Structure of Higher Level Languages

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

Tiago Cogumbreiro
Today we will learn...

- academic honesty policy
- 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.
Academic dishonesty
Plagiarism in University

Copying code from others is wrong because:

- you do not learn
- you risk being expelled
- you are risking the other person being expelled
- you risk not completing your degree
- you risk being put on a list of cheaters (other universities may reject your application)
Plagiarism in the Industry

Is wrong, because:

- it is illegal
- you risk being dismissed from employment
- you risk being sued
Copying code (when it is right)

- software licenses define clear rules on how you can copy, use, and change other people's code
- open source promotes sharing of code
  - attribution is important (unless public domain)
  - good way to land on a job
Plagiarism in CS 450

Zero Tolerance

- student's responsibility to learn the Student's code of conduct
- we use plagiarism detection *(renaming functions is not enough)*
- we compare against solutions from past years (and instructor)
- be careful when working with others, any sharing code may trigger
- the plagiarism detection tool can detect code sharing among students
Plagiarism in CS 450

Last call

- statistically, there will be plagiarism this semester
- if I contact you regarding plagiarism, there will be zero tolerance (no second chances)
- you *may* void a submission before I contact you without repercussions
- I give you enough time to complete assignments, there is no excuse
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 \((\text{double } 3)\)

\[
\begin{align*}
(\text{double } 3) &= (\text{apply1 } \text{frozen-double } 3) \\
&= (\text{apply1 } (\text{frozen } * 2) 3) \\
&= (\text{define fr (frozen } * 2))
\quad ((\text{frozen-func fr}) (\text{frozen-arg1 fr}) 3) \\
&= (* 2 3) \\
&= 6
\end{align*}
\]
Functions stored in data structures

Apply a list of functions to a value

```racket
#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

```racket
#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)
= (* 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 \( \text{curry}_{f,n,a}(x) \) 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:

\[
\begin{align*}
\text{curry}_{f,n+1,[a_1,\ldots,a_n]}(x) &= \text{curry}_{f,n,[a_1,\ldots,a_n,x]} \\
\text{curry}_{f,0,[a_1,\ldots,a_n]}(x) &= f(a_1,\ldots,a_n,x)
\end{align*}
\]

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