CS450

Structure of Higher Level Languages

Lecture 2: Definitions, function definition, booleans

Tiago Cogumbreiro

Today we will learn...



- evaluation composed expressions step-by-step
- the logical connectives in Racket
- defining variables
- function declarations
- evaluating functions

Cover up until Section 1.1.8 of the SICP book.

Evaluating a function call

Evaluating a function call



Evaluation works from left-to-right from top-to-bottom

Arithmetic expressions example



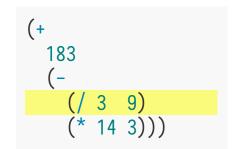
$$\left((11 \cdot 15) + (14 + 4) \right) + \left(\frac{3}{9} - (14 \cdot 3) \right)$$

Arithmetic expressions example



A longer example





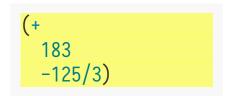
A longer example



```
(+
183
(-
(/ 3 9)
(* 14 3)))
```

```
(+
183
(-
1/3
(* 14 3)))
```

```
(+
183
(-
1/3
42))
```



424/3

Is this example a legal Racket program?



```
#lang racket
sin
```

Is this example a legal Racket program?



```
#lang racket
sin
```

Yes! sin is a variable, so a valid expression. Hence, Racket just prints what is in variable sin.

```
$ racket sin.rkt
#procedure:sin>
```

Note: In Racket lingo the word *procedure* is a synonym for function.

Racket specification



```
program = #lang racket expression*
expression = value | variable | function-call | · · ·
value = number | · · ·
function-call = ( expression+ )
```

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))</pre>
```

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 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?

Voic



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

```
(define pi 3.14159) \leftarrow A definition evaluates to \rightarrow #<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