CS450

Structure of Higher Level Languages

Lecture 2: Branching and function definitions

Tiago Cogumbreiro

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 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 variable 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
```

Do variables evaluate?

> Variables are considered expressions, so the runtime must lookup the value ***bound*** to a variable as one step of the evaluation.

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

Exercises and recap

Exercise: conditionals

- The modulo operator (%) in Racket is function `modulo`
- The equality operator in Racket is `equal?`

Translate the following code to Racket:

```
n = 16
if n % 15 == 0:
    return "fizzbuzz"
if n % 3 == 0:
    return "fizz"
if n % 5 == 0:
    return "buzz"
return n
```

Exercise: conditionals (solution)

```
(define n 16)
(cond [(equal? (modulo n 15) 0) "fizzbuzz"]
      [(equal? (modulo n 3) 0) "fizz"]
      [(equal? (modulo n 5) 0) "buzz"]
      [else n])
```

Exercise: evaluation

```
#| How many evaluation steps should we expect? |#
(define x (* 10 2))
(+ x (* 4 2))
```

Solution

Exercise: evaluation

```
#| How many evaluation steps should we expect? |#
(define x (* 10 2))
(+ x (* 4 2))
```

Solution

5 steps

```
; Step 1: eval *
(define x 20)
(+ x (* 4 2))
```

Exercise: evaluation

```
#| How many evaluation steps should we expect? |#
(define x (* 10 2))
(+ x (* 4 2))
```

Solution

5 steps

```
; Step 1: eval *
(define x 20)
(+ x (* 4 2))
```

```
; Step 2: eval define
; x = 20
(+ x (* 4 2))
```

Exercise: evaluation

```
#| How many evaluation steps should we expect? |#
(define x (* 10 2))
(+ x (* 4 2))
```

Solution

5 steps

```
; Step 1: eval *
(define x 20)
(+ x (* 4 2))
```

```
; Step 2: eval define
; x = 20
(+ x (* 4 2))
```

```
; Step 3: eval x
; x = 20
(+ 20 (* 4 2))
```

Exercise: evaluation

```
#| How many evaluation steps should we expect? |#
(define x (* 10 2))
(+ x (* 4 2))
```

Solution

5 steps

```
; Step 1: eval *
(define x 20)
(+ x (* 4 2))

; Step 4: eval *
(+ 20 8)
```

```
; Step 2: eval define
; x = 20
(+ x (* 4 2))
```

```
; Step 3: eval x
; x = 20
(+ 20 (* 4 2))
```

Exercise: evaluation

```
#| How many evaluation steps should we expect? |#
(define x (* 10 2))
(+ x (* 4 2))
```

Solution

5 steps

```
; Step 1: eval *
(define x 20)
(+ x (* 4 2))
```

```
; Step 2: eval define
; x = 20
(+ x (* 4 2))
```

```
; Step 3: eval x
; x = 20
(+ 20 (* 4 2))
```

```
; Step 4: eval *
(+ 20 8)
```

```
; Step 5: eval +
28
```

Function declaration

Function declaration

A function declaration 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-dec = ( 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

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+ ))
```

Exercise

The McCarthy 91 function was invented by computer scientist John McCarthy to motivate formal verification.

$$M(n) = n - 10 \text{ if } n > 100$$
$$M(n) = M(M(n+11)) \text{ if } n \leq 100$$

- Implement the function in Racket
- What is $M(99)$?

Exercise

The McCarthy 91 function was invented by computer scientist John McCarthy to motivate formal verification.

$$M(n) = n - 10 \text{ if } n > 100$$
$$M(n) = M(M(n + 11)) \text{ if } n \leq 100$$

- Implement the function in Racket
- What is $M(99)$?

The McCarthy 91 function is equivalent to

$$\begin{aligned} M(n) &= n - 10 \quad \text{if } n > 100 \\ M(n) &= 91 \quad \text{if } n \leq 100 \end{aligned}$$

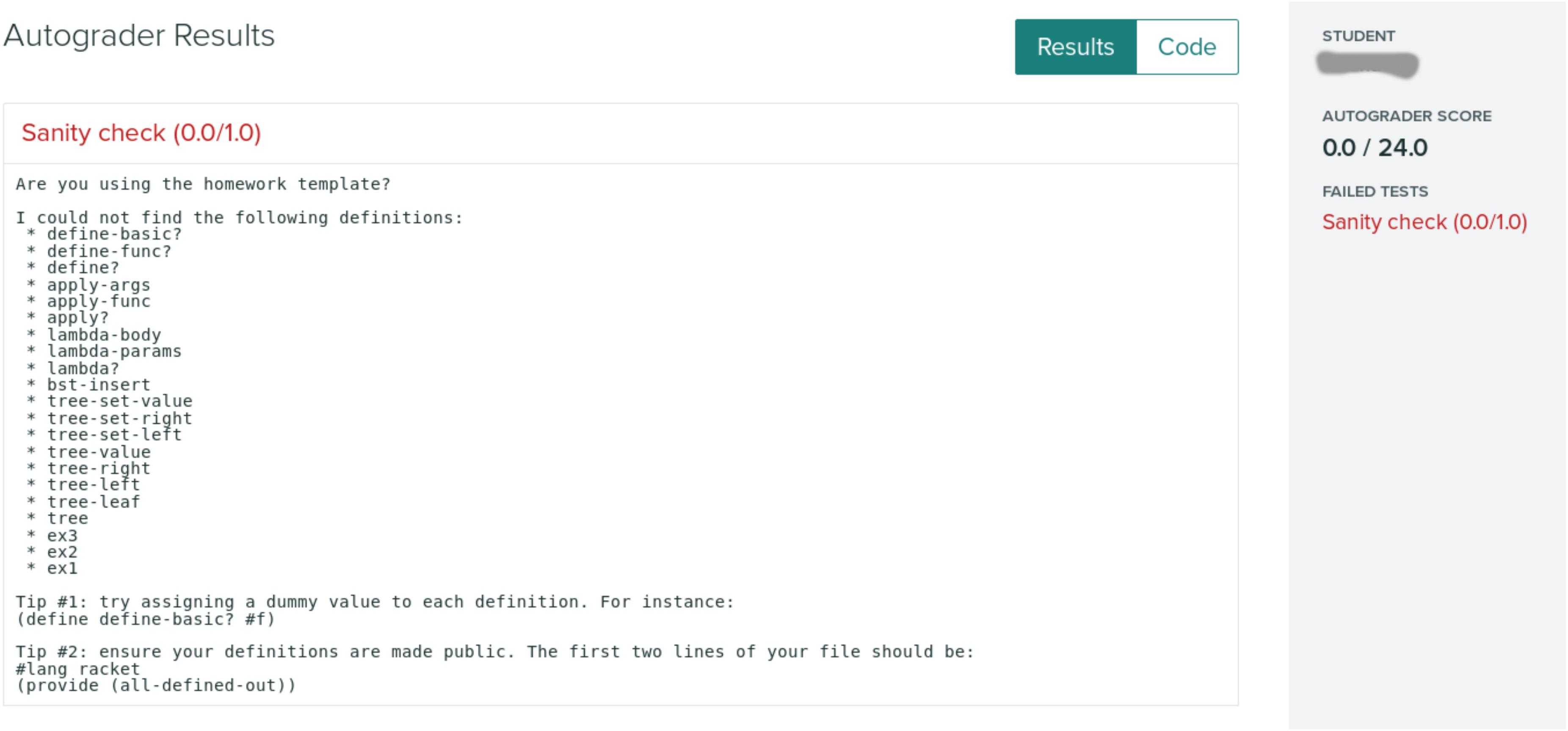
Homework example

Template excerpt

The homework template is in our Directory page:

```
(define ex1 'todo)
(define ex2 'todo)
(define ex3 'todo)

;; ...
```



Autograder Results

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

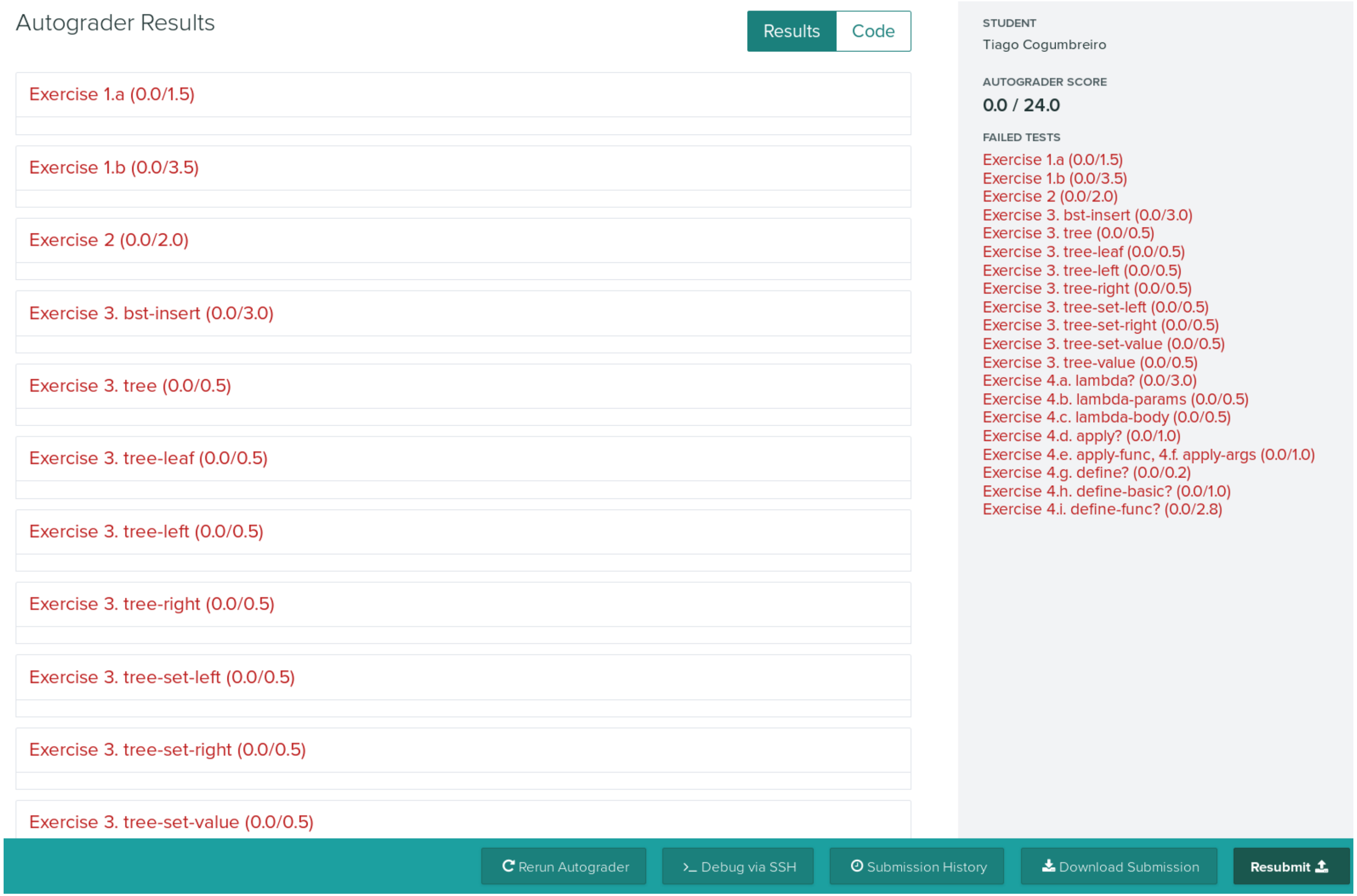
[redacted]

AUTOGRADER SCORE

0.0 / 24.0

FAILED TESTS

Sanity check (0.0/1.0)



Autograder Results

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)

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)