CS 135 Fall 2019

Tutorial 6: Structures
Goals of this tutorial

You should be able to...

- Understand how to construct and select from \textit{structures}.
- Utilize \textit{accumulative recursion}.
- Create \textit{templates} for user-defined \textit{structures}.
Review - Constructors and Selectors

- **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

Examples:

```scheme
(define mypoint (make-posn 8 1))
(posn-x mypoint) ⇒ 8
(posn-y mypoint) ⇒ 1
```
Review - Structure Predicate

`posn?` will return `true` if the parameter given is a `posn`, and `false` otherwise.

contract:

`;; posn?: Any → Bool`

Examples:

```
(posn? (make-posn 5 4)) ⇒ true
(posn? "posn") ⇒ false
(posn? (make-posn 'tutorial "three") ⇒ true
```

Structure predicates will determine **only** if the provided parameter uses the constructor function.

It will not check if the data inside the structure are of the correct type.
Review - Posn

Possible uses:

- coordinates of a point on a two-dimensional plane
- positions on a screen or in a window
- a geographical position

Note:

- An expression such as (make-posn 8 1) is considered a value, which will not be simplified further by the Stepper or our semantic rules.
- The expression (make-posn (+ 4 4) (− 3 2)) would be simplified further to (eventually) (make-posn 8 1).
Review - User-defined Structures

The following structures represent a card and a hand of 3 cards:

(define-struct card (suit value))

;; A Card is a (make-card Sym Nat)

;; requires: suit is (anyof 'hearts 'spades 'clubs 'diamonds)

;; 1 <= value <= 13

(define-struct hand (card1 card2 card3))

;; A Hand is a (make-hand Card Card Card)
Review - Making A Card/Hand

These structures behave just like posn, which we looked at earlier:

```scheme
(define first-card (make-card 'hearts 3))
(define second-card (make-card 'spades 1))
(define third-card (make-card 'clubs 12))
(define my-hand (make-hand first-card second-card third-card))

(card-suit first-card) ⇒ 'hearts
(card-value (hand-card1 my-hand)) ⇒ 3
```
Problem 1: card-template

Write a template function for Card called card-template.

```
(define-struct card (suit value))
;; A Card is a (make-card Sym Nat)
;; requires: suit is (anyof ’hearts ’spades ’clubs ’diamonds)
;; 1 <= value <= 13
```

Note: Do not comment out the template. If the template function has been written correctly using “...”, then there will be no errors when it is run.

Black highlighting inside of templates are ignored by our scripts, so you do not have to worry about losing marks due to black highlighting inside of templates.
Problem 2: card-score

A card has a score given by the following rules:

- Hearts score 5 points
- Diamonds score 4 points
- Spades score 0 points
- Clubs score -5 points
- A card scores points equal to the sum of its value and its suit score

Write a function called card-score that consumes a card and produces its score according to the rules above.

(check-expect (card-score (make-card 'hearts 13)) 18)
Problem 2: card-score

(define-struct card (suit value))
;; A Card is a (make-card Sym Nat)
;; requires: suit is one of ('hearts 'spades 'clubs 'diamonds)
;; 1 <= value <= 13

;; Constants
(define hearts-score 5)
(define diamonds-score 4)
(define spades-score 0)
(define clubs-score -5)
Problem 3: hand-score

Next, write a function hand-score that consumes a Hand and produces the sum of only the positive card scores in the hand. Include a purpose, contract, and examples (no tests).

(define-struct hand (card1 card2 card3))

;; A Hand is a (make-hand Card Card Card)
Problem 3: hand-score

;;; (hand-score hand) produces the sum of the positive card scores in hand
;;; hand-score: Hand → Nat
;;; Example:
(check-expect (hand-score my-hand) 0)
Problem 4: who-won

We can represent chess pieces in DrRacket with the following structure:

```
(define-struct piece (colour value))
```

;; A Piece is a (make-piece Sym Nat)

;; requires: colour is (anyof 'white 'black)

Note that each chess piece has a value associated with it.

;; Chess Piece Values:
(define pawn 1)
(define bishop 3)
(define knight 3)
(define rook 5)
(define queen 9)

;; Creating chess pieces
(define white-rook (make-piece 'white rook))
(define white-pawn (make-piece 'white pawn))
(define white-queen (make-piece 'white queen))
(define black-knight (make-piece 'black knight))
(define black-queen (make-piece 'black queen))
Problem 4: who-won

If a draw occurs in a game of chess, the tie can be broken by comparing the sum of the values of each captured piece.

Write a function who-won that consumes a list of captured chess pieces ((listof Piece)) and produces 'white if white wins the draw, 'black if black wins the draw, or 'draw if the comparison results in another draw.

(check-expect (who-won (list black-knight white-queen black-knight))
  'black)
Problem 4: who-won

To help us figure out who won, we need to sum up the values for both the black and white pieces. We can do this in three ways:

- Use simple recursion (process the list twice)
- Use accumulative recursion (process the list twice)
- Use accumulative recursion (process the list once)

Create helper functions called `sum-pieces`, `sum-pieces/acc`, and `sum-pieces/acc2` that sum up the values for both the black and white pieces using simple recursion (processes the list twice), accumulative recursion (processes the list twice), and accumulative recursion again (processes the list once), respectively.
sum-pieces - Accumulative Recursion

;; (sum-pieces/acc chess-pieces colour value-so-far) Produces value-so-far, which
;; represents the sum of the values of the pieces of the given colour in chess-pieces
;; sum-pieces/acc: (listof Pieces) (anyof 'white 'black) Nat → Nat

;; Examples:
(check-expect (sum-pieces/acc (list white-rook white-pawn black-knight) 'white 0) 6)
sum-pieces - Accumulative Recursion 2

;; (sum-pieces/acc2 chess-pieces white-value black-value) Produces 'black, white, or ’draw depending on white-value and black-value gotten from chess-pieces
;; sum-pieces/acc2: (listof Pieces) Nat Nat → (anyof ’black ’white ’draw)
;; Examples:
(check-expect (sum-pieces/acc2 (list white-rook white-pawn black-knight) 0 0) ’black)
Problem 4: who-won

Now that we’ve finished writing our helper functions, write the function who-won using either sum-pieces, sum-pieces/acc, or sum-pieces/acc2.

(check-expect (who-won (list white-rook white-pawn black-knight)) 'black)
Problem 4: who-won - Simple Recursion

;; (who-won chess-pieces) Produces 'white, 'black, or 'draw
;; depending on the values of the captured pieces in chess-pieces.
;; who-won: (listof Pieces) → Sym
;; Example:
(check-expect (who-won (list white-rook white-pawn black-knight)) 'black)
(check-expect (who-won (list white-rook black-queen)) 'white)
Problem 4: who-won - Simple Recursion

(define (who-won chess-pieces)
  (cond [(> (sum-pieces chess-pieces 'white) (sum-pieces chess-pieces 'black)) 'black]
        [(> (sum-pieces chess-pieces 'black) (sum-pieces chess-pieces 'white)) 'white]
        [else 'draw]))

;; Tests:
(check-expect (who-won (list white-queen black-queen)) 'draw)
Problem 4: who-won - Accumulative Recursion

(define (who-won chess-pieces)
  (cond 
    [ (> (sum-pieces/acc chess-pieces 'white 0))
      (sum-pieces/acc chess-pieces 'black 0))
      'black]
    [ (> (sum-pieces/acc chess-pieces 'black 0))
      (sum-pieces/acc chess-pieces 'white 0))
      'white]
    [else 'draw]])
Problem 4: who-won - Accumulative Recursion 2

(define (who-won chess-pieces)
  (sum-pieces/acc2 chess-pieces 0 0))