CS450

Structure of Higher Level Languages

Lecture 4: Recursion, nested definitions

Tiago Cogumbreiro

Exercises with lists

A template of recursion

Copying a list

```
(define (copy l)
  (match l
    [(list)
     ; input = '()
     (list)
     ; expected output = '()
    ]
    [(list h l ...) ; input = (list 1 2 3)
      ; h = 1
      ; l = (list 2 3)
      (define result (copy l))
      ; result = (list 2 3)
      (cons h result)
      ; expected output = (list 1 2 3)
    ]))
```

Notes

- `(define (copy l) l)` would be a simpler solution, but we're trying to illustrate a **recursion pattern**.
- Remember to write the expected inputs and outputs for a given example
- **Part in yellow:** given the expected inputs, how would you build the expected output?

Lists: example 1

Summation of all elements of a list

Spec

```
(require rackunit)
(check-equal? 10 (sum-list (list 1 2 3 4)))
(check-equal? 0 (sum-list (list)))
```

Lists: example 1

Summation of all elements of a list

Solution

```
(define (sum l)
  (match l
    [(list) ; input = '()
      0]     ; expected output = 0
    [(list h l ...) ; input = '(1 2 3 4)
      ; h = 1
      ; l = '(2 3 4)
      ; result = (sum (list 2 3 4)) = 2 + 3 + 4 = 9
      (define result (sum l))
      (+ h result)
      ])) ; expected output = (+ 1 9) = 10
```

Notes

- We use the same pattern as in `copy`
- Again, the expression being returned combines `h` and `result`

Spot the error! (#1)

```
#lang racket
(define (sum-list l)
  (match l
    [empty 0]
    [(list h l ...) (+ h (sum-list l))]))
```

Spot the error! (#1)

```
#lang racket
(define (sum-list l)
  (match l
    [empty 0]
    [(list h l ...) (+ h (sum-list l))]))
```

- For `match` consider `empty` to be defined as `(define empty (list))`, not as a keyword
- Pattern `empty` means: anything you find assign it to a variable called `empty`; same as writing `[x 0]`
- The first branch matches with anything you give it, so this function never recurses

Spot the error! (#2)

```
#lang racket
(define (sum-list l)
  (match l
    [(list) 0]
    [(list h t ...) (+ h (sum-list l))]))
```

Spot the error! (#2)

```
#lang racket
(define (sum-list l)
  (match l
    [(list) 0]
    [(list h t ...) (+ h (sum-list l))]))
```

- We wanted to recurse on `t`, but instead recursed on the original list `l`
- This leads to an **infinite loop**
- **Good practice:** use `l` as the rest of the list, and make this error impossible.

Spot the error! (#3)

```
#lang racket
(define (sum-list l)
  (match l
    [(list) 0]
    [(h l ...) (+ h (sum-list l))]))
```

Spot the error! (#3)

```
#lang racket
(define (sum-list l)
  (match l
    [(list) 0]
    [(h l ...) (+ h (sum-list l))]))
```

- We forgot to specify the data-type `list` in the second pattern
- Racket will raise an exception notifying us that the pattern is incorrect

```
ex.rkt:5:5: match: syntax error in pattern
  in: (h l ...)
  location...:
   ex.rkt:5:5
```

Lists: example 2

Returns a list from n down to 1

Spec

```
(require rackunit)
(check-equal? (list) (count-down 0))
(check-equal? (list 3 2 1) (count-down 3))
```

Lists: example 2

Returns a list from n down to 1

Spec

```
(require rackunit)
(check-equal? (list) (count-down 0))
(check-equal? (list 3 2 1) (count-down 3))
```

Solution

```
#lang racket
(define (count-down n)
  (cond [(<= n 0) (list)]
        [else (cons n (count-down (- n 1)))]))
```

Lists: example 3

Point-wise pairing of two lists

Spec

```
(require rackunit)
(check-equal? (list (cons 3 30) (cons 2 20) (cons 1 10))
              (zip (list 3 2 1) (list 30 20 10)))
(check-equal? (list (cons 3 30) (cons 2 20) (cons 1 10))
              (zip (list 3 2 1) (list 30 20 10 5 4 3 2 1)))
(check-equal? (list (cons 3 30) (cons 2 20) (cons 1 10))
              (zip (list 3 2 1 90 180 270) (list 30 20 10)))
```

Lists: example 3

Point-wise pairing of two lists

Lists: example 3

Point-wise pairing of two lists

Solution

```
#lang racket
(define pair list) ; Encode a pair as a lis
(define (zip l1 l2)
```

Lists: example 3

Point-wise pairing of two lists

Solution

```
#lang racket
(define pair list) ; Encode a pair as a lis
(define (zip l1 l2)
```

```
  (match* (l1 l2)
    [((list) _) (list)]
    [(_ (list)) (list)]
    [((list h1 l1 ...) (list h2 l2 ...))
      (cons
        (pair h1 h2)
        (zip l1 l2))]))
```

- Use `match*` to pattern match two values at once

Using nested definitions

Build a list from 1 up to n

> Our goal is to build a list from 1 up to some number. Here is a template of our function and a test case for us to play with. For the sake of simplicity, we will not handle non-positive numbers.

```
#lang racket
(define (countup-from1 x) #f)

(require rackunit)
(check-equal? (list 1) (countup-from1 1))
(check-equal? (list 1 2) (countup-from1 2))
(check-equal? (list 1 2 3 4 5) (countup-from1 5))
```

> Hint: write a helper function `count` that builds counts from `n` up to `m`.

Exercise 1: attempt #1

- We write a helper function `count` that builds counts from `n` up to `m`.

```
#lang racket
(define (countup-from1 x)
  (count 1 x))

(define (count from to)
  (cond
    [(equal? from to) (list to)]
    [else (cons from (count (+ 1 from) to))]))
```

Exercise 1: attempt #1

- We write a helper function `count` that builds counts from `n` up to `m`.

```
#lang racket
(define (countup-from1 x)
  (count 1 x))

(define (count from to)
  (cond
    [(equal? from to) (list to)]
    [else (cons from (count (+ 1 from) to))]))
```

Let us refactor the code and hide function `count`

Exercise 1: attempt #2

> We move function `count` to be internal to function `countup-from1`, as it is a helper function and therefore it is good practice to make it ***private*** to `countup-from1`.

```
(define (countup-from1 x)
  ; Internally defined function, not visible from
  ; the outside
  (define (count from to)
    (cond [(equal? from to) (list to)]
          [else (cons from (count (+ 1 from) to))]))
  ; The same call as before
  (count 1 x))
```

When to nest functions?

Nest functions:

- If they are unnecessary outside
- If they are under development
- If you want to hide them: **Every function in the public interface of your code is something you'll have to maintain!**

Intermission:

Nested definitions

Nested definition: local variables

> Nested definitions bind a variable within the body of a function and are only visible within that function (these are local variables)

```
#lang racket
(define (f x)
  (define z 3)
  (+ x z))

(+ 1 z) ; Error: z is not visible outside function f
```

Nested definitions shadow other variables

- Nested definitions silently shadow any already defined variable

```
#lang racket
(define z 10)
(define (f x)
  (define x 3) ; Shadows parameter
  (define z 20)  ; Shadows global
  (+ x z))

(f 1) ; Outputs 23
```

No redefined local variables

- It is an error to re-define local variables

```
#lang racket
(define (f b)
  ; OK to shadow a parameter
  (define b (+ b 1))
  (define a 1)
  ; Not OK to re-define local variables
  ; Error: define-values: duplicate binding name
  (define a (+ a 1))
  (+ a b))
```

Back to Exercise 1

Exercise 1: attempt #2

> Notice that we have some redundancy in our code. In function `count`, parameter `to` remains unchanged throughout execution.

```
(define (countup-from1 x)
  ; Internally defined function, not visible from
  ; the outside
  (define (count from to)
    (cond [(equal? from to) (list to)]
          [else (cons from (count (+ 1 from) to))]))
  ; The same call as before
  (count 1 x))
```

Exercise 1: attempt #3

> We removed parameter `to` from function `count` as it was constant throughout the execution. Variable `to` is captured/copied when `count` is defined.

```
(define (countup-from1 to)
  ; Internally defined function, not visible from
  ; the outside
  (define (count from)
    (cond [(equal? from to) (list to)]
          [else (cons from (count (+ 1 from)))]))
  ; The same call as before
  (count 1))
```

Example 1: summary

- Use a nested definition to hide a function that is only used internally.
- Nested definitions can refer to variables defined outside the scope of their definitions.
- The last expression of a function's body is evaluated as the function's return value

Measuring performance

Example 2

Maximum number from a list of integers

Example 2: attempt 1

- Finding the maximum element of a list.

```
#lang racket
(define (max xs)
  (cond
    [(empty? xs) (error "max: expecting a non-empty list!")]
    [(empty? (rest xs)) (first xs)]               ; The list only has one element (the max)
    [(> (first xs) (max (rest xs))) (first xs)] ; The max of the rest is smaller than 1st
    [else (max (rest xs))]))                    ; Otherwise, use the max of the rest

; A simple unit-test
(require rackunit)
(check-equal? 10 (max (list 1 2 10 4 0)))
```

- We use function `error` to abort the program with an exception.

Example 2: attempt 1

> Finding the maximum element of a list.

Let us benchmark `max` with sorted list (worst-case scenario):

- 20 elements: 18.43ms
- 21 elements: 36.63ms
- 22 elements: 75.78ms

> Whenever we add an element we double the execution time. Why?

Example 2: attempt 1

> Whenever we hit the else branch (because we can't find the maximum), we re-compute the max element.

```
(define (max xs)
  (cond
    [(empty? xs) (error "max: expecting a non-empty list!")]
    [(empty? (rest xs)) (first xs)]                ; The list only has one element (the max)
    [(> (first xs) (max (rest xs))) (first xs)] ; The max of the rest is smaller than 1st
    [else (max (rest xs))]))                    ; Otherwise, use the max of the rest
```

Example 2: attempt 2

We use a local variable to cache a duplicate computation.

```
(define (max xs)
  (cond
    [(empty? xs) (error "max: expecting a non-empty list!")]
    [(empty? (rest xs)) (first xs)]
    [else
      (define rest-max (max (rest xs))) ; Cache the max of the rest
      (cond
        [(> (first xs) rest-max) (first xs)]
        [else rest-max])]))
```

- Attempt #1: 20 elements in 75.78ms
- Attempt #2: 1,000,000 elements in 101.15ms

Example 2 takeaways

- Use nested definitions to cache intermediate results
- Identify repeated computations and cache them in nested (local) definitions