

"Lisp Cycles", <http://xkcd.com/297/>

CS 152: *Programming Language Paradigms*



Introduction to Scheme

Prof. Tom Austin

San José State University

Key traits of Scheme

1. Functional
2. Dynamically typed
3. Minimal
4. Lots of lists!



Interactive Racket

```
$ racket
```

```
Welcome to Racket
```

```
v6.0.1.
```

```
> (* (+ 2 3) 2)
```

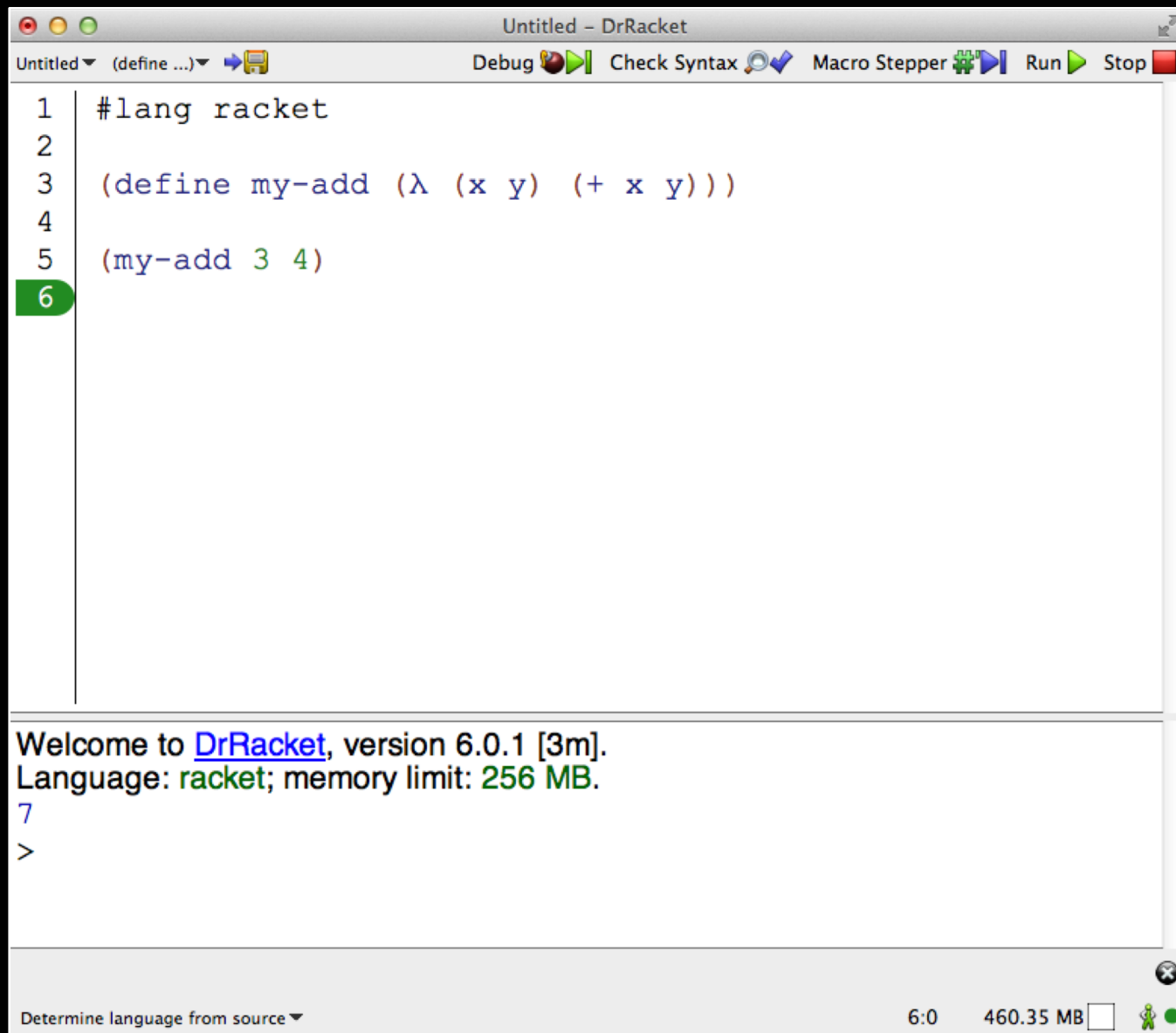
```
10
```

```
>
```

Running Racket from Unix Command Line

```
$ cat hello-world.rkt
#lang racket
(displayln "Hello world")
$ racket hello-world.rkt
Hello world
$
```

DrRacket



The image shows a screenshot of the DrRacket IDE. The window title is "Untitled - DrRacket". The menu bar includes "Untitled", "(define ...)", and a save icon. The toolbar contains "Debug", "Check Syntax", "Macro Stepper", "Run", and "Stop".

```
1 #lang racket
2
3 (define my-add (λ (x y) (+ x y)))
4
5 (my-add 3 4)
6
```

Below the code editor, the output area displays the following text:

```
Welcome to DrRacket, version 6.0.1 [3m].
Language: racket; memory limit: 256 MB.
7
>
```

At the bottom of the window, there is a status bar with the text "Determine language from source", the time "6:0", the memory usage "460.35 MB", and a small green robot icon.

Racket Simple Data Types

- **Booleans:** `#t`, `#f`

 - `(not #f)` \Rightarrow `#t`

 - `(and #t #f)` \Rightarrow `#f`

- **Numbers:** `32`, `17`

 - `(/ 3 4)` \Rightarrow `3/4`

 - `(expt 2 3)` \Rightarrow `8`

- **Characters:** `#\c`, `#\h`

Racket Compound Data Types

- **Strings**

- `(string #\S #\J #\S #\U)`

- `"Spartans"`

- **Lists:** `' (1 2 3 4)`

- `(car ' (1 2 3 4)) => 1`

- `(cdr ' (1 2 3 4)) => ' (2 3 4)`

- `(cons 9 ' (8 7)) => ' (9 8 7)`

Setting values

```
(define zed "Zed")
```



Global
variables only

Setting values

```
(define zed "Zed")
```

```
(displayln zed)
```

```
{let
```

```
  ([z2 (string-append zed zed)]
```

```
  [sum (+ 1 2 3 4 5)])
```

```
(displayln z2)
```

```
(displayln sum) }
```

Setting values

```
(define zed "Zed")
```

```
(displayln zed)
```

```
{let
```

```
  ([z2 (string-append zed zed)])
```

```
  ([sum (+ 1 2 3 4 5)])
```

```
(displayln z2)
```

```
(displayln sum) }
```

List of local
variables



Setting values

```
(define zed "Zed")
```

```
(displayln zed)
```

Variable names



```
{let
```

```
  ([z2 (string-append zed zed)]
```

```
   [sum (+ 1 2 3 4 5)])
```

```
(displayln z2)
```

```
(displayln sum) }
```

Setting values

```
(define zed "Zed")  
(displayln zed)
```

Variable
definitions



```
{let  
  ([z2 (string-append zed zed)  
   [sum (+ 1 2 3 4 5)])  
  (displayln z2)  
  (displayln sum) }
```

Setting values

```
(define zed "Zed")
```

```
(displayln zed)
```

```
{let
```

```
  ([z2 (string-append zed zed)]
```

```
  [sum (+ 1 2 3 4 5)])
```

```
(displayln z2)
```

```
(displayln sum) }
```

Scope of
variables



let and let*

```
(let ([x 3]
      [y (* x 2)])
  (displayln y))
```

```
x: unbound identifier  
in module in: x
```

let and let*

```
(let* ([x 3]  
      [y (* x 2)])  
  (displayln y))
```

6

All the data types discussed so far are called *s-expressions* (s for symbolic).

Note that programs themselves are also s-expressions. **Programs are data.**

Lambdas (λ)

```
(lambda  
  (x)  
  (* x x) )
```

Also known as
"functions"

Lambdas (λ)

(**λ**
 (x)
 (* x x))

In DrRacket,
 λ can be typed
with Ctrl + \

Lambdas (λ)

(lambda

(x)

Parameter list

(* x x))

Lambdas (λ)

(lambda

(x)

(* x x))



Function body

Lambdas (λ)

((lambda

(x)

(* x x))

3)

Evaluates to 9

Lambdas (λ)

```
(define square  
  (lambda (x) (* x x)))  
  
(square 4)
```

Alternate Format

```
(define (square x)
  (* x x))

(square 4)
```

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```

If Statements

```
(if (< x 0)  
    (+ x 1)  
    (- x 1))
```



Condition

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```

"Then" branch

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```



"Else" branch

Cond Statements

```
(cond
  [(< x 0) "Negative"]
  [(> x 0) "Positive"]
  [else "Zero"])
```

Scheme does not let you reassign variables

"If you say that a is 5, you can't say it's something else later, because you just said it was 5.

What are you, some kind of liar?"

--Miran Lipovača

Recursion

- Base case
 - when to stop
- Recursive step
 - calls function with a **smaller version** of the same problem

Algorithm to count Russian dolls



Recursive step



- Open doll
- *Count number dolls in the inner doll*
- Add 1 to the count

Base case

- No inside dolls
- return 1



An iterative definition of a `count-elems` function

Set `count` to 0.

For each element:

 Add 1 to the count.

The answer is the count.

BAD!!! Not
the way that
functional
programmers
do things.

A recursive definition of
a `count-elems` function

Base case:

If a list is empty, then the answer is 0.

Recursive step:

Otherwise, the answer is 1 more than the size of the tail of the list.

Recursive Example

```
(define (count-elems lst)
  (cond [(= 0 (length lst)) 0]
        [else (+ 1
                  (count-elems (cdr lst))
                )]))
```

```
(count-elems ' ( ) )
```

```
(count-elems ' (1 2 3 4) )
```

Recursive Example

Base case

```
(define (count-elems lst)
  (cond [(= 0 (length lst)) 0]
        [else (+ 1
                  (count-elems (cdr lst))
                )]))
```

```
(count-elems '())
```

```
(count-elems '(1 2 3 4))
```

Recursive Example

```
(define (count-elems lst)
  (cond [(= 0 (length lst)) 0]
        [else (+ 1
                  (count-elems (cdr lst))
                )]))
```

Recursive step

```
(count-elems ' ( ) )
```

```
(count-elems ' (1 2 3 4) )
```

```
(count-elems '(1 2 3 4))  
=> (+ 1 (count-elems '(2 3 4)))  
=> (+ 1 (+ 1 (count-elems '(3 4))))  
=> (+ 1 (+ 1 (+ 1 (count-elems '(4)))))  
=> (+ 1 (+ 1 (+ 1 (+ 1  
                    (count-elems '()))))))  
=> (+ 1 (+ 1 (+ 1 (+ 1 0))))  
=> (+ 1 (+ 1 (+ 1 1)))  
=> (+ 1 (+ 1 2))  
=> (+ 1 3)  
=> 4
```

Mutual recursion

```
(define (is-even? n)
  (if (= n 0)
      #t
      (is-odd? (- n 1))))
```

```
(define (is-odd? n)
  (if (= n 0)
      #f
      (is-even? (- n 1))))
```

let* and letrec

```
(let*  
  ([is-even?  
    (lambda (n)  
      (or (zero? n)  
          (is-odd? (sub1 n))))])  
  [is-odd?  
    (lambda (n)  
      (and (not (zero? n))  
           (is-even? (sub1 n))))])  
  (is-odd? 11))
```

```
is-odd?: unbound  
identifier in  
module in: is-odd?
```

let* and letrec

(letrec

#t

```
( [is-even?
```

```
  (lambda (n)
```

```
    (or (zero? n)
```

```
        (is-odd? (sub1 n))))]
```

```
[is-odd?
```

```
  (lambda (n)
```

```
    (and (not (zero? n))
```

```
         (is-even? (sub1 n))))])
```

```
(is-odd? 11))
```

Text Adventure Example
(in class)

Lab1

Part 1: Implement a `max-num` function.
(Don't use the `max` function for this lab).

Part 2: Implement the "fizzbuzz" game.

sample run:

```
> fizzbuzz 15
"1 2 fizz 4 buzz fizz 7 8
fizz buzz 11 fizz 13 14
fizzbuzz"
```

First homework

Java example with large num

```
public class Test {  
    public void main(String[] args) {  
        System.out.println(  
            99999999999999999999999999999999 * 2);  
    }  
}
```


HW1: implement a BigNum module

HW1 explores how you might support big numbers in Racket if it did *not* support them.

- Use a list of 'blocks' of digits, least significant block first. So 9,073,201 is stored as:
' (201 73 9)
- Starter code is available at on course website.

Overview of Homework

- Grade school addition
- Big number addition

- Grade school multiplication
- Big number multiplication

Before next class

- Read chapters 3-5 of *Teach Yourself Scheme*.
- If you accidentally see `set!` in chapter 5,
pluck out your eyes lest you become impure.
 - Alternately, just never use `set!`
(or get a 0 on your homework/exam).