Mixed data
Mixed data

Racket provides predicates to identify data types, such as `number?` and `symbol?`.

`define-struct` also creates a predicate that tests whether its argument 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 rating))
;; A MovieInfo is a (make-movieinfo Str Str Num)

;; An MMInfo is one of:
;; * a SongInfo
;; * a MovieInfo

Here “MM” is an abbreviation for “multimedia”.
The template for mminfo

The template for mixed data is a \texttt{cond} with each type of data, and if the data is a structure, we apply the template for structures.

\begin{verbatim}
;; mminfo-template: MMInfo -> Any
(define (mminfo-template info)
 (cond
   [[(songinfo? info) ; if info is of type SongInfo
      (...)
      ...(songinfo-performer info)
      ...(songinfo-title info)
      ...(songinfo-genre info)]
    [(movieinfo? info) ; if info is of type MovieInfo
      (...)
      ...(movieinfo-director info)
      ...(movieinfo-title info)
      ...(movieinfo-rating info))]])
\end{verbatim}
(define favsong (make-songinfo "Kris Kristofferson" "Why Me Lord?" 'Country))

(define favmovie (make-movieinfo "Orson Welles" "Citizen Kane" 98))

;; (mminfo-artist info) produces performer/director name from info
;; mminfo-artist: MMInfo -> Str
;; Examples:
;; (check-expect (mminfo-artist favsong) "Kris Kristofferson")
;; (check-expect (mminfo-artist favmovie) "Orson Welles")
(define (mminfo-artist info)
  (cond
    [(songinfo? info) (songinfo-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 `SongInfo` and `MovieInfo`, there is no `define-struct` expression associated with `MMInfo`.

For the contract
```smaller
;; 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
```smaller
;; mminfo-artist: (anyof SongInfo MovieInfo) -> Str
```
Consider the following function:

```
(define (f x)
  (cond
    [(posn? x) x]
    [(songinfo? x) x]
    [(symbol? x) 15]))
```

Which of the following would be the most appropriate contract for f?

A. \( f : (\text{anyof} \text{ Posn} \text{ SongInfo} \text{ Sym}) \rightarrow \text{Num} \)
B. \( f : \text{Any} \rightarrow \text{Num} \)
C. \( f : (\text{anyof} \text{ Posn} \text{ SongInfo} \text{ Sym}) \rightarrow (\text{anyof} \text{ Posn} \text{ SongInfo} \text{ Num}) \)
D. \( f : \text{Any} \rightarrow \text{Any} \)
E. \( f : (\text{anyof} \text{ Posn} \text{ SongInfo} \text{ Sym}) \rightarrow (\text{anyof} \text{ Posn} \text{ SongInfo} \text{ Sym} \text{ Num}) \)
Consider the following function:

```
(define (f x)
  (cond
    [(posn? x) x]
    [(songinfo? x) x]
    [(symbol? x) 15]))
```

Which of the following would be the most appropriate contract for \( f \)?

A. \( f: \) \((\text{anyof Posn SongInfo Sym}) \rightarrow \text{Num}\)
B. \( f: \) \(\text{Any} \rightarrow \text{Num}\)
C. \( f: \) \((\text{anyof Posn SongInfo Sym}) \rightarrow (\text{anyof Posn SongInfo Num})\)
D. \( f: \) \(\text{Any} \rightarrow \text{Any}\)
E. \( f: \) \((\text{anyof Posn SongInfo Sym}) \rightarrow (\text{anyof Posn SongInfo Sym Num})\)
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.
Lists vs. structures
Lists versus structures

In M08 we wrote a Payroll data definition for processing tax withholdings:

```plaintext
;; A Payroll is one of:
;; * empty
;; * (cons (list Str Num) Payroll)
```

Example payroll:

```plaintext
(list (list "Asha" 50000)
     (list "Joseph" 100000)
     (list "Sami" 10000))
```
We also developed a corresponding template:

```scheme
;;; (payroll-template pr)
;;; payroll-template: Payroll -> Any
(define (payroll-template pr)
  (cond
   [(empty? pr) ...]
   [(cons? pr) (... ...)
    ...(first (first pr))
    ...(first (rest (first pr)))
    (payroll-template (rest pr)))]))
```
;; (name 1st) produces the 1st item from lst (name).
(define (name 1st) (first 1st))

;; (amount 1st) produces the 2nd item from lst (amount).
(define (amount 1st) (first (rest 1st)))

;; (payroll-template pr)
;; payroll-template: Payroll -> Any
(define (payroll-template pr)
  (cond
   [(empty? pr) ...]
   [(cons? pr) (...[[(name (first pr))]...(amount (first pr))](payroll-template (rest pr)))])))
Payroll with structures

We can accomplish the same goals with structures.

(define-struct payroll (name amount))

;; A Payroll is a (make-payroll Str Num)

(define (payroll-template pr)
  (cond
   [(empty? pr) ...]
   [(cons? pr) (...(payroll-name (first pr))
       ...(payroll-amount (first pr))
       (payroll-template (rest pr))))]]))

When should each of these two approaches be used?
Why use lists containing lists?

The name-list function will produce a list of names from either a Payroll or a TaxOwed.

If we used structures, we would require two different (but very similar) functions or extra complexity in the same function to distinguish which structure selector to use.

We will exploit this ability to reuse code written to use “generic” lists when we discuss abstract list functions later in the course.
Why use structures?

Structure is often present in a computational task, or can be defined to help handle a complex situation.

Using structures helps avoid some programming errors (e.g., accidentally extracting a list of salaries instead of names).

Structures automatically create the selector functions we needed to make the list-based code readable.

Our design recipes can be adapted to give guidance in writing functions using complicated structures.

Structures are provided in all mainstream programming languages.
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 the similar uses of structures and fixed-size lists, and be able to write functions that consume either type of data.