CS450

Structure of Higher Level Languages

Lecture 09: Dynamically-created funcs, storing funcs in lists, currying

Tiago Cogumbreiro

Today we will learn...



- storing functions in data-structures
- creating functions dynamically
- currying functions

Section 2.2.1 in SICP. Try out the interactive version of section 2.2 of the SICP book.

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.

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)
  (cond [(empty? funcs) value]
        [else (pipeline (rest funcs) ((first funcs) 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
```

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)
= (*23)
```

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 $\operatorname{curry}_{f,n,a}(x)$ 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:

```
egin{aligned} \operatorname{curry}_{f,n+1,[a_1,\ldots,a_n]}(x) &= \operatorname{curry}_{f,n,[a_1,\ldots,a_n,x]} \ \operatorname{curry}_{f,0,[a_1,\ldots,a_n]}(x) &= f(a_1,\ldots,a_n,x) \end{aligned}
```

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

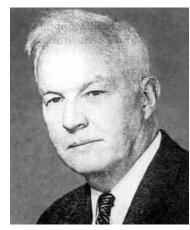
Haskell Curry



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