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

```racket
#racket lang
; Version 1:
(* 3.14159 (* 10 10))
; Version 2:
(* 3.14159 100)
;          ^^^- Evaluated (* 10 10)
; Version 3:
314.159
;^^^^^^- Evaluated (* 3.14159 * 100)
```
Arithmetic expressions example

\[((11 \cdot 15) + (14 + 4)) + \left(\frac{3}{9} - (14 \cdot 3)\right)\]
Arithmetic expressions example

\[
((11 \cdot 15) + (14 + 4)) + \left(\frac{3}{9} - (14 \cdot 3)\right)
\]
A longer example

\[
\begin{align*}
( + & ) \\
( + & 11 15 ) \\
( + & 14 4 ) \\
( - & ) \\
( / & 3 9 ) \\
( * & 14 3 ) \\
\end{align*}
\]

\[
\begin{align*}
( + & ) \\
( + & 165 ) \\
( + & 14 4 ) \\
( - & ) \\
( / & 3 9 ) \\
( * & 14 3 ) \\
\end{align*}
\]

\[
\begin{align*}
( + & ) \\
( + & 165 ) \\
( + & 18 ) \\
( - & ) \\
( / & 3 9 ) \\
( * & 14 3 ) \\
\end{align*}
\]

\[
\begin{align*}
( + & ) \\
( + & 183 ) \\
( - & ) \\
( / & 3 9 ) \\
( * & 14 3 ) \\
\end{align*}
\]
A longer example

\[
\begin{align*}
( + & \quad (+ & \quad (+ & (\quad \quad \quad 11 & 15) & 165) & 183) \\
( + & (\quad \quad \quad 14 & 4)) & (+ & (\quad \quad \quad 14 & 4)) & 183) \\
(- & (\quad \quad \quad 3 & 9) & (- & (\quad \quad \quad 3 & 9) & 424/3) \\
(* & (\quad \quad \quad 14 & 3)) & (* & (\quad \quad \quad 14 & 3)) & (*) & (\quad \quad \quad 14 & 3)) \\
\end{align*}
\]
Is this example a legal Racket program?

```racket
#lang racket
sin
```
Is this example a legal Racket program?

```racket
#lang racket
sin
```

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

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

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


game = #lang racket expression*
expression = value | variable | function-call | ...
value = number | ...
function-call = ( expression+ )
Logic
Values

- Numbers
- Void
- **Booleans**
- Lists
- ...
Boolean, numeric comparisons

\[
\begin{align*}
value &= \text{number} \mid \text{boolean} \mid \ldots \\
boolean &= \#t \mid \#f
\end{align*}
\]

- False: \#f
- True: anything that is not \#f
- Logical negation: function \((\text{not } e)\) 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 \((\text{not } (\#eq x y))\)
Logical and/or

\[
\text{expression} = \text{value} \mid \text{variable} \mid \text{function-call} \mid \text{or} \mid \text{and} \mid \cdots \\
\text{or} = ( \text{or} \ \text{expression}^* ) \\
\text{and} = ( \text{and} \ \text{expression}^* )
\]

- 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:

\[
\text{and} \ (\text{< a-left b-right}) \\
\quad (\text{> a-right b-left}) \\
\quad (\text{> a-top b-bottom}) \\
\quad (\text{< a-bottom b-top})
\]

As an example of short-circuit logic, the expression

\[
\text{or} \ #t \ (f \ x \ y \ z)
\]

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.

\[
\text{expression} = \text{value} \mid \text{variable} \mid \text{function-call} \mid \text{or} \mid \text{and} \mid \text{cond} \\
\text{cond} = ( \text{cond} \ \text{branch} ) \\
\text{branch} = [ \text{condition} \ \text{expression} ] \\
\text{condition} = \text{expression} \mid \text{else}
\]

Example

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

\[(\text{cond\ (}\ (\text{> x 3) 100} \\text{)} \text{\ (}\ (\text{> x 1) 200} \text{\ (}\text{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

```racket
#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.

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

; pi = 3.14159

#<void>

; Prints #<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.
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>
;Evaluator define
(* pi 2)
```

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

```
; pi = 3.14159
6.28318
;Evaluator function
```

```
; pi = 3.14159
;Print 6.28318
```
The following is legal Racket code:

```racket
#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