# CS46K Programming Languages

## Structure and Interpretation of Computer Programs

## Chapter 7: Scheme Language for Functional Programming

LECTURE 9: SCHEME LANGUAGE

DR. ERIC CHOU

IEEE SENIOR MEMBER

# Objectives

- Scheme Language

- Racket (Dr. Racket) Programming environment

- Scheme data types, lists, function, conditional, and other programming struct.

- Functional Programming using Scheme

# Overview

# History

- Lisp was created in 1958 by John McCarthy at MIT
  - Stands for **LIS**t **P**rocessing
  - Initially ran on an IBM 704
    - Origin of the car and cdr functions
- Scheme developed in 1975
  - A dialect of Lisp
  - Named after Planner and Conniver languages
    - But the computer systems then only allowed 6 characters
- Racket developed in 1995

# Scheme/Lisp Application Areas

- Artificial Intelligence
  - expert systems
  - planning
- Simulation, modeling
- Rapid prototyping

# Functional Languages

**Imperative Languages**
- Ex. Fortran, Algol, Pascal, Ada
- based on von Neumann computer model

**Functional Languages**
- Ex. Scheme, Lisp, ML, Haskell
- Based on mathematical model of computation and lambda calculus

# Textbook

- The Scheme Programming Language Fourth Edition: https://www.scheme.com/tspl4/

- YouTube Video: https://youtu.be/6k78c8EctXI

- Racket Reference: https://docs.racket-lang.org/index.html

- Scheme Keywords: http://community.schemewiki.org/?scheme-keywords

# Tools

- MIT-GNU Scheme: https://www.gnu.org/software/mit-scheme/

- Online Scheme Interpreter (TutorialPoints): https://www.tutorialspoint.com/execute_scheme_online.php

- Online Scheme Interpreter (Jdoodle): https://www.jdoodle.com/execute-scheme-online/

- Racket Language: https://racket-lang.org/

File   Edit   View   Language   Racket   Insert   Scripts   Tabs   Help

a1.rkt ▾   (define ...) ▾                                                      Check Syntax 🔍✔   Debug 🐞▶|   Macro Stepper #▶|   Run ▶   Stop ■

```racket
1  #lang racket
2  (+ 3 2)
3  (quotient 15 5)
4  (+ (* 3
5        (+ (* 2 4)
6           (+ 3 5)))
7     (+ (- 10 7)
8        6))
```

Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
5
3
57
> |

Determine language from source ▾                                            6:2      494.75 MB

# Basic Scheme Language

# Introduction

- **Scheme** is an imperative language with a functional core. The functional core is based on the lambda calculus. In this lecture only the functional core and some simple I/O is presented.

- In functional programming, parameters play the same role that assignments do in imperative programming. Scheme is an applicative programming language. By applicative, we mean that a Scheme function is applied to its arguments and returns the answer. Scheme is a descendent of LISP. It shares most of its syntax with LISP but it provides lexical rather than dynamic scope rules. LISP and Scheme have found their main application in the field of artificial intelligence.

# Introduction

- The purely functional part of Scheme has the semantics we expect of mathematical expressions. One word of caution: Scheme evaluates the arguments of functions prior to entering the body of the function (eager evaluation). This causes no difficulty when the arguments are numeric values. However, non-numeric arguments must be preceded with a single quote to prevent evaluation of the arguments. The examples in the following sections should clarify this issue.

- Scheme is a **weakly typed** language with **dynamic type checking** and lexical scope rules.

# The Structure of Scheme Programs

- A Scheme program consists of a set of function definitions. There is no structure imposed on the program and there is no main function. Function definition may be nested. A Scheme program is executed by submitting an expression for evaluation. Functions and expressions are written in the form

  (**function_name arguments**)

- This syntax differs from the usual mathematical syntax in that the function name is moved inside the parentheses and the arguments are separated by spaces rather than commas. For example, the mathematical expression 3 + 4 * 5 is written in Scheme as

  (+ 3 (* 4 5))

# Syntax

- The programming language Scheme is syntactically close to the lambda calculus.

  **Scheme Syntax**

  ```
  E in Expressions
  I in Identifiers (variables)
  K in Constants
  E ::= K|I|(E_0 E*)|(lambda (I*) E2)|(define I E')
  ```

- The star `*' following a syntactic category indicates zero or more repetitions of elements of that category thus Scheme permits lambda abstractions of more than one parameter.

- Scheme departs from standard mathematical notation for functions in that functions are written in the form (Function-name Arguments...) and the {arguments are separated by spaces and not commas}.

# Syntax

For example,

```
(+ 3 5)
(fac 6)
(append '(a b c) '(1 2 3 4))   ; literals
```

The first expression is the sum of 3 and 5, the second presupposes the existence of a function **fac** to which the argument of 6 is presented and the third presupposes the existence of the function append to which two lists are presented. Note that the quote is required to prevent the (eager) evaluation of the lists. Note uniform use of the standard prefix notation for functions.

# Syntax

- Number literals: `1, 2, 3`

- Function call: `fac, append`

- List literals: `'(1 2 3)`

# Data Types

# Types

- Among the constants (atoms) provided in Scheme are numbers, the boolean constants **#t** and **#f**, the empty list (), and strings. Here are some examples of **atoms** and a string:

```
A, abcd, THISISANATOM, AB12, 123,

9Ai3n, "A string"
```

# Atom

- Atoms are used for variable names and names of functions. A list is an ordered set of elements consisting of atoms or other lists. Lists are enclosed by parenthesis in **Scheme** as in **LISP**. Here are some examples of lists.

```
(A B C)

(138 abcde 54 18)

(SOMETIMES (PARENTHESIS (GET MORE)) COMPLICATED)

()
```

# List

- Lists are can be represented in functional notation. There is the empty list represented by () and the list **cons**truction function **cons** which constructs lists from elements and lists as follows: a list of one element is

```
(cons X ())
```

- and a list of two elements is

```
(cons X (cons Y ()))
```

# Some primitive (atomic) data types:

- numbers
  - integers (examples: 1, 4, -3, 0)
  - reals (examples: 0.0, 3.5, 1.23E+10)
  - rationals (e.g. 2/3, 5/2)

- symbols (e.g. fred, x, a12, set!)

- boolean: Scheme uses the special symbols **#f** and #t to represent false and true.

- strings (e.g. "hello sailor")

- characters (eg #\c)

# Some primitive (atomic) data types:

- Case is generally not significant (except in characters or strings). Note that you can have funny characters such as + or - or ! in the middle of symbols. (You can't have parentheses, though.) Here are some of the basic operators that scheme provides for the above datatypes.

# Operators

- Arithmetic operators (+, -, *, /, abs, sqrt)

- Relational (=, <, >, <=, >=) (for numbers)

- Relational (eqv?, equal?) for arbitrary data (more about these later)

- Logical (and, or, not): and and or are short circuit logical operators.

# Predicates

- Some operators are predicates, that is, they are truth tests. In Scheme, they return #f or #t. Peculiarity: in MIT Scheme, the empty list is equivalent to #f, and #f is printed as (). But good style is to write #t or #f whenever you mean true or false, and to write () when you really mean the empty list. Also see "Boolean Peculiarities" below.

```
number? integer? pair? symbol? boolean? string?

eqv? equal?

= < > <= >=
```

# Applying operators, functions

- Ok, so we know the names of a bunch of operators. How do we use them? Scheme provides us with a uniform syntax for invoking functions:

```
(function arg1 arg2 ... argN)
```

- This means all operators, including arithmetic ones, have prefix syntax. Arguments are passed by value (except with special forms, discussed later, to allow for nice things like short circuiting).

# Examples:

```
                                                        example1.rkt
#lang racket
(+ 2 3)
(abs -4)
(+ (* 2 3) 8)
(+ 3 4 5 1)
;; note that + and * can take an aribitrary number of arguments
;; actually so can - and / but you'll get a headache trying to remember
;; what it means
;; semicolon means the rest of the line is a comment
                                                                    5
                                                                    4
                                                                   14
                                                                   13
```

# Pair and List

- The cons function actually accepts any two values, not just a list for the second argument. When the second argument is not empty and not itself produced by cons, the result prints in a special way. The two values joined with cons are printed between parentheses, but with a dot (i.e., a period surrounded by whitespace) in between:

```
> (cons 1 2)
'(1 . 2)
> (cons "banana" "split")
'("banana" . "split")
```

# Pair and List

- Thus, a value produced by cons is not always a list. In general, the result of cons is a pair. The more traditional name for the cons? function is pair?, and we'll use the traditional name from now on.

- The name rest also makes less sense for non-list pairs; the more traditional names for first and rest are car and cdr, respectively. (Granted, the traditional names are also nonsense. Just remember that "a" comes before "d," and cdr is pronounced "could-er.")

# Pair

Examples:

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
> (pair? empty)
#f
> (pair? (cons 1 2))
#t
> (pair? (list 1 2 3))
#t
```

example2.rkt

```
#lang racket
(define alist (list 1 2 3 4 5))
alist
;;
(car alist)
(cdr alist)
(list? alist)
(pair? alist)
(define apair (cons 1 2))          '(1 2 3 4 5)
apair                               1
;;                                  '(2 3 4 5)
(car apair)                        #t
(cdr apair)                        #t
(list? apair)                      '(1 . 2)
(pair? alist)                      1
                                   2
                                   #f
                                   #t
```

# List Functions

**length** -- length of a list
**equal?** -- test if two lists are equal (recursively)
**eq?** -- test the reference of the two lists
**car** -- first element of a list
**cdr** -- rest of a list
**cons** -- make a new list cell (a.k.a. cons cell)
**list** -- make a list
**null?** -- is the list empty?
**pair?** -- is this thing a nonempty list?

# Variables

# Variables

- Scheme has both local and global variables. In Scheme, a variable is a name which is bound to some data object (using a pointer). There are no type declarations for variables. The rule for evaluating symbols: a symbol evaluates to the value of the variable it names. We can bind variables using the special form define:

```
(define symbol expression)
```

- Using define binds symbol (your variable name) to the result of evaluating expression. define is a special form because the first parameter, symbol, is not evaluated.

# Define a variable

**Define:**

```
(define x 1)
(set! x (+ x 3))    ;; update value
(cons x '())
```

**Query:**

> '(4)

# Variables

- The line below declares a variable called clam (if one doesn't exist) and makes it refer to 17:

```
(define clam 17)
clam              => 17
(define clam 23)  ; this rebinds clam to 23
(+ clam 1) => 24
(define bert '(a b c))
(define ernie bert)
```

- Scheme uses pointers: bert and ernie now both point at the same list.

# Variables

- we'll only use define to bind global variables, and we won't rebind them once they are bound, except when debugging.

# Lexically scoped variables with let and let*

- We use the special form let to declare and bind local, temporary variables.
  **Example:**

```
;; general form of let
(let ((name1 value1)
      (name2 value2)
      ...
      (nameN valueN))
   expression1
   expression2
   ...
   expressionQ)
```

# Lexically scoped variables with let and let*

```
;; reverse a list and double it
;; less efficient version:
(define (r2 x)
  (append (reverse x) (reverse x)))
;; more efficient version:
(define (r2 x)
  (let ((r (reverse x)))
       (append r r)))
```

# let*

- The one problem with Let is that while the bindings are being created, expressions cannot refer to bindings that have been made previously. For example, this doesn't work, since x isn't known outside the body:

```
(let ((x 3)
      (y (+ x 1)))
  (+ x y))
```

- To get around this problem, Scheme provides us with let*:

```
(let* ((x 3)
       (y (+ x 1)))
  (+ x y))
```

# Function

# Lambdas: Anonymous Functions

• You can use the lambda special form to create anonymous functions. This special form takes

```
 (lambda (param1 param2 ... paramk)   ; list of formals

          expr)                              ; body
```

lambda expression evaluates to an anonymous function that, when applied (executed), takes `k` arguments and returns the result of evaluating `expr`. As you would expect, the parameters are lexically scoped and can only be used in `expr`.

# Lambdas: Anonymous Functions

**Example:**
```
(lambda (x1 x2)
         (* (- x1 x2) (- x1 x2)))
```

- Evaluating the above example only results in an anonymous function, but we're not doing anything with it yet. The result of a lambda expression can be directly applied by providing arguments, as in this example, which evaluates to 49:
```
((lambda (x1 x2)
         (* (- x1 x2) (- x1 x2)))
  2 -5)    ; <--- note actuals here
```

# Defining Named Functions

- If you go to the trouble of defining a function, you often want to save it for later use. You accomplish this by binding the result of a lambda to a variable using define, just as you would with any other value. (This illustrates how functions are first-class in Scheme. This usage of define is no different from binding variables to other kinds of values.)

```
(define square-diff
        (lambda (x1 x2)
                (* (- x1 x2) (- x1 x2))))
```

# Defining Named Functions

• Because defining functions is a very common task, Scheme provides a special shortcut version of define that doesn't use lambda explicitly:

```
(define (function-name param1 param2 ... paramk)

        expr)
```

# Defining Named Functions

Here are some more examples using define in this way:

```
(define (double x)
         (* 2 x))


(double 4) => 8


(define (centigrade-to-fahrenheit c)
         (+ (* 1.8 c) 32.0))


(centigrade-to-fahrenheit 100.0) => 212.0
```

# Defining Named Functions

- The x in the double function is the formal parameter. It has scope only within the function. Consider the three different x's here...

```
(define x 10)

(define (add1 x)
   (+ x 1))

(define (double-add x)
   (double (add1 x)))

(double-add x)    => 22
```

# Defining Named Functions

- Functions can take 0 arguments:
  ```
  (define (test) 3)
  (test)  => 3
  ```

- Note that this is not the same as binding a variable to a value:
  ```
  (define not-a-function 3)
  not-a-function      => 3
  (not-a-function)  =>
  ;The object 3 is not applicable.
  ```

example3.rkt

```racket
#lang racket
(define (double x)
   (* 2 x))
(double 4)
(define (centigrade-tofahrenheit c) (+ (* 1.8 c) 32.0))
(centigrade-tofahrenheit 100.0)
;;
(define x 10)
(define (add1 x)(+ x 1))
(define (double-add x)(double (add1 x)))
(double-add x)
;;
(define (test) 3)
(test)
;;
(define not-a-function 3) ;; this is a variable
not-a-function
;;(not-a-function)
```

8
212.0
22
3
3

# Equality

# Equality and Identity: equal?, eqv?, eq?

- Scheme provides three primitives for equality and identity testing:

  1. **eq?** is pointer comparison. It returns #t iff its arguments literally refer to the same objects in memory. Symbols are unique ('fred always evaluates to the same object). Two symbols that look the same are eq. Two variables that refer to the same object are eq.

  2. **eqv?** is like **eq?** but does the right thing when comparing numbers. eqv? returns #t iff its arguments are eq or if its arguments are numbers that have the same value. eqv? does not convert integers to floats when comparing integers and floats though.

# Equality and Identity: equal?, eqv?, eq?

3. **equal?** returns true if its arguments have the same structure. Formally, we can define equal? recursively. equal? returns #t iff its arguments are eqv, or if its arguments are lists whose corresponding elements are equal (note the recursion). Two objects that are eq are both eqv and equal. Two objects that are eqv are equal, but not necessarily eq. Two objects that are equal are not necessarily eqv or eq. eq is sometimes called an identity comparison and equal is called an equality comparison.

# Example

```
(define clam '(1 2 3))
(define octopus clam)             ; clam and octopus refer to the same list
(eq? 'clam 'clam)           => #t
(eq? clam clam)             => #t
(eq? clam octopus)          => #t
(eq? clam '(1 2 3))         => #f ; (or (), in MIT Scheme)
(eq? '(1 2 3) '(1 2 3))     => #f
(eq? 10 10)                 => #t ; (generally, but implementation-dependent)
(eq? 10.0 10.0)             => #f ; (generally, but implementation-dependent)
(eqv? 10 10)                => #t ; always
(eqv? 10.0 10.0)            => #t ; always
(eqv? 10.0 10)              => #f ; no conversion between types
(equal? clam '(1 2 3))      => #t
(equal? '(1 2 3) '(1 2 3))  => #t
```

```racket
#lang racket
(define clam '(1 2 3))
(define octopus clam)
(eq? 'clam 'clam)
(eq? clam clam)
(eq? clam octopus)
(eq? clam '(1 2 3))
(eq? '(1 2 3) '(1 2 3))
(eq? 10 10)
(eq? 10.0 10.0)
(eqv? 10 10)
(eqv? 10.0 10.0)
(eqv? 10.0 10)
(equal? clam '(1 2 3))
(equal? '(1 2 3) '(1 2 3))
```

example4.rkt

#t
#t
#t
#f
#f
#t
#t
#t
#t
#f
#t
#t

## =

- Scheme provides = for comparing two numbers, and will coerce one type to another. For example, `(equal? 0 0.0)` returns <span style="color:red">#f</span>, but `(= 0 0.0)` returns <span style="color:green">#t</span>.

# Logical operators

# Logical operators

- Scheme provides us with several useful logical operators, including and, or, and not. Operators and and or are special forms and do not necessarily evaluate all arguments. They just evaluate as many arguments as needed to decide whether to return #t or #f (like the && and || operators in C++).

- However, one could easily write a version that evaluates all of its arguments.

# Logical operators

```
(and expr1 expr2 ... expr-n)
; return true if all the expr's are true
; ... or more precisely, return expr-n if all the expr's evaluate to
; something other than #f.  Otherwise return #f


(and (equal? 2 3) (equal? 2 2) #t)  => #f


(or expr1 expr2 ... expr-n)
; return true if at least one of the expr's is true
; ... or more precisely, return expr-j if expr-j is the first expr that
; evaluates to something other than #f.  Otherwise return #f.
```

# Logical operators

```
(or (equal? 2 3) (equal? 2 2) #t)  => #t

(or (equal? 2 3) 'fred (equal? 3 (/ 1 0)))  => 'fred

(define (single-digit x)
    (and (> x 0) (< x 10)))

(not expr)
; return true if expr is false

(not (= 10 20))      => #t
```

# Boolean Peculiarities

- In R4 of Scheme the empty list is equivalent to #f, and everything else is equivalent to #t. However, in R5 the empty list is also equivalent to #t! Moral: only use #f and #t for boolean constants.

# Conditionals

# **if** special form

   (**if condition** `true_expression` `false_expression`)

- If condition evaluates to true, then the result of evaluating true_expression is returned; otherwise the result of evaluating false_expression is returned. if is a special form, like quote, because it does not automatically evaluate all of its arguments.

# if special form

```
(if (= 5 (+ 2 3)) 10 20)      => 10
(if (= 0 1) (/ 1 0) (+ 2 3)) => 5
; note that the (/ 1 0) is not evaluated

(define (my-max x y)
    (if (> x y) x y))


(my-max 10 20)                        => 20

(define (my-max3 x y z)
    (if (and (> x y) (> x z))
        x
        (if (> y z)
            y
            z)))
```

# cond -- a more general conditional

- The general form of the cond special form is:

```
(cond (test1 expr1)
      (test2 expr2)
      ....
      (else exprn))
```

- As soon as we find a test that evaluates to true, then we evaluate the corresponding expr and return its value. The remaining tests are not evaluated, and all the other expr's are not evaluated. If none of the tests evaluate to true then we evaluate exprn (the "else" part) and return its value. (You can leave off the else part but it's not good style.)

# Example

```
(define (weather f)
    (cond ((> f 80) 'too-hot)
          ((> f 60) 'nice)
          ((< f 35) 'too-cold)
          (else 'typical-seattle)))
```

```racket
#lang racket                                    example5.rkt
(define (my-max x y)
  (if (> x y) x y))
(my-max 10 20)
;;
(define (my-max3 x y z)
  (if (and (> x y) (> x z))
      x
      (if (> y z)
          y
          z)))
(my-max3 2 5 1)
;;
(define (weather f)
  (cond ((> f 80) 'too-hot)
        ((> f 60) 'nice)
        ((< f 35) 'too-cold)
        (else 'typical-seattle)))
(weather 10)
(weather 40)
(weather 70)
(weather 100)
```

20
5
'too-cold
'typical-seattle
'nice
'too-hot

# Scheme Cheat Sheet

# Scheme Cheat Sheet

- Scheme is an extraordinarily simple language. It implements just a few fundamental concepts that can be combined in interesting and powerful ways to build up complex systems.

# The Read-Eval-Print loop

- This is the heart of the scheme interpreter. The evaluator implements the following algorithm:

- **Eval(expr: symbolic-expression)**
  1. Is expr an atom? If it is self-evaluating, just return it. Otherwise (it is a name), return its value from the current environment
  2. Else, is the expression a list? If the first element is a special form, do the rule for that form. Otherwise (it must be a function), evaluate the first element of the list and apply it to the values of the rest of the list.

# Special Forms: define, quote, lambda, cond, let/let*

```scheme
;; Define is used to define new names.  Names may refer to any value
;; (which may be data or a function)
(define x 10)
(define double (lambda (x) (* x 2)))


;; Quote is used to "quote" literal data (symbols or lists)
(quote hello)   => hello
(quote (1 2 3)) => (1 2 3)
'(1 2 foo bar)  => (1 2 foo bar)  ; the tick-mark ' is syntactic sugar
```

# Special Forms: define, quote, lambda, cond, let/let*

```
;; Lambda is used to generate new functions

(lambda (x) (+ x 10))                      ; an anonymous function

(define plus10 (lambda (x) (+ x 10)))    ; we've named the function now


;; Cond is a general conditional

(cond

  ((eq? 'foo 'bar) 'hello)

  ((= 10 20) 'goodbye)

  (#t 'sorry))                    => sorry
```

# Special Forms: define, quote, lambda, cond, let/let*

```
;; Let is used to declare/use temporary variables
(let

   ((x 10)

    (y 20))

   (+ x y))
```

```racket
#lang racket
(define x 10)
(define double (lambda (x) (* x )))
;;
(quote hello)
(quote (1 2 3))
'(1 2 foo bar)
;;
(lambda (x) (+ x 10))
(define plus10 (lambda (x)(+ x 10)))
(plus10 3)
;;
(cond ((eq? 'foo 'bar) 'hello)
      ((= 10 20) 'goodbye)
      (#t 'sorry))
;;
(let ((x 10)(y 20)) (+ x y))
```

```
'hello
'(1 2 3)
'(1 2 foo bar)
#<procedure:...eme/cheatsheet1.rkt:7:0>
13
'sorry
30
```

# Built-in Types and Functions

- Scheme supports numbers (integers, rationals, floats), characters, strings, booleans, symbols, lists, and vectors. Below are some examples of the built-in functions we can use on these types:

# Built-in Types and Functions

```
;; arithmetic:  +, -, *, /

;; relational: <, <=, >, >=, =

(+ 1 2)                          => 3
(= 1 2)                          => #f    ; use = for numbers
```

# Built-in Types and Functions

```
;; Equality and identity:   eq? and equal?

(eq? 'hello 'goodbye)        => #f   ; eq? is an identity test

(eq? 'hello 'hello)          => #t   ; two values are eq if they are the same

(eq? '(1 2) '(1 2))          => #f   ; object...

(equal? '(1 2) '(1 2))       => #t   ;

(define foo '(1 2))

(define bar foo)

(eq? foo bar)                => #t

(equal? foo bar)             => #t   ; two values are equal if they look the same

(equal? foo '(1 2))          => #t
```

```racket
#lang racket
(+ 1 2)                                                          3
(- 1 2)                                                          -1
;;
(eq? 'hello 'goodbye)                                           #f
(eq? 'hello 'hello)                                             #t
(eq? '(1 2) '(1 2))          ;; different objects               #f
(equal? '(1 2) '(1 2))                                          #t
(define foo '(1 2))
(define bar foo)
(eq? foo bar)               ;; same reference                   #t
(equal? foo bar)                                                #t
(equal? foo '(1 2))                                             #t
```

# Built-in Types and Functions

```
;; Lists:  cons, car, and cdr
;; Making new lists, via quoting, cons, or list
(define foo '(1 2 3))
(define bar (cons 1 (cons 2 (list 3))))   #scheme allow (cons 3 ())
(define baz (list 1 2 3))                  # but not racket
```

# Built-in Types and Functions

```scheme
;; Process lists with car, cdr, and null?
(null? '(1 2))                 => #f
(null? ())                     => #t
(car '(1 2))                   => 1
(cdr '(1 2))                   => (2)
```

```racket
#lang racket
;;
(define foo '(1 2 3))
(define bar (cons 1 (cons 2 (list 3))))
(define baz (list 1 2 3))
;;
foo
bar
baz
;;
(null? '(1 2))
(null? '())
(car '(1 2))
(cdr '(1 2))
```

```
'(1 2 3)
'(1 2 3)
'(1 2 3)
#f
#t
1
'(2)
```

# Iteration via recursion

```
;; Exponentiation function x^n
(define exp
  (lambda (x n)
    (cond ((= n 0) 1)
      (#t (* x (exp x (- n 1)))))))


;; List length
(define length
  (lambda (a-list)
    (cond ((null? a-list) 0)
      (#t (+ 1 (length (cdr a-list)))))))
```

# Higher order functions

- Functions are first-class in Scheme, meaning we can pass them as arguments to other functions.

```scheme
;; takes two functions and an argument, returns (f (g x))
(define compose
  (lambda (f g x)
    (f (g x))))

(compose even? (lambda (x) (- x 1)) 10)    => #f

;; takes a function and applies it to every element of a list
(define map
  (lambda (f a-list)
    (cond ((null? a-list) a-list)
      (#t (cons (f (car a-list)) (map f (cdr a-list)))))))

(map even? '(1 2 3 4))            => (#f #t #f #t)
```

```racket
#lang racket                                          cheatsheet4.rkt
;; Exponential funcion x^n
(define exp
   (lambda (x n)
      (cond ((= n 0) 1)
            (#t (* x (exp x (- n -)))))))))
;;
(define length
    (lambda (alist)
       (cond ((null? alist 0)
              (#t (+ 1 (length (cdr alist)))))))))
;;
(define compose
   (lambda (f g x)
      (f (g x)))))
(compose even? (lambda (x) (- x 1)) 10)
;;
(define map
   (lambda (f alist)
      (cond ((null? alist) alist)
            (#t (cons (f (car alist))(map f (cdr alist)))))))   #f
(map even? '(1 2 3 4))                                          '(#f #t #f #t)
```

# Functional Programming in Scheme

# Lamba calculus

- A calculus is a "method of analysis … using special symbolic notation"
  - In this case, a way of describing mathematical functions
- Syntax:
  - $f(x) = x + 3$:          $\lambda$, x, x+3
  - $f(3)$                        ($\lambda$, x, x+3) 3
  - $f(x) = x^2$               $\lambda$, x, x*x
- In pure lamba calculus, EVERY function takes one (and only one) argument

# Lamba calculus

- So how to do functions with multiple arguments?
  - f(x,y) = x-y                    $\lambda$ x, $\lambda$ y, x-y
    - This is really a function of a function
    - ($\lambda$ x, $\lambda$ y, x-y) 7 2            yields f(7,2) = 7-2 = 5
    - ($\lambda$ x, $\lambda$ y, x-y) 7    yields f(7,y) = y-x
- Note that when supplying only one parameter, we end up with a function
  - Where the other parameter is supplied
- This is called *currying*
  - A function can return a value *OR* a function

# Functions

- Map an element from the domain into the range

- Domain can be?

- Range can be?

- No side effects (this is how we would like our Scheme functions to behave also)

- Can be composed

# Scheme

- List data structure is workhorse

- Program/Data equivalence

- Heavy use of recursion

- Usually interpreted, but good compilers exist

# Eval

- Eval takes a Scheme object and an environment, and evaluates the Scheme object.

```
(define x 3) (define y (list '+ x 5))
(eval y user-initial-environment)
```

- The top level of the Scheme interpreter is a read-eval-print loop: read in an expression, evaluate it, and print the result.

# Eval



Screenshot of DrRacket (a2.rkt):

```
#lang racket
(define y 1)
1

(eval '(+ 2 1))
```

Annotation: Not allowed in the define area

Interactions window:

```
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
1
+: unbound identifier;
 also, no #%app syntax transformer is bound in: +
> (eval '(+ 2 1))
3
> (eval '(+ y 4))
5
```

Annotation: **Evaluate the lists based on defines**

```
1    #lang racket
2    (define a 'b)
3    (define b 'c)
4    (define c 50)
5
```

Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
'b
b: unbound identifier;
 also, no #%top syntax transformer is bound in: b
> a
'b
> (eval a)
'c
> (eval (eval a))
50
>
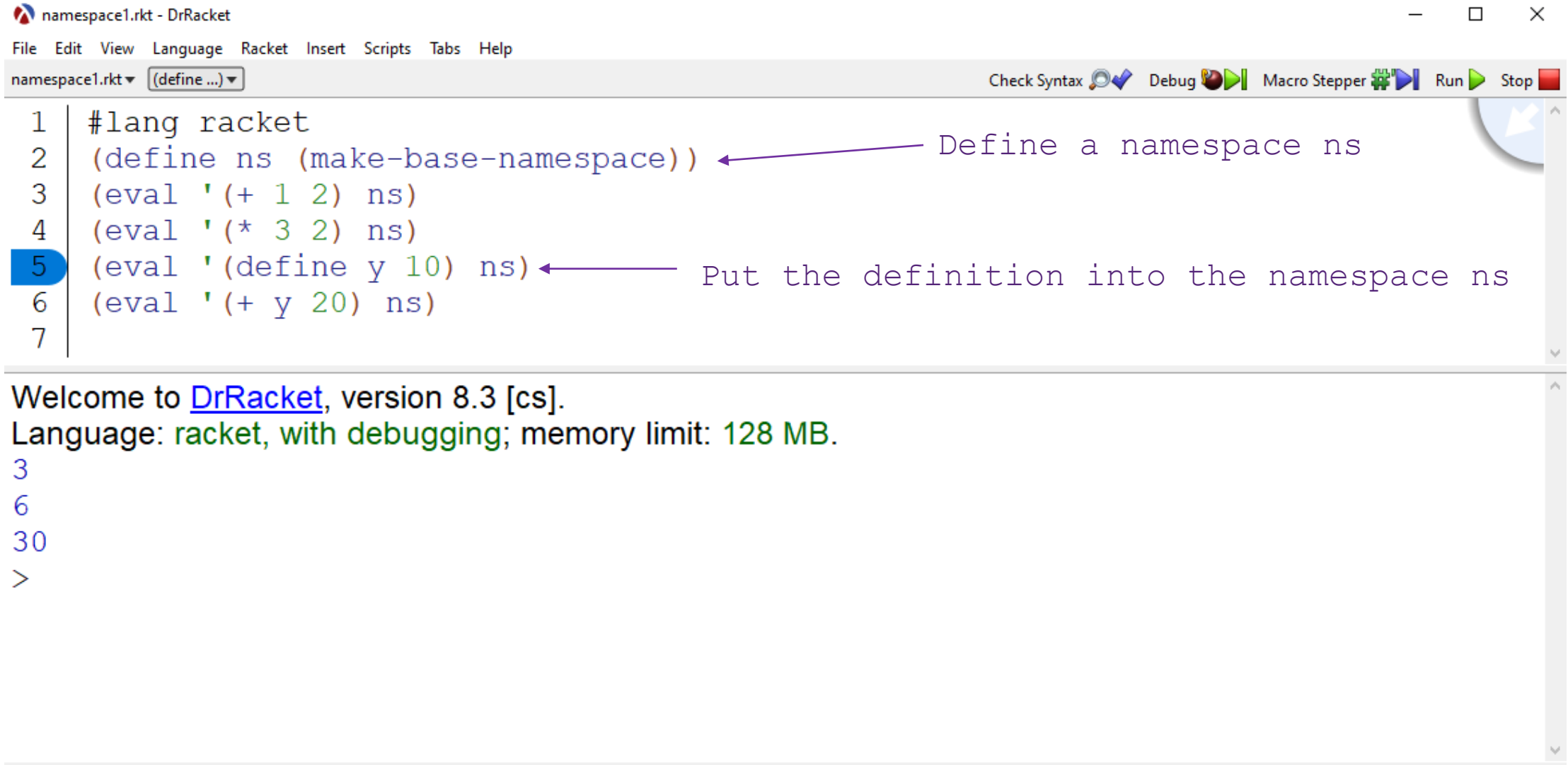
# Namespaces

- The `eval` function used with just one argument evaluates that argument using the **top-level bindings** for names.

- You can also call it with an additional optional namespace argument, to cause evaluation in some different environment. See the eval section of the Racket Guide for details.

- The key point to remember is that evaluation takes place within **some environment** that determines the bindings for names.

namespace1.rkt - DrRacket

File   Edit   View   Language   Racket   Insert   Scripts   Tabs   Help

namespace1.rkt ▾   (define ...) ▾          Check Syntax 🔍✔   Debug 🐞▶|   Macro Stepper #▶|   Run ▶   Stop ■

```racket
1  #lang racket
2  (define ns (make-base-namespace))          ← Define a namespace ns
3  (eval '(+ 1 2) ns)
4  (eval '(* 3 2) ns)
5  (eval '(define y 10) ns)          ← Put the definition into the namespace ns
6  (eval '(+ y 20) ns)
7
```

Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
3
6
30
>
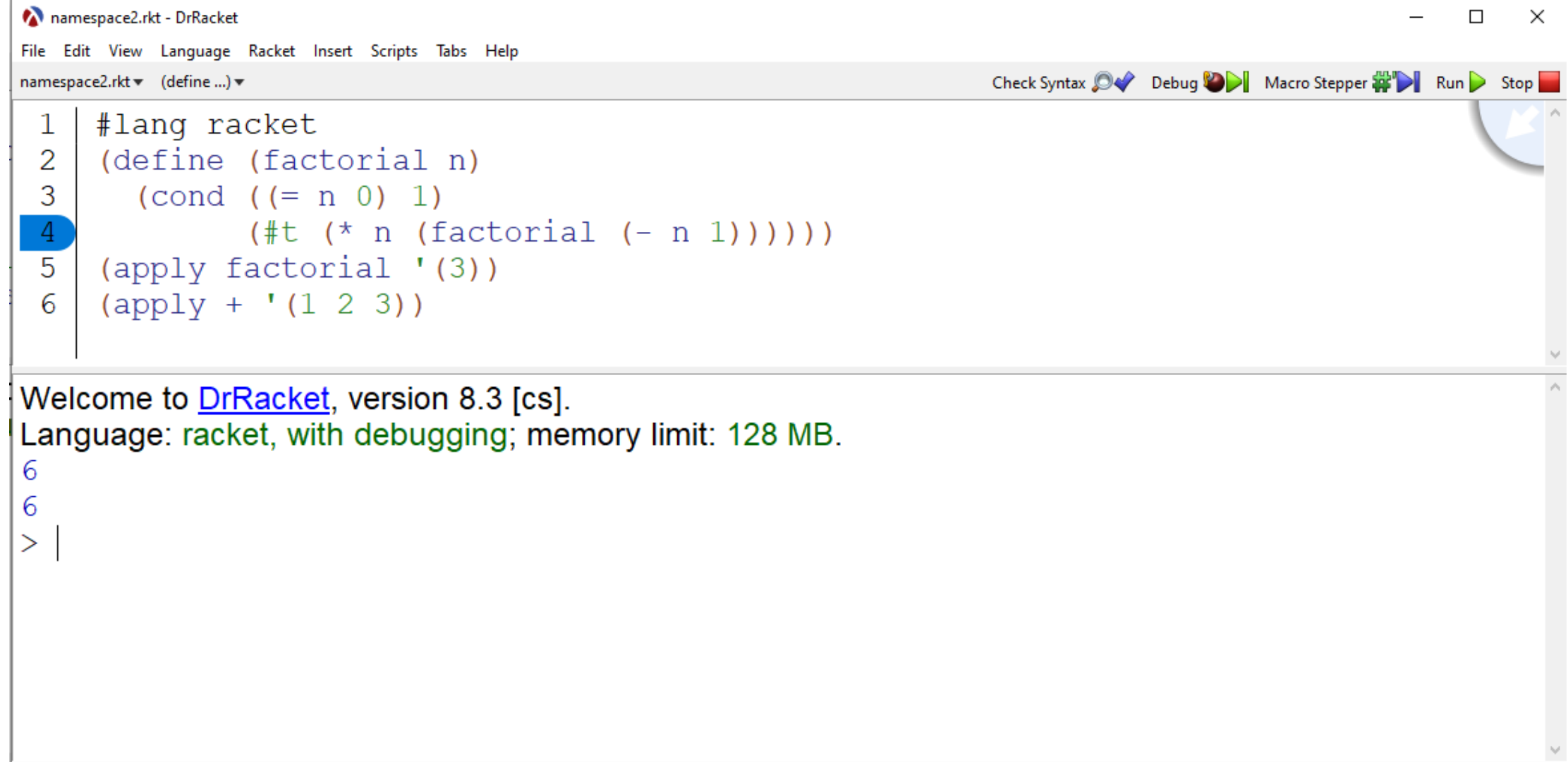
# Apply

- The apply function applies a function to a list of its arguments.

**Examples:**

```
(apply factorial '(3))
(apply + '(1 2 3 4))
```

namespace2.rkt

# More Scheme Features

- Static/Lexical scoping

- Dynamic typing

- Functions are first-class citizens

- Tail recursion is optimized

# First class citizens in a PL

- Something is a first-class object in a language if it can be manipulated in "any" way": (example ints and chars)
  - passed as a parameter
  - returned from a subroutine
  - assigned into a variable

# Higher Order functions

- Higher Order functions – take a function as a parameter or returns a function as a result.


Example: the map function
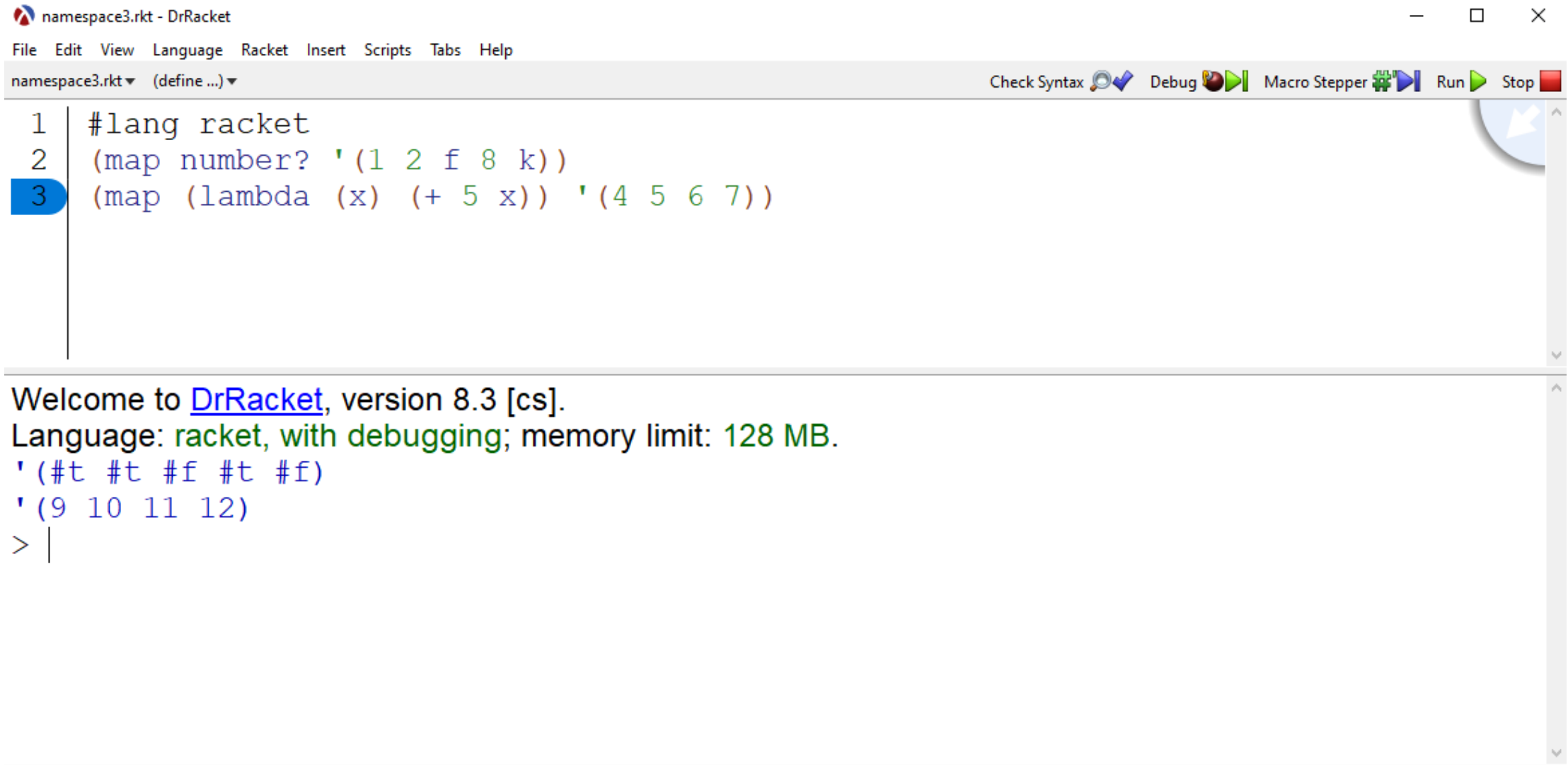
```
(map function list)
(map number? `(1 2 f 8 k))
(map (lambda (x) (+ 5 x)) `(4 5 6 7))
```
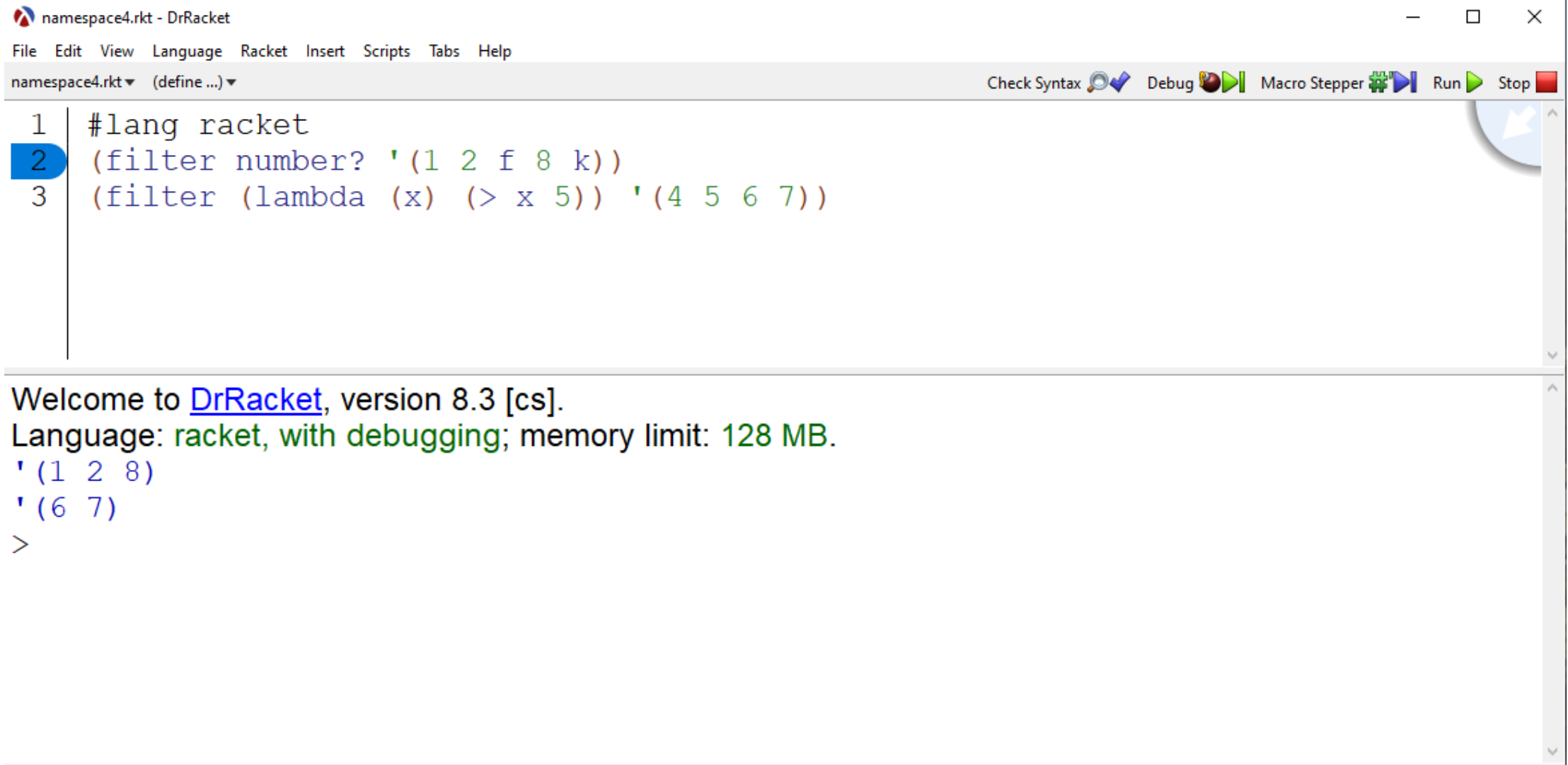
# Functional Programming

- **map:** racket supported

- **filter:** racket supported

- **reduced:** racket not supported

namespace3.rkt - DrRacket

File   Edit   View   Language   Racket   Insert   Scripts   Tabs   Help

namespace3.rkt ▼   (define ...) ▼                                              Check Syntax 🔍✔   Debug 🔴▶|   Macro Stepper #▶|   Run ▶   Stop ■

```
1   #lang racket
2   (map number? '(1 2 f 8 k))
3   (map (lambda (x) (+ 5 x)) '(4 5 6 7))
```
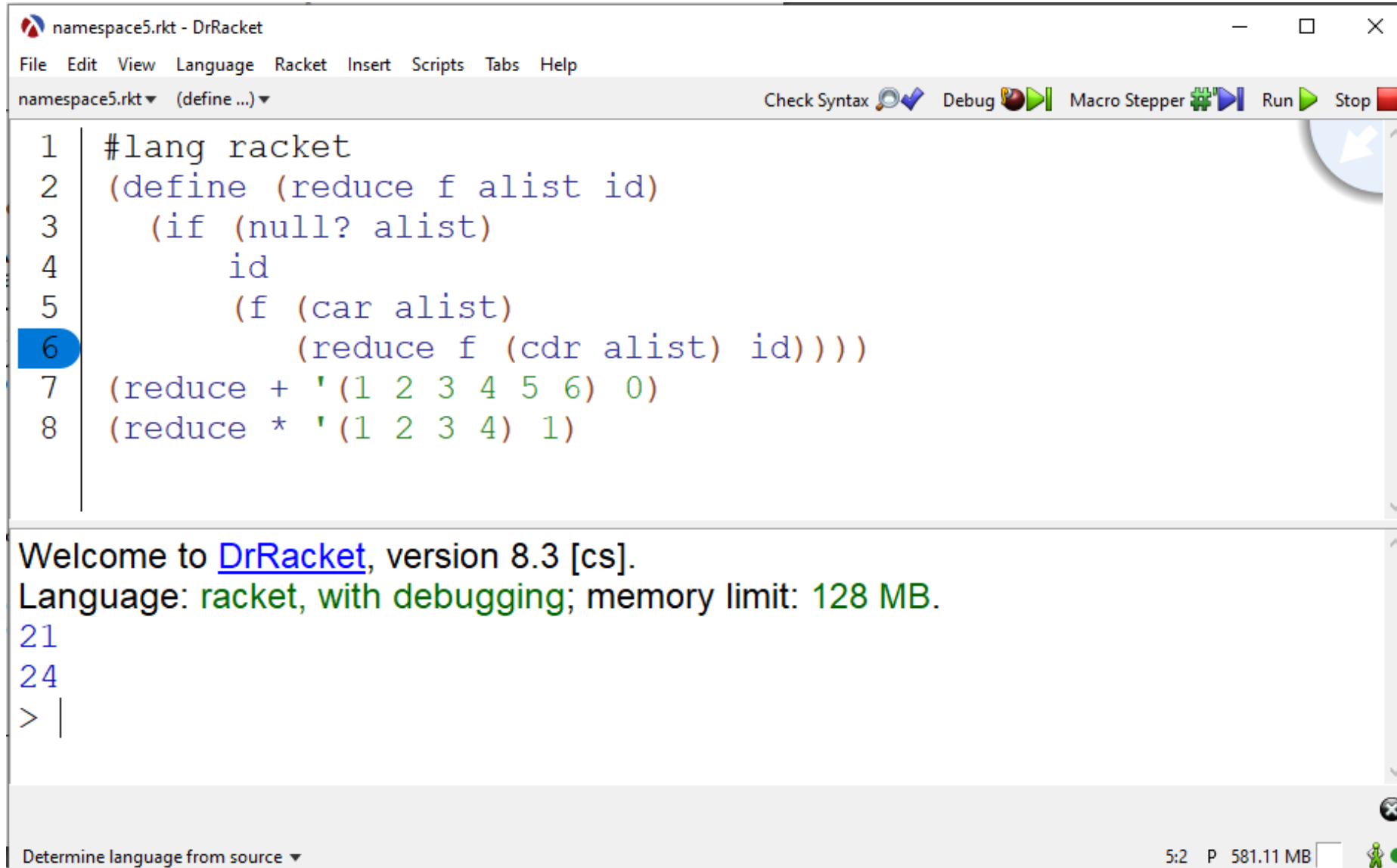
Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
'(#t #t #f #t #f)
'(9 10 11 12)
>

namespace4.rkt - DrRacket

File   Edit   View   Language   Racket   Insert   Scripts   Tabs   Help

namespace4.rkt ▾   (define ...) ▾                              Check Syntax 🔍✔   Debug 🐞▶|   Macro Stepper #▶|   Run ▶   Stop ■

```racket
#lang racket
(filter number? '(1 2 f 8 k))
(filter (lambda (x) (> x 5)) '(4 5 6 7))
```

Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
'(1 2 8)
'(6 7)
>

```racket
#lang racket
(define (reduce f alist id)
  (if (null? alist)
       id
       (f (car alist)
          (reduce f (cdr alist) id))))
(reduce + '(1 2 3 4 5 6) 0)
(reduce * '(1 2 3 4) 1)
```

Welcome to DrRacket, version 8.3 [cs].
Language: racket, with debugging; memory limit: 128 MB.
21
24
>

# Implementation Issues

- Variety of implementations
- Conceptually (at least)
  - Everything is a pointer
  - Everything is allocated on the heap
    - (and garbage collected)
- Reality
  - Often a small run-time written in C or C++
  - Many optimizations used

# Augmenting Recursion

- Builds up a solution:

```
(define (func x)
(if end-test end-value
  (augmenting-function augmenting-value (func reduced-
  x))))
```

- Factorial is the classic example:

```
(define (factorial n)
   (if (= n 0) 1
   (* n (factorial (- n 1)))))
```

# Tail Recursion

- In tail recursion, we don't build up a solution, but rather, just return a recursive call on a smaller version of the problem.

```
(define (func x)
  (cond (end-test-1 end-value-1)
        (end-test-2 end-value-2)
        (else (func reduced-x))))
```

# Tail Recursion

```
(define (all-positive x)
 (cond ((null? x) #t)
    ((<= (car x) 0) #f)
    (else (all-positive (cdr x)))))
```

• The recursive call would be recognized and implemented as a loop.

# Applicative Order

- Arguments passed to functions are evaluated before they are passed.

# Special Forms

- Examples:
    - define, lambda, quote, if, let, cond, and, or


- Arguments are passed to special forms WITHOUT evaluating them.
- Special forms have their own rules for evaluation of arguments.
- This is known as normal order

# Scheme example

- Construct a program to simulate a DFA
  - From Scott, p. 521 (3$^{rd}$ ed.)

- Three main functions
  - **simulate:** the main function, calls the others
  - **isfinal?:** tells if the DFA is in a final state
  - **move:** moves forward one transition in the DFA
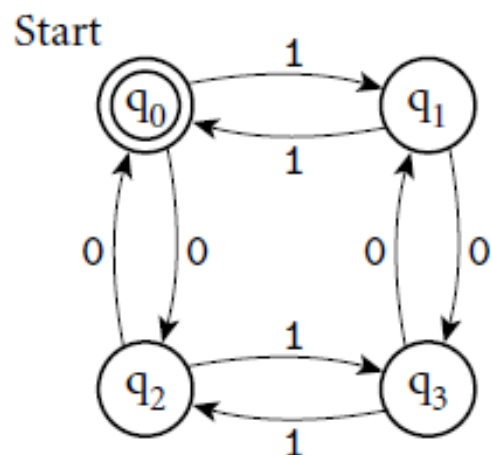
```
(define zero-one-even-dfa
  '(q0                                                    ; start state
    (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)     ; transition fn
     ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
    (q0)))                                                ; final states
```

Figure 10.2 DFA to accept all strings of zeros and ones containing an even number of each.
At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 10.1.

```racket
#lang racket
(define simulate
  (lambda (dfa input)
    (cons (car dfa)
          (if (null? input)
              (if (isfinal? dfa) '(accept) '(reject))
              (simulate (move dfa (car input))
                        (cdr input))))))
;;
(define isfinal?
  (lambda (dfa)
    (memq (car dfa) (caddr dfa))))
;;
```
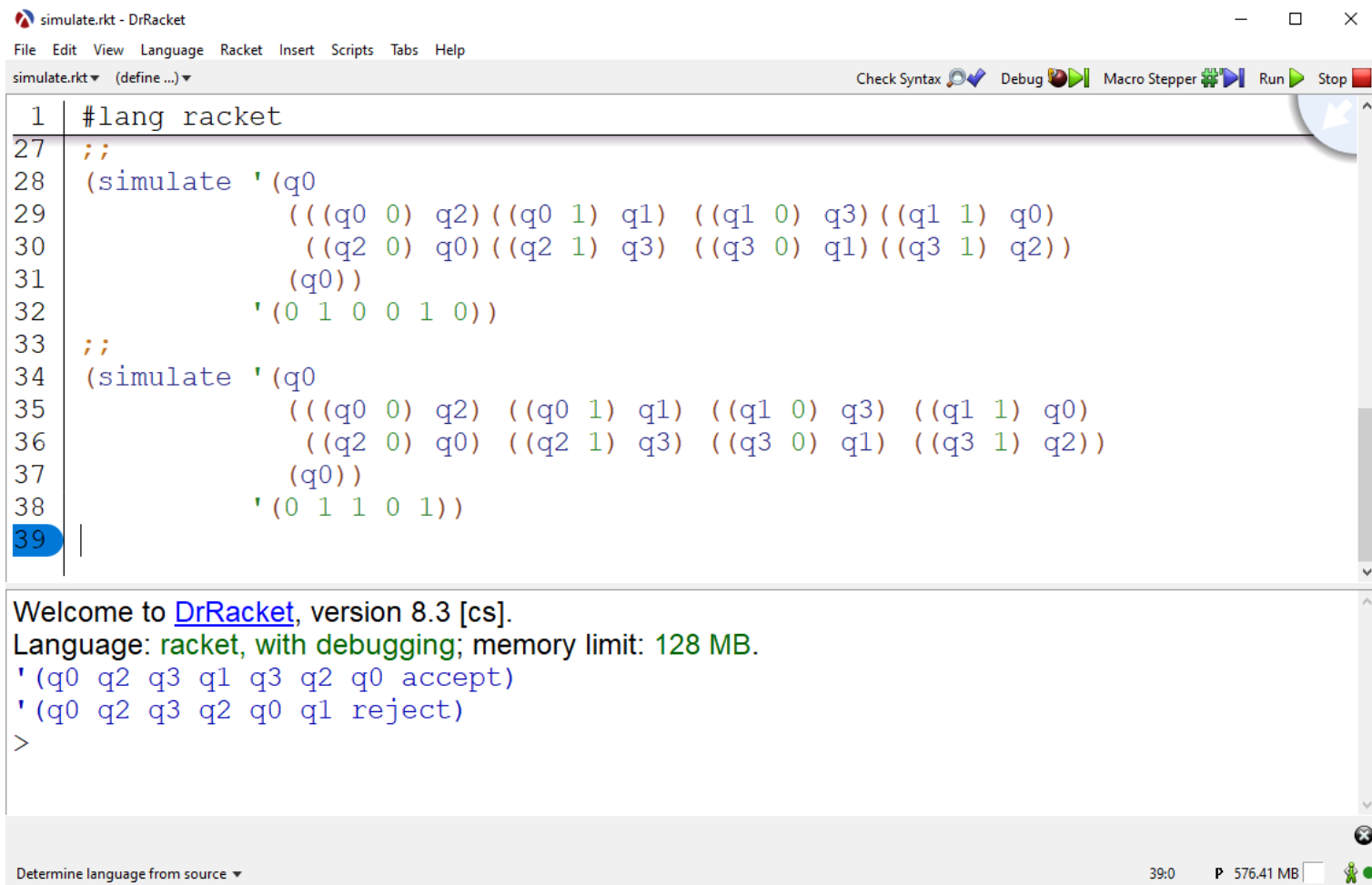
```
(define move
  (lambda (dfa symbol)
    (let ((curstate (car dfa))(trans (cadr dfa))
                             (finals (caddr dfa)))
      (list
       (if (eq? curstate 'error)
       'error
       (let ((pair (assoc (list curstate symbol)
                          trans)))
         (if pair (cadr pair) 'error)))
        trans
        finals)))))
;;
;;
```

```
(simulate '(q0
            (((q0 0) q2)((q0 1) q1) ((q1 0) q3)((q1 1) q0)
             ((q2 0) q0)((q2 1) q3) ((q3 0) q1)((q3 1) q2))
            (q0))
           '(0 1 0 0 1 0))
;;
(simulate '(q0
            (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)
             ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
            (q0))
           '(0 1 1 0 1))
```

Dr. Racket Results:

# Summary

# Scheme and Python

- Java and Python are very powerful real-world languages, but they often struggle as languages for learning. The extra syntax they have for enabling the programmer to easily access the more advanced features quickly complicate the whole matter, thus defeating the point of being easy to learn.

- Scheme supports many features often only found in interpreted languages such as closures, first class functions and advanced meta-programming, even though Scheme can be compiled easily. In fact, many of these features which are today almost exclusively associated with interpreted languages started with Scheme and unlike Python, JavaScript and Ruby which inherited many features from Scheme, Scheme is a minimalistic language.

# End of Chapter 7