Racket Forms and Callsites

CIS352
Kris Micinski
Forms

• A form is a recognized syntax in the language
  • (if …), (and …) are forms
  • But +, list refer to functions
  • Core forms defined by the language (if/and/define/…)
  • You can define new forms too! More on this later…
  • Scheme prefers to give a small number of general forms.
Forms

• The tag just after the open-paren determines the form:
  • (define foo value) — Define a variable
  • (define (foo a0 a1 …) body) — Define a function
  • (if guard e-true e-false), (or e0 e1 …), etc
  • By default, otherwise, (e0 e1 …) is a function call
Value and Expressions

• Every language has a set of **values**
  • Primitive objects representable at runtime
  • Expressions evaluate to values
  • Numbers, strings, but also functions (closures)
• An **expression** is any syntax that evaluates to a value
  • Very important term to know!
Which of the following are expressions:

- `(define x 23)`
- `x`
- `(+ x 3)`
- `(define (foo x) (+ x 1))`
- `(if x (foo x) (bar x))`
Which of the following are expressions:
• `(define x 23)` — Doesn’t evaluate to a value
• `x`
• `(+ x 3)`
• `(define (foo x) (+ x 1))` — Doesn’t eval to value
• `(if x (foo x) (bar x))`
Exercise

Define a function that takes an argument, x, and returns:

- x times 2, if x is greater than 0
- x times -2, otherwise
(define (f x)
  (if (< x 0)
      (* 2 x)
      (* -2 x))))
Define a function that takes an argument, $x$, and returns:

- $x$ divided by 2, if $x$ is even
- $x$ times 3 plus 1, if $x$ is odd

**Hint:** use = and modulo to check if $x$ is even/odd
(define (collatz x)
  (if (= 0 (modulo x 2))
      (/ x 2)
      (+ 1 (* 3 x))))
Cond

- (cond [clause0 body0] … [else body-else])
- Each clause is evaluated in order
- Evaluates body of first matching clause
- Else may be
Definitions and the Environment

CIS352
Kris Micinski
Definitions

• The form `define` is used to define variables

• Define comes in two forms
  • `(define id expr)` — Define variable `id` as `expr`
  • `(define (f a0 ... ) body ...+)`
    • Define a function `f` with arguments `a0, ...`
    • At least one body (typically only one)
Exercise

• Define a variable named x to be 42
• Define a function foo, which behaves as the identity function
The Environment

- The environment at some point in the program includes the set of variables in scope (accessible) at that point.
- Every syntactic point has a (potentially) unique environment.

```
(define x 23)
(+ x 1) ;; x is 23
(define y 24)
(+ x y) ;; x & y defined
```
Environments Nest

• Note that environments are hierarchical

• Definitions inside a function do not escape the function

• This relates to **lexical scope** which we will define soon

```
(define y 5)
(define (foo)
    (displayln y) ;; 5
    (define y 4)
    y)            ;; 4
(foo)            ;; 4
y                ;; 5
```
Exercise

What does the following function return:

```
(define (foo)
  (define + 1)
  (define / (* 2 +))
  (- + /))
```
Exercise

What does the following function return:
-1

Upshot: “built-in” functions are not special

(define (foo)
  (define + 1)
  (define / (* 2 +))
  (- + /))
Let

- Definitions with define are not expressions
- \((\text{let } ([\text{var } e]) \text{ e-body})\)
  - Expression: evaluates \(\text{e-body}\) with \text{var} defined as \(e\)
  - Can have more than one \text{var}

\[
(\text{let } ([x \ 2])
\quad (+ \ x \ 3)) \quad ;; \ 5
\]

\[
(\text{let } ([x \ 2]
\quad [y \ 3])
\quad (+ \ x \ y)) \quad ;; \ 5
\]
Let

• Let does not allow simultaneous bindings to see each other

• I think of it as “parallel let”

(let ([x 2] [y x]) ;; bad
 (+ x y)) ;; 5
Let* lets you define a sequence of variables

I think of it as “sequential let”

```
(let* ([x 2] [y x]) ;; good
 (+ x y)) ;; 5
```