CS450

Structure of Higher Level Languages

Lecture 6: Functions as data-structures, currying

Tiago Cogumbreiro

Functions as data-structures

Exercises

Exercise 1

What is the output of this program?

```
(define x 10)
(define (f x)
  (+ x 20))
(f 30)
```

Exercise 1

What is the output of this program?

```
(define x 10)
(define (f x)
  (+ x 20))
(f 30)
```

Output: 50

Because, parameter `x` shadows the outermost definition.

Exercise 2

What is the output of this program?

```
(define x 10)
(define f (lambda (x) (+ x 20)))
(f 30)
```

Exercise 2

What is the output of this program?

```
(define x 10)
(define f (lambda (x) (+ x 20)))
(f 30)
```

Output: 50

The code above is **equivalent** to the code below:

```
(define (f x) (+ x 20))
```

Exercise 3

What is the output of this program?

```
(define (factory k)
  (lambda () k))

(factory 10)
```

Exercise 3

What is the output of this program?

```
(define (factory k)
  (lambda () k))

(factory 10)
```

Output: `#<procedure>`

Although if Racket displayed code, we would get: `(lambda () 10)`

```
((factory 10))
; Outputs: 10
```

Exercise 3

Step-by-step evaluation

```
(factory 10) =
( (lambda (k) (lambda () k)) 10) =
; ^^^^^^^^^^^^^^^^^^^^^^^^^^^\______ the value *bound* to factory
(lambda () 10)
```

Why is factory replaced by a lambda?

User input

```
(define (factory k)
  (lambda () k))
```

Internal representation

```
(define factory
  (lambda (k)
    (lambda () k)))
```

Exercise 3

Looking at function application more closely

```
(
  (lambda (k)        ; <- parameter k
    (lambda () k)) ; <- body of function
  10                 ; <- argument
)
; Remove outer lambda and replace each parameter by argument
; (lambda () k)       <- body of function
;              \___ replace parameter k by argument 10
(lambda () 10)        ; <- return value
```

The abstract syntactic tree (AST)

Exercises

Exercise

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))
g
```

Exercise

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))
g
```

Output: `(lambda (b) (cond [b 1] [else 2]))`

Q2: How do I call `g` to obtain `1`?

Exercise

Q1: What is the output of this program?

```
(define (f x y)
  (lambda (b)
    (cond [b x] [else y])))

(define g (f 1 2))
g
```

Output: `(lambda (b) (cond [b 1] [else 2]))`

Q2: How do I call `g` to obtain `1`?

Solution: `(g #t)`

Implementing a pair with functions alone

If we can capture one parameter, then we can also capture two parameter. **Let us implement a pair-data structure with only functions!** **source: SICP 2.1.3**

```
(define (pair l r)
  (lambda (op)  ; <- we use a parameter to choose which stored data to return
    (match op
      ; if the operation is 'left then return l
      ['left l]
      ; if the operation is 'right then return r
      ['right r])))

; We now define our accessors
(define (pair-left p) (p 'left))    ; Returns the first element of the pair
(define (pair-right p) (p 'right))  ; Returns the second element of the pair

(define p (pair 10 20))  ; Same as: (define (p op) (match op ['left 10] ['right 20]))
(pair-left p)            ; Returns 10 because (pair-left p) -> (p 'left) -> 10
(pair-right p)           ; Returns 20 because (pair-right p) -> (p 'right) -> 20
```

Functions as values

What is functional programming

Functional programming has different meanings to different people

- Avoid mutation
- **Using functions as values**
- A programming style that encourages recursion and recursive data structures
- A programming model that uses ***lazy*** evaluation (discussed later)

First-class functions

- **Functions are values:** can be passed as arguments, stored in data structures, bound to variables, ...
- **Functions for extension points:** A powerful way to factor out a common functionality

Functions as parameters

Functions as parameters

Monotonic increasing function (for one input)

> Function `monotonic?` takes a function `f` as a parameter and a value `x`, and then checks if `f` increases monotonically for a given `x`.

Example

```
#lang racket
(define (double n) (* 2 n))
(define (monotonic? f x)
  (>= (f x) x))
;; Tests
(require rackunit)
(check-true (monotonic? double 3))
(check-false (monotonic? (lambda (x) (- x 1)) 3))
```

How do we evaluate?

```
(monotonic? double 3)
```

Functions as parameters

Monotonic increasing function (for one input)

> Function `monotonic?` takes a function `f` as a parameter and a value `x`, and then checks if `f` increases monotonically for a given `x`.

Example

```
#lang racket
(define (double n) (* 2 n))
(define (monotonic? f x)
  (>= (f x) x))
;; Tests
(require rackunit)
(check-true (monotonic? double 3))
(check-false (monotonic? (lambda (x) (- x 1)) 3))
```

How do we evaluate?

```
  (monotonic? double 3)

= (>= (double 3) 3)
= (>= ((lambda (n) (* 2 n)) 3) 3)
= (>= (* 2 3) 3)
= (>= 6 3)
= #t
```

Functions as parameters

Recursively apply a function n-times

> Function `apply-n` takes a function `f` as parameter, a number of times `n`, and some argument `x`, and then recursively calls `(f (f (... (f x))))` an `n`-number of times.

```
#lang racket
(define (apply-n f n x)
  (cond [(<= n 0) x]
        [else (apply-n f (- n 1) (f x))]))
;; Tests
(require rackunit)
(define double (lambda (x) (* 2 x)))
(check-equal? (* 2 (* 2 (* 2 1))) (apply-n double 3 1))
(check-equal? (+ 3 (+ 3 (+ 3 1))) (apply-n (lambda (x) (+ 3 x)) 3 1))
```

Example apply-n

Let us unfold the following...

```
(apply-n double 3 1)                    ; (<= 3 0) = #f
```

Example `apply-n`

Let us unfold the following...

```
  (apply-n double 3 1)                    ; (<= 3 0) = #f

= (apply-n double (- 3 1) (double 1))
= (apply-n double 2 2)                    ; (<= 2 0) = #f
= (apply-n double (- 2 1) (double 2))
= (apply-n double 1 4)                    ; (<= 1 0) = #f
= (apply-n double (- 1 1) (double 4))
= (apply-n double 0 8)                    ; (<= 0 0) = #t
= 8
```

Functions as data-structures

Functions as data-structures

The following is a function that returns a constant value (returns 3 always):

```
(define three:2 (lambda () 3))

(three:2) ; <- We need to call the function to obtain its contents
          ;    Note that we are passing 0 parameters.
```

Note the difference...

The following is a variable binding (not a function!):

```
(define three:1 3)
```

Variable `three:1` ***evaluates*** to the number `3`.

UMass Boston

A factory of constant-return functions

We can generalize the procedure by creating a function that returns a new function declaration that returns a given parameter `n`.

```
(define (factory n)
  (lambda () n)) ; <-- calling 'factory' creates a new function dynamically
                 ;     each new function captures the given 'n',
                 ;     which changes according to how it was called

(define three:4 (factory 3)) ; <- same as: (define three:4 (lambda () 3))
(three:4) ; Returns: 3

(define four:1 (factory 4)) ; <- same as: (define four:1 (lambda () 4))
(four:1) ; Returns 4
```

Functions in data structures

Functions stored in data structures

"Freeze" one parameter of a function

> In this example, a `frozen` data-structure stores a binary-function and the first argument. Function `apply1` takes a frozen data structure and the second argument, and applies the stored function to the two arguments.

```
(struct frozen (func arg1) #:transparent)

(define (apply1 fr arg)
  (define func (frozen-func fr))     ; Bind a function to a local variable
  (define arg1 (frozen-arg1 fr))
  (func arg1 arg))                   ; Call a function bound to a local variable

(define frozen-double (frozen * 2)) ; Store function '*' in a data structure
(define (double x) (apply1 frozen-double x))
(check-equal? (* 2 3) (double 3))
```

Unfolding `(double 3)`

```
  (double 3)
= (apply1 frozen-double 3)
= (apply1 (frozen * 2) 3)
= (define fr (frozen * 2))
  ((frozen-func fr) (frozen-arg1 fr) 3)
= (* 2 3)
= 6
```

Functions stored in data structures

Apply a list of functions to a value

```
#lang racket
(define (double n) (* 2 n))
; A list with two functions:
; * doubles a number
; * increments a number
(define p (list double (lambda (x) (+ x 1))))
; Applies each function to a value
(define (pipeline funcs value)
  (match funcs
    [(list) value]
    [(list f funcs) (pipeline funcs (f value))]))
; Run the pipeline
(check-equal? (+ 1 (double 3)) (pipeline p 3))
```

Creating functions dynamically

Returning functions

Functions in Racket automatically capture the value of any variable referred in its body.

Example

```
#lang racket
(define (frozen-* arg1)
  (define (get-arg2 arg2)
    (* arg1 arg2))
  ; Returns a new function
  ; every time you call frozen-*
  get-arg2)
(require rackunit)
(define double (frozen-* 2))
(check-equal? (* 2 3) (double 3))
```

Evaluating `(frozen-* 2)`

```
  (frozen-* 2)
= (define (get-arg2 arg2) (* 2 arg2)) get-arg2
= (lambda (arg2) (* 2 arg))
```

Evaluating `(double 3)`

```
  (double 3)
= ((frozen-* 2) 3)
= ((lambda (arg2) (* 2 arg2)) 3)
= (* 2 3)
= 6
```

Exercise

String concatenation

```
(define (string-concat l) 'todo)

(check-equal?
  (string-concat (list "Hello" ", " "world" "!"))
  "Hello, world!")
```

String concatenation (solution)

```
(define (string-concat l)
  (match l
    [(list) ""]
    [(list h l ...) (string-append h (string-concat l))]
  ))

(check-equal?
  (string-concat (list "hello" ", " "world" "!"))
  "Hello, world!")
```

Currying functions

Revisiting "freeze" function

Freezing binary-function

```
(struct frozen (func arg1) #:transparent)

(define (apply1 fr arg)
  (define func (frozen-func fr))
  (define arg1 (frozen-arg1 fr))
  (func arg1 arg))

(define frozen-double (frozen * 2))
(define (double x) (apply1 frozen-double x))
(check-equal? (* 2 3) (double 3))
```

Attempt #1

```
(define (freeze f arg1)
  (define (get-arg2 arg2)
    (f arg1 arg2))
  get-arg2)

(define double (freeze * 2))
(check-equal? (* 2 3) (double 3))
```

> Our `freeze` function is more general than `freeze-*` and simpler than `frozen-double`. We abstain from using a data-structure and use Racket's variable capture capabilities.

Generalizing "frozen" binary functions

Attempt #2

```
(define (freeze f)
  (define (expect-1 arg1)
    (define (expect-2 arg2)
      (f arg1 arg2))
    expect-2)
  expect-1)

(define frozen-* (freeze *))
(define double (frozen-* 2))
(check-equal? (* 2 3) (double 3))
```

Evaluation

```
  (define frozen-* (freeze *))
= (define frozen-*
    (define (expect-1 arg1)
      (define (expect-2 arg2)
        (* arg1 arg2))
      expect-2)
    expect-1)

  (define double (frozen-* 2))
= (define double
    (define (expect-2 arg2) (* 2 arg2))
    expect-2)

  (double 3)
= (* 2 3)
```

Currying functions

Currying is the general technique of "freezing" functions with multiple parameters. It provides a way of delaying (and caching) the passage of multiple arguments by means of new functions.

> A curried function $\mathbf{curry}_{f,n,a}(x)$ is a unary function annotated with an uncurried function f arguments a and a number of expected arguments n that can be recursively defined as:

$$\mathbf{curry}_{f,n+1,[a_1,...,a_n]}(x) = \mathbf{curry}_{f,n,[a_1,...,a_n,x]}$$
$$\mathbf{curry}_{f,0,[a_1,...,a_n]}(x) = f(a_1,\ldots,a_n,x)$$

```
#lang racket
(define frozen-* (curry *))
(define double (frozen-* 2))
(require rackunit)
(check-equal? (* 2 3) (double 3))
```

Haskell Curry

Did you know?

- In some programming languages functions are curried by default. Examples include Haskell and ML.
- The term currying is named after Haskell Curry, a notable logician who developed combinatory logic and the Curry-Howard correspondence (practical applications include proof assistants).

  Haskell was born in Millis, MA (1 hour drive from UMB).

Source: public domain

UMass Boston

Uncurried functions

All arguments must be provided at call-time, otherwise error.

Python example

```
def add(l, r):
  return l + y

add(10)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: add() missing 1 required positional argument: 'r'
```

UMass Boston

Curried functions

> If we provide one argument to a 2-parameters function, the result is a 1-parameter function that expects the second argument.

Haskell example

```haskell
-- Define addition
add x y = x + y
-- Define adding 10 to some number
add10 = add 10
-- 10 + 30
add10 30
-- 40
```

Currying in Racket

Function `curry` **converts** an uncurried function into a curried function.

```
#lang racket
(define curried-add (curry +))
(define add10 (curried-add 10))
(require rackunit)
(check-equal? (+ 10 30) (add10 30))
```

Currying functions

Currying is the general technique of "freezing" functions with multiple parameters. It provides a way of delaying (and caching) the passage of multiple arguments by means of new functions.

> A curried function $\mathbf{curry}_{f,n,a}(x)$ is a unary function annotated with an uncurried function f arguments a and a number of expected arguments n that can be recursively defined as:

$$\mathbf{curry}_{f,n+1,[a_1,...,a_n]}(x) = \mathbf{curry}_{f,n,[a_1,...,a_n,x]}$$
$$\mathbf{curry}_{f,0,[a_1,...,a_n]}(x) = f(a_1, \ldots, a_n, x)$$

Exercise

What is the output of this program?

Program

```
(define curried-add
  (lambda (arg1)
    (lambda (arg2)
      (+ arg1 arg2))))

(define a (curried-add 10))
(define b (curried-add 20))
a
b
(a 30)
(b 40)
```

Exercise

What is the output of this program?

Program

```
(define curried-add
  (lambda (arg1)
    (lambda (arg2)
      (+ arg1 arg2))))

(define a (curried-add 10))
(define b (curried-add 20))
a
b
(a 30)
(b 40)
```

Output

```
(lambda (arg2) (+ 10 arg2))
(lambda (arg2) (+ 20 arg2))
40
60
```