CS450

Structure of Higher Level Languages

Lecture 5: Structs, functions as values, and currying

Tiago Cogumbreiro

Deadlines are final!

1. **Email requests for deadline extensions a few days before the deadline will be ignored.**
2. You have around 14 days to work on each homework assignment.
   Being unable to work for 3 days is no excuse to demand an extension.
3. Allowing extensions, would just compound the work with the following assignment.
4. In the case of an exceptional event, contact me as soon as possible. (See point 1.)

Tip #1: avoid fighting the autograder

1. **It's not personal:** The autograder is not against you
2. **It's not picky:** The autograder is not against one specific solution
3. **Correlation is not causation:** Having a colleague with the same problem as you have, does **not** imply that the autograder is wrong
4. **Spend your time wisely**: don't spend it thinking the autograder is wrong

Instead, discuss

1. **Use the autograder for your benefit:** submit solution to test your hypothesis
2. **Think before resubmitting:** try explaining your solution to someone
3. **Ask before resubmitting:** write test cases and discuss those test cases with others

10% of your grade is participation, so discuss!

Tip #2: participate

10% of your grade is participation

> Software engineering and academic life is about *communication*: you are expected to interact to solve your homework assignments.

1. Exercises are explained succinctly on purpose: **ask questions** to know more
2. Exercises have few test cases on purpose: **share test-cases** to know more

Make time in your schedule to interact

Tip #3: time management

Work on your homework assignment incrementally

- after each class you can solve a new exercise (with few exceptions)
- when you get stuck in an exercise: (1) **ask** in our forum, and while you are waiting (2) **continue working** on other exercises
- don't leave everything to the weekend before submission

Tip #4: Do **not** use car/cdr for list manipulation

- Reserve `car/cdr` to pairs only. **This family of functions leads to subtle bugs due to typos.**

Bad

```
(and
  ; ...
  (list? (cadr node))
  (andmap symbol? (cdr node)))
```

Good

```
(and
  ; ...
  (list? (second node))
  (andmap symbol? (rest node)))
```

- Can you spot the bug in this excerpt from `define-func?`?

Tip #4: Do **not** use car/cdr for list manipulation

Reserve `car/cdr` to pairs only. **This family of functions leads to subtle bugs due to typos.**

Bad

```
(and
  ; ...
  (list? (cadr node))
  (andmap symbol? (cdr node)))
```

Good

```
(and
  ; ...
  (list? (second node))
  (andmap symbol? (rest node)))
```

Can you spot the bug in this excerpt from `define-func?`? The code should be

```
(andmap symbol? (second node))
```

Tip #5: avoid using `if`

Bad-style uses of `if`/`cond`

1. We are not covering `if` in this course.
2. Most of you are holding it in the wrong way.

> Whenever you write
>
> ```
> (if condition foo #f)
> ```
>
> or
>
> ```
> (cond [condition foo] [else #f])
> ```
>
> rewrite it to
>
> ```
> (and condition foo)
> ```
>
> The `and`-version is simpler and more concise.

Today we will...

- Learn the use of structs to create data structures (exercise 5 of HW2)
- Implement an AST using structs (exercise 5 of HW2)
- Introduce functions as values (exercise 1 of HW2)
- Currying (exercise 4 of HW2)

> Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington.

Revisiting user data structures

User data structures

Recall the 3D point from Lecture 3

```
; Constructor
(define (point x y z) (list x y z))
; Accessors
(define (point-x pt) (first pt))
(define (point-y pt) (second pt))
(define (point-z pt) (third pt))
```

And the name data structure

```
; Constructor
(define (name f m l) (list f m l))
; Accessor
(define (name-first n) (first n))
(define (name-middle n) (second n))
(define (name-last n) (third n))
```

How do we prevent such errors?

```
(define p (point 1 2 3))
(name-first p) ; This should be an error, and instead it happily prints 1
```

Introducing struct

UMASS BOSTON

```
#lang racket
(require rackunit)
(struct point (x y z) #:transparent)
(define pt (point 1 2 3))
(check-equal? 1 (point-x pt))  ; the accessor point-x is automatically defined
(check-equal? 2 (point-y pt))  ; the accessor point-y is automatically defined
;
(struct name (first middle last))
(define n (name "John" "M" "Smith"))
(check-equal? "John" (name-first n))
(check-true (name? n))    ; We have predicates that test the type of the value
(check-false (point? n)) ; A name is not a point
(check-false (list? n))  ; A name is not a list
; (point-x n) ;; Throws an exception
; point-x: contract violation
;    expected: point?
;    given: #<name>)
```

Bennefits of using structs

- Reduce boilerplate code
- Ensure type-safety

Implementing Racket's AST

Grammar

```
expression = value | variable | apply | define
value = number | void | lambda
apply = ( expression+ )
lambda = ( lambda ( variable* ) term+)
```

Implementing values

```
value = number | void | lambda
lambda = ( lambda ( variable* ) term+)
```

Implementing values

value = *number* | *void* | *lambda*
lambda = (lambda (*variable**) *term*+)

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

> We are using a prefix `r:` because we do not want to redefined standard-libary definitions.

Implementing expressions

```
expression = value | variable | apply
apply = ( expression+ )
```

Implementing expressions

expression = *value* | *variable* | *apply*
apply = (*expression+*)

```
(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

> In `r:apply` we distinguish between the expression that represents the function `func`, and the (possibly empty) list of arguments `args`.

Implementing terms

```
term = define | expression
define = ( define identifier expression ) | ( define ( variable+ ) term+)
```

Implementing terms

term = *define* | *expression*
define = `( define` *identifier expression* `)` | `( define` `(` *variable+* `)` *term+*`)`

```
(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)
```

> For our purposes of defining the semantics in terms of implementing an interpreter, we do not want to distinguish between a basic definition and a function definition, as this would unnecessarily complicate our code. We, therefore, represent a definition with a single structure, which pairs a variable and an expression (eg, a lambda). In our setting, the distinction between a basic and a function definition is syntactic (not semantic).

Summary of `struct`

```
(struct point (x y z) #:transparent)
```

Simplifies the definition of data structures:

- Creates selectors automatically, eg, `point-x`
- Creates type query, eg, `point?`
- Ensures that functions of a given struct can only be used on values of that struct. *Because, not everything is a list.*

> What is `#:transparent`? A transparent struct prints its contents when rendered as a string.

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 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

UMASS BOSTON

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 arg 2)) 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)
```

Curying 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:

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

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

Currying

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

How do we implement Currying?

How do we implement Currying?

We need two components:

1. A function that accepts each argument in curried form. (Lecture 6)
2. When function (1) receives its last argument, it must apply the curried-function to the stored curried arguments. (Lecture 9)