CS450

Structure of Higher Level Languages

Lecture 2: Definitions, function definition, booleans

Tiago Cogumbreiro

Homework Assignment 1

February 12 at 5:30pm

Sorry, but no late submissions will be accepted!

Logistics

- **Office hours: 3:30pm til 5:00pm**
- **Peruse the Syllabus**
  Available in the course page, in the *Resources* page
- **Register and upload an assignment**
  We need to iron out any quirks. Your feedback is important!
- **Browse *tentative* the class schedule**
  of the semester is available on our course page
- **Homework Assignment #1 (HW1) has *2* parts**
  Part 1 was delivered in Lecture 1; Part 2 is available in the *Resources* page
- **Register in the autograder website using the email address listed in HW1 Part I**
- **You can submit as many times as you want**
- **Use the HW1 template**
  The template helps up submit partial answers and helps understand exercise 3.
- **Uploaded Racket scripts *must* have a `.rkt` extension**

HW1 Errata

Typo in the example listed in Exercise 1.b

The example should be:

```
(define ex2
  (list
    (* 3.14159 (* 10 10))
    (* 3.14159 100)
    314.159))
```

Autograder Results

Results | Code

Sanity check (0.0/1.0)

```
Are you using the homework template?

I could not find the following definitions:
 * define-basic?
 * define-func?
 * define?
 * apply-args
 * apply-func
 * apply?
 * lambda-body
 * lambda-params
 * lambda?
 * bst-insert
 * tree-set-value
 * tree-set-right
 * tree-set-left
 * tree-value
 * tree-right
 * tree-left
 * tree-leaf
 * tree
 * ex3
 * ex2
 * ex1

Tip #1: try assigning a dummy value to each definition. For instance:
(define define-basic? #f)

Tip #2: ensure your definitions are made public. The first two lines of your file should be:
#lang racket
(provide (all-defined-out))
```

STUDENT

[redacted]

AUTOGRADER SCORE

0.0 / 24.0

FAILED TESTS

Sanity check (0.0/1.0)

Autograder Results

Results | Code

Exercise 1.a (0.0/1.5)

Exercise 1.b (0.0/3.5)

Exercise 2 (0.0/2.0)

Exercise 3. bst-insert (0.0/3.0)

Exercise 3. tree (0.0/0.5)

Exercise 3. tree-leaf (0.0/0.5)

Exercise 3. tree-left (0.0/0.5)

Exercise 3. tree-right (0.0/0.5)

Exercise 3. tree-set-left (0.0/0.5)

Exercise 3. tree-set-right (0.0/0.5)

STUDENT

Tiago Cogumbreiro

AUTOGRADER SCORE

0.0 / 24.0

FAILED TESTS

Exercise 1.a (0.0/1.5)
Exercise 1.b (0.0/3.5)
Exercise 2 (0.0/2.0)
Exercise 3. bst-insert (0.0/3.0)
Exercise 3. tree (0.0/0.5)
Exercise 3. tree-leaf (0.0/0.5)
Exercise 3. tree-left (0.0/0.5)
Exercise 3. tree-right (0.0/0.5)
Exercise 3. tree-set-left (0.0/0.5)
Exercise 3. tree-set-right (0.0/0.5)
Exercise 3. tree-set-value (0.0/0.5)
Exercise 3. tree-value (0.0/0.5)
Exercise 4.a. lambda? (0.0/3.0)
Exercise 4.b. lambda-params (0.0/0.5)
Exercise 4.c. lambda-body (0.0/0.5)
Exercise 4.d. apply? (0.0/1.0)
Exercise 4.e. apply-func, 4.f. apply-args (0.0/1.0)
Exercise 4.g. define? (0.0/0.2)
Exercise 4.h. define-basic? (0.0/1.0)
Exercise 4.i. define-func? (0.0/2.8)

Today we will learn about...

- the logical connectives in Racket
- defining variables
- function declarations
- evaluating functions

> Cover up until Section 1.1.8 of the SICP book.

Logic

Values

- Numbers
- Void
- **Booleans**
- Lists
- ...

Boolean, numeric comparisons

```
value = number | boolean | ...
boolean = #t | #f
```

- False: `#f`
- True: anything that is not `#f`
- Logical negation: function `(not e)` negates the boolean result of expression `e`
- Numeric comparisons: `<`, `>`, `≤`, `≥`, `=`

> To avoid subtle bugs, avoid using non-`#t` and non-`#f` values as true. In particular, **contrary to C** the number `0` corresponds to true. *Tip:* There is no numeric inequality operator. Instead, use `(not (= x y))`

Logical and/or

```
expression = value | variable | function-call | or | and | ...
or = ( or expression* )
and = ( and expression* )
```

- Logical-and with short-circuit: `and` (0 or more arguments, 0-arguments yield `#t`)
- Logical-or with short-circuit: `or` (0 or more arguments, 0-arguments yield `#f`)

Boolean examples

Operations `and`/`or` *accept multiple parameters.* Rectangle intersection:

```
(and (< a-left b-right)
     (> a-right b-left)
     (> a-top b-bottom)
     (< a-bottom b-top))
```

As an example of **short-circuit** logic, the expression

```
(or #t (f x y z))
```

evaluates to `#t` and does **not** evaluate `(f x y z)`. Recall that `and` also short-circuits.

Branching

Branching with cond

cond evaluates each branch sequentially until, until the *first* branch's condition evaluates to true.

```
expression = value | variable | function-call | or | and | cond
cond = ( cond branch )
branch = [ condition expression ]
condition = expression | else
```

Example

If x is greater than 3 returns 100, otherwise if x is between 1 and 3 return 200, otherwise returns 300:

```
(cond [(> x 3) 100]
      [(> x 1) 200]
      [else 300])
```

Creating variables

Variable definition

A definition *binds* a variable to the result of evaluating an expression down to a value.

```
( define identifier expression )
```

Examples

```
#lang racket
(define pi 3.14159)
pi
(* pi 2)
```

```
$ racket def-val.rkt
3.14159
6.28318
```

Revisiting the language specification

A *program* consists of zero or more terms.

```
#lang racket
term*
```

A *term* is either an *expression* or a *definition*.

```
term = expression | definition
```

If everything evaluates down to a value,

then what does `define` evaluate to?

Void

Definitions evaluate to `#<void>`, which is the only value that is not printed to the screen.

```
(define pi 3.14159)      ⟵ A definition evaluates to ⟶ #<void>
```

> The void value cannot be created directly. Another way of getting a void value `#<void>` is by calling function `(void)`.

Try running this program and confirm that its output is empty:

```
#lang racket
(void)
```

Evaluating variable definition

When we execute a Racket program, we have an *environment* to bookkeep each variable, that is a map from variable names to values.

```
(define pi 3.14159)
(* pi 2)
```

```
; pi = 3.14159
#<void>
;^^^^^- Eval define
(* pi 2)
```

```
; pi = 3.14159
; Prints #<void>
(* pi 2)
```

Evaluating variable definition

When we execute a Racket program, we have an *environment* to bookkeep each variable, that is a map from variable names to values.

```
(define pi 3.14159)
(* pi 2)
```

```
; pi = 3.14159
#<void>
;^^^^^^- Eval define
(* pi 2)
```

```
; pi = 3.14159
; Prints #<void>
(* pi 2)
```

```
; pi = 3.14159
(* 3.14159 2)
;  ^^^^^^^- Subst pi
```

```
; pi = 3.14159
6.28318
;^^^^^^- Eval func
```

```
; pi = 3.14159
; Print 6.28318
```

Beware of re-definitions

The following is legal Racket code:

```
#lang racket
(define pi 3.14159)
(* pi 2)
(define + #f)
(+ pi 2)
```

Redefinitions lead to subtle errors!

- Redefinitions produce subtle side-effects and may void existing assumptions
- As we will see, redefinitions also complicate the semantics and code analysis

Function declaration

Function declaration

A function declaration is creates an anonymous function and consists of:

- **parameters**: zero or more parameters (identifiers, known as symbols)
- **body** which consist of one or more terms

When calling a function we replace each argument by the parameter defined in the lambda. If the number of parameters is not the expected one, then we get an error. The return value of the function corresponds to the evaluation of the *last* term in the body (known as the **tail position**).

```
function-def = ( lambda ( variable* ) term+)
```

We can define `circumference` as a function and parameterize the radius:

```
#lang racket
(define circumference (lambda (radius) (* 2 3.14159 radius)))
(circumference 2)
```

```
$ racket func.rkt
12.56636
```

Evaluating a `lambda`

```
(define circ
 (lambda (radius) (* 2 3.14159 radius)))
(circ 2)
```

```
; circ = lambda ...
#<void>
;^^^^^- Eval define
(circ 2)
```

```
; circ = lambda ...
; Prints #<void>
(circ 2)
```

```
; circ = lambda ...
((lambda (radius) (* 2 3.14159 radius)) 2)
;^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^- Subst circ
```

```
; circ = lambda ...
(* 2 3.14159 2)
;^^^^^^^^^^^^^^- Applied func
```

```
; circ = lambda ...
12.56636
```

```
; circ = lambda ...
; Prints 12.56636
```

> For more information on evaluation, read Section 1.1.5 of SICP.

Function definition

Racket introduces a shorthand notation for defining functions.

```
( define (variable+ ) term+ )
```

A function definition expects one or more variables (symbols). The first variable is the function variable. The remaining variables are the arguments of the function declaration. The one-or-more terms consist of the body of the function declaration.

Which is a short-hand for:

```
( define variable (lambda ( variable* ) term+ ))
```

Reviewing what we learned

Exercises

1. Write a function that squares a number (with and without the shorthand function-definition notation)

Exercises

1. Write a function that squares a number (with and without the shorthand function-definition notation)
2. Write a function that computes the absolute value of a number.

Exercises

1. Write a function that squares a number (with and without the shorthand function-definition notation)
2. Write a function that computes the absolute value of a number.
3. Can we implement an `if` expression with short-circuit semantics?

```
(define (new-if predicate then-clause else-clause)
  (cond [predicate then-clause]
        [else else-clause]))
```

Exercises

1. Write a function that squares a number (with and without the shorthand function-definition notation)
2. Write a function that computes the absolute value of a number.
3. Can we implement an `if` expression with short-circuit semantics?

```
(define (new-if predicate then-clause else-clause)
  (cond [predicate then-clause]
        [else else-clause]))
```

4. Write a function ⇒ that implements the logical implication, and another function that implements ≠

Exercises

1. Write a function that squares a number (with and without the shorthand function-definition notation)
2. Write a function that computes the absolute value of a number.
3. Can we implement an `if` expression with short-circuit semantics?

```
(define (new-if predicate then-clause else-clause)
  (cond [predicate then-clause]
        [else else-clause]))
```

4. Write a function ⇒ that implements the logical implication, and another function that implements ≠
5. Write a square-root function `safe-div` that returns `#f` when division is undefined. Implement a `plus`-operator that can deal with undefined numbers. Learn to use `(error)` instead.

Reminders

Tip 1: No end-of-line characters

> Racket has no end-of-line delimiters (contrary to, say, C-like languages which use semi-colons)

```
2 + 3
```

corresponds to

```
2 ; ← evaluate a number
+ ; ← evaluate a variable
3 ; ← evaluate a number
```

when run it would print:

```
2
#<procedure:+>
3
```

Tip 2: Re-definitions hide old definitions

A definition with the same name overwrites the previous definition.

```
#lang racket
(define + 3)
+
```

Prints

3