (movieinfo-title info)]
    [else (tvseries-title 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 or values, we will use the notation

(**anyof** type1 type2 ... typeK v1 ... vT)

For example, if we hadn't defined **StreamingVideo** as a new type, we could have written the contract for **streamingvideo-title** as

:: streamingvideo-title: (anyof MovieInfo TVSeries) → 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
```

An example of a DoubleFeature is

```
(define classic-movies
```

```
  (make-doublefeature
```

```
    (make-movieinfo "Welles" "Citizen Kane" 119 "Drama")
```

```
    (make-movieinfo "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.