Structure of Higher Level Languages

Lecture 6: Nested definitions; caching

Tiago Cogumbreiro
Today we will learn about...

1. tips for solving HW1
2. using nested definitions
3. measuring running time

Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington. (Video available)
Tips for solving HW1
HW1: Question 4

1. Do all parts except lambda?, define?, and define-func?.
2. Write lambda?
3. Write define-func?
4. Write define?

More tips

- Function application is simpler than it seems
- All acceptance-tests from define-func? should pass in define?
Racket spec

HW1: Question 4

```
program = #lang racket term*

term = definition | expression

definition = basic-def | function-def
basic-def = (define identifier expression)
function-def = (define (variable+) term+)

expression = value | variable | function-call | function-decl | ...
value = number | ...
function-call = (expression+)
function-decl = (lambda (variable+) term+)
```
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 \( m \).

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

(define (count from to)
  (cond
   [(= 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 \( m \).

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

(define (count from to)
  (cond
   [ (= 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)

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

(+ 1 \texttt{z}) \; \texttt{Error: z is not visible outside function f}
```
Nested definitions shadow other variables

Nested definitions silently shadow any already defined variable

```racket
#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

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

```scheme
(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.

```racket
#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. We use functions `first` and `rest` as synonyms for `car` and `cdr`, as it reads better.
Example 2: attempt 1

Finding the maximum element of a list.

Let us benchmark \texttt{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. \textbf{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