CS 135 Winter 2020

Tutorial 6: Accumulative Recursion

Announcements

- Midterm will start at 7pm next Monday.
- There is a midterm help session in STC0060 from 10:00-11:30 tomorrow.
- We are offering extra office hour from 1:30 to 2:30 today.
- Assignment 6 is due on **Tuesday, March 10**, at **9:00 PM**.
- Office hour time poll will be closed TODAY.
- Office hours will be changed starting next week.

Goals of this tutorial

You should be able to...

- Utilize **accumulative recursion**.
- Understand how to construct and select from **structures**.
- Create **templates** for **structures**.

Clicker Question: Debugging Accumulative recursion

Will the below function always produce the correct result?

```
;; (sum-greater-than-ten lon sofar) produces the sum of numbers in lon that is greater tha
;; sum-greater-than-ten: (listof Nat) Nat→ Nat
(define (sum-greater-than-ten lon sofar)
    (cond
        [(empty? lon) 0]
        [(> (first lon) 10) (sum-greater-than-ten (rest lon) (+ (first lon) sofar))]
        [else (sum-greater-than-ten (rest lon) sofar)]))
```

A Yes

B No

Review - Accumulative Recursion

- **Accumulative Recursion** is a special type of **Recursion** where the result so far is passed by as a parameter known as **Accumulator**.

- Usually a wrapper function is required to ”hide” the **Accumulator**

Problem 1: factorial

In assignment 4, you were asked to debug a function that produces the factorial for a natural number n:

```
(define (factorial n)
        (cond [(zero? n) 1]
              [else (* n (factorial (sub1 n)))]))
```

Write a function factorial2 that behaves exactly like factorial, but instead uses **Accumulative Recursion**

Problem 1: Design Recipe

```
;; (factorial2 n) Produces the factorial of n.
;; factorial2: Nat → Nat
;; requires: n > 0
;; Example:
(check-expect (factorial2 5) 120)
(check-expect (factorial2 0) 1)

(define (factorial2 n) ...)

;; Test:
(check-expect (factorial2 1) 1)
```

Problem 2: Accumulate Primes

Use accumulative recursion to write a function called primes-up-to that produces a list of all prime numbers from 2 to a given natural number n. Use the following code in your solution:

```
;; (has-factor? n lon) Determine whether the list lon
;; contains at least one factor of n.
;; has-factor?: Nat (listof Nat) → Bool
(define (has-factor? n lon)
        (cond [(empty? lon) false]
              [else (or (= 0 (remainder n (first lon)))
                        (has-factor? n (rest lon)))]))
```

Problem 2: Accumulate Primes Design Recipe

```
;; (primes-up-to/acc i n primes-so-far) Produces
;; a list of all primes less than or equal to n,
;; given the list of primes-so-far less than i
;; primes-up-to/acc: Nat Nat (listof Nat) → (listof Nat)
;; requires: primes-so-far is the list of primes < i
;; Example:
(check-expect (primes-up-to/acc 2 7 empty) (list 7 5 3 2))

(define (primes-up-to/acc i n primes-so-far)...)
```

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
```

DrRacket automatically creates three (types of) functions whenever a new structure is defined:

- Constructor
- Selector(s)
- Predicate

Review - Making A Card (Constructor)

constructor function: make-card

with contract:

;; make-card: Sym Nat $\rightarrow$ Card

- An expression such as (make-card 'hearts 3) is considered a value, which will not be simplified further by the Stepper or our semantic rules.
- The expression (make-card 'hearts (+ 2 1)) would be simplified further to (eventually) (make-card 'hearts 3).

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

Review - Structure (Selectors)

selector functions: card-suit and card-value

with contracts:

;; card-suit: Card $\rightarrow$ Sym

;; card-value: Card $\rightarrow$ Nat

Examples:

```
(define first-card (make-card 'hearts 3))
```

(card-suit first-card) $\Rightarrow$ 'hearts

(card-value first-card) $\Rightarrow$ 3

Review - Structure (Predicate)

card? will return true if the parameter given is a Card, and false otherwise.

contract:

;; card?: Any → Bool

Examples:

(card? (make-card 'hearts 3)) ⇒ true

(card? first-card) ⇒ true

(card? "card") ⇒ false

(card? (make-card '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.

Problem 3: 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 3: card-template

```
;; card-template: Card → Any
(define (card-template card)
        (... (card-suit card) ...
         ... (card-value card) ...))
```

Extra Practice: 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)
```

Extra Practice: 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)
```