Racket Intro

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

What is Racket?

- Racket is a programming language
- Racket is also a metaprogramming language: one of its primary intentions is for designing other programming languages
- Originally a Scheme implementation: you will find it very similar to SlytherLisp
- Primary paradigms: functional, language-oriented

Setting up Racket

For this course, we will use **Racket 7**. This version is already installed on the ALAMODE machines.

If you are installing on your own Linux machine, check your distribution's package manager for Racket. It is OK if it is version 6: 99% of what we do in this course will be compatible.

Racket Editor Advice

- Racket ships with an IDE: Dr. Racket. This works well, especially for learning. If you're not sold on using another editor, I would recommend it.
- If you're a Vim guru, I recommend adding the **vim-sexp** plugin to assist with s-expression editing.
- If you're an Emacs guru, try out **lispy** or **smartparens**.
- Other editors: see if an *s-expression aware* editing mode is available.

It all starts with a #lang

The very first line of a Racket source file is typically this line:

```
#lang racket
```

This tells the Racket interpreter to use the Racket programming language. We will explore other languages in Racket throughout the semester, and you'll even have the chance to make your own.

Defining a Function

The same syntax as SlytherLisp can be used to define a function:

```
#lang racket

(define (fahrenheit->celsius temperature)
  (* (/ 9 5) (- temperature 32)))
```

Just like SlytherLisp, functions return the value of the last expression in their body.

Lists

A cons cell (called *pair* in Racket) can be built with the `cons` function:

```
(cons 1 2)
```

We can use this to build lists, of course:

```
(cons 1 (cons 2 (cons 3 '())))
```

Note the difference to SlytherLisp above: SlytherLisp uses `NIL` to denote the empty list, whereas Racket uses `'()`.

As a convenience, you can use the `list` function to build a list:

```
(list 1 2 3)
```

`list*` will produce a list where the last CDR is the last element instead of `'()`.

Quoting

Racket follows similar quoting rules to SlytherLisp: the ' symbol causes the adjacent element to not be evaluated:

```
> (define x 10)
> x
10
> 'x
'x
> '(1 2 3)
'(1 2 3)
> (list 1 (+ 2 3) 25)
'(1 5 25)
> '(1 (+ 2 3) 25)
'(1 (+ 2 3) 25)
```

Quasiquotations

A quasiquotation starts with a backtick, and elements which start with a comma are evaluated, elements which don't start with a comma are not:

```
> `(1 ,(+ 5 5) (+ 5 5))
'(1 10 (+ 5 5))
> (define x 10)
> `(1 (+ ,x 5) ,x)
'(1 (+ 10 5) 10)
```

Note that quasiquotations are not available in SlytherLisp, this is something you could possibly implement for D6 (language extension) or D7 (extra credit).

Quasiquotation Splicing

`,@` in a quasiquoted pattern will unquote and "splice" the elements of a list directly:

```
> `(1 ,@(list (+ 2 3) (+ 4 5)))
'(1 5 9)

;; compare to...
> `(1 ,(list (+ 2 3) (+ 4 5)))
'(1 (5 9))
```

Practice

With your learning group, evaluate each of these expressions. Write the result down on a sheet of paper:

1. `(cons '(1 2 3) '(1 2 3))`
2. `(cons 1 '(2 3))`
3. `(car `((+ 1 2) ,(+ 1 2) 3))`
4. `(cdr `((+ 1 2) ,(+ 1 2) 3))`
5. `(cdr (cdr (list* 'x 'y 'z)))`
6. `(eval (cons + '(1 2 3)))`

Functions as Data

One of the most important concepts of the Racket language is that functions themselves are data and can be passed around just like any other type of data.

```
(define (call-n-times f n arg)
  (if (= n 0)
    arg
    (f (call-n-times f (- n 1) arg))))
```

This forms the basis for **higher-order functions**: functions that operate on and/or return functions.

Common Higher Order Functions

All of these functions are available in Racket:

- `(map f lst)`: apply `f` to each element in `lst` and return the resultant list.
- `(filter f lst)`: apply `f` to each element in `lst`, which should return `#t` or `#f` for each element. Return the list with only the elements which `f` returned `#t`.
- `(compose f g ... z)`: return a function which computes `(f (g (... (z args...) ...)))`.
- `(curry f arg ...)`: return a function which computes `(f arg ... args...)`
- `(curryr f arg ...)`: return a function which computes `(f args... arg...)`

Anonymous Functions

It becomes convenient to write functions without assigning them to a variable frequently when using these functions. This is a function which takes an `x` and squares it:

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

For example:

```
> (map (lambda (x) (* x x)) '(1 2 3))
'(1 4 9)
```

For fun: implement this squaring function using `curry` or `curryr` and `expt`.

Examples: Map

```
> (map add1 '(1 1 2 3 5))
'(2 2 3 4 6)
> (define (invert x) (/ 1 x))
> (map invert '(1 1 2 3 5))
'(1 1 1/2 1/3 1/5)
> (map (compose invert invert) '(1 1 2 3 5))
'(1 1 2 3 5)
> (map (compose exact->inexact fahrenheit->celsius) '(32 50 75))
'(0.0 32.4 77.4)
```

Examples: Filter

```
> (filter even? '(1 2 3 4 5))
'(2 4)
> (filter odd? '(1 2 3 4 5))
'(1 3 5)
> (filter (lambda (x) (> (- x 3) 0)) '(1 2 3 4 5))
'(4 5)
```

Binding Local Variables

The `let` macro will bind local variables:

```
(let ([a 10]
      [b 20])
  (printf "~a ~a~%" a b))
```

`let*` will bind in order, equivalent to nesting a bunch of lets together:

```
(let* ([a 10]
       [b (+ a 10)])
  (printf "~a ~a~%" a b))
```

Equivalent to:

```
(let ([a 10])
  (let ([b (+ a 10)])
    (printf "~a ~a~%" a b)))
```

With your learning group: devise a case where `let` and `let*` would both produce different results (not resulting in an error)

Putting it all together

1. Write a function which will take the `sin` of all odd numbers in a list. (use `map`/`filter`)
2. Do it again using recursion. (Hint: use `null?` to check for the empty list)