# Racket Quick Refresher

*CS 241*

*David Duan / Dec. 13, 2018*

## Contents

# 1 Racket

## 1.1 Racket vs. Scheme

Racket can be thought of as a dialect of R5RS Scheme plus many useful extensions (not a strict superset!). One major difference is R5RS functiosns `set-car!` and `set-cdr!` are not provided in Racket. To constuct mutable lists, use `mcons` and `mlist` functions or use immutable lists that contain `boxes`, mutable cells holdings a single value.

**Remark** Start your program with the line `#lang racket` to enable all Racket features.

## 1.2 Racket Command Line

To run your programs on the command line, use `racket myfile.rkt`. The `racket` binary is included with the DrRacket distribution, but you may need to set up your shell paths. Since Marmoset uses `racket` to test your programs, your final round of local testing should always use command line binary.

**Remark** To suppress the top-level expression output, you can wrap it with `(void ...)`.

# 2 Language Fundamentals

## 2.1 Quick Hello World

```
;; This is a comment
#|
   This is a
   Multi-line
   Comment
|#

(define str "Hello World") ;; Define a constant
(display str)              ;; Invoke the display function to print the value of str
```

## 2.2 Identifiers and Scope

An *identifier*, also known as a *variable*, is a name that appears in a program, whose meaning is determined by the identifier's *binding*. By convention, identifiers in Racket are typically written in lowercase, with hyphens between words.

Every identifier binding has a *scope* that determines where in the program the binding is visible. Outside that scope, the binding won't work. The scope of a binding depends on how and where the identifier is introduced.

**Remark** Bindings can be defined in one module and shared with another (**Section 2.7**).

## 2.3 Booleans and Conditionals

### 2.3.1 Booleans

In Racket, `#t` and `#f` represent True and False. When used in a conditional expression, every value other than `#f` counts as true; there is no "false-ish" values.

```
> (if "" 'true 'false)
'true
> (if '() 'true 'false)
'false
```

Boolean operators `and` and `or` work in usual short-circuiting way.

### 2.3.2 Conditionals

The `if` expression has two branches, one for true case and one for false case. If you only need the true case, use `when`; if you only need the false case, use `unless`.

```
> (define right #t)
> (define wrong #f)
> (if right 'true 'false)
'true
> (if wrong 'true 'false)
'false
> (when wrong 'true)     ;; only execute the clause if the condition is #t
> (when right 'true)     ;; only execute the clause if the condition is #t
'true
> (unless right 'true)   ;; only execute the clause if the condition is #f
> (unless wrong 'false)  ;; only execute the clause if the condition is #f
'true
```

### 2.3.3 Multiple Branches: `cond` and `case`

The select branch `cond` supports any number of conditions and an optional `else` clause. It stops at the first one that *not evaluated to* `#f`. If no match is found and no `else` is provided, you get `<#void>`.

The `case` expression creates a conditional that compares a value against lists of literal values. If the value is `equal?` to one of the items in the list on the left, then the right side of the branch is evaluated and returned as teh result. You can also provide an `else`.

### 2.3.4 Even better: `match`

The `match` expression is a more general version of `case`: it supports a wider pattern vocabulary that can match literal values, including predicts, lists, pairs, structures and other values. Moreover, `match` can also assign elements of the match to internal variables for further processing.

## 2.4 Stringlike Types

### 2.4.1 Characters

A *character* corresponds to a Unicode scalar value. A printable character normally prints as `#\` followed by the represented character, e.g. `#\λ` or `#\a`. An unprintable character normally prints as `#\u` followed by the scalar value as hexadecimal number. Two common exceptions are `#\space` and `\newline`, which represent the space and linefeed character, respectively.

### 2.4.2 Strings

A standard *string* is enclosed in double quotes, e.g. `"This is a string"`. To make multi-line strings, insert `\n` between lines, e.g. `"This is\na multi-line\nstring."`.

Always use `equal?` for string comparisons as `eq?` might not behave as you've expected:

```
> (define str1 "abc")
> (define str2 (string-append "a" "b" "c"))
> (equal? str1 str2)
#t ;; value comparison: str1 and str2 both have values "abc"
> (eq? str1 str2)
#f ;; pointer comparison: str1 and str2 point to different chunk of memory
```

### 2.4.3 Symbols

A *symbol* is like a string, but written with a single `'` prefix rather than surrounding double quotes. A symbol cannot contain spaces unless the whole symbol is surrounded with verticla bars `|`.

Symbols are *interned*, whose allocation and storage is specially managed by Racket so that each unique instance of that data type only appears once in the memory. For this reason, symbols can be compared using the fastest comparison, `eq?`, which relies on pointer comparisons rather than value comparisons.

### 2.4.4 Path and Path Strings

A *path string* is not a distinct type, but rather a subset of the strings that can be converted into a valid *path*, which represents a file location. Unlike strings, paths are made of a sequence of bytes rather than Unicode characters.

## 2.5 Numbers

Numbers in Racket are similar to those you'd find in other languages, with two caveats:

1. All numbers are *complex* numbers, meaning they have a real part and an imaginary part. In practice, this means you get complex numbers in your environment "for free" without the need to import external libraries.
2. Racket distinguishes *exact* and *inexact*.
    a. An *exact* number is a number whose real and complex parts are both integers or rational fractions. By default, Racket uses exact numbers when it can. This means accurate arithmetic is possible: `(+ 1/3 1/5)` equals `8/15`.

b. An *inexact* number is anything else, including all floating-point number. To force an exact number to behave as an inexact number, add decimal to it. The result of an operation between an inexact number and another number will always be inexact, as you would expect.

## 2.6 Basic Data Structure: Pairs and Lists

A *Pair* is an immutable data structure that holds two values.

- To create a pair: `(cons 1 2)`.
- To retrieve the first/second value: `(car (cons 1 2))` and `(cdr (cons 1 2))`.

Extending pairs to more than two elements, you get a *list*. A list can contain any kind and any number of values. If a list has zero values, it's called an *empty list*.

- To create a list: `(cons 1 (cons 2 3))`, `(list 1 2 3)` or `'(1 2 3)`.
- Predicate: `(list? lst)`, `(empty? lst)`
- Accessor: `car` or `first`; `cdr` or `rest`; `(list-ref lst pos)`
- Utility: `(lenth lst)`, `(flatten lst)`, `(append lst)`, etc.
- Higher-order functions: `apply`, `map`, `filter`, etc.

**Remark** By default, square and curly brackets work the same way as parentheses. In practice, parentheses are preferred; square and curly brackets are reserved for special situations where they improve readability.

**Remark** We will discuss higher-order function in detail later, see **Section 3.6**.

### 2.6.1 Quasiquote

Racket has a convenient syntax for representing data literals: prefix any expression with `'` (single quote) and the expression, rather than being evaluated, will be returned as data. Sometimes one doesn't want to escape evaluation of an entire list. In this case, one can use the `` ` `` (backquote) operator, also called *quasiquote* with the `,` (comma) operator, also called the *unquote*. Quasiquote behaves like quote, except that unquoted expressions are evaluated:

```
'(1 2 (+ 3 4))   ;; => (list 1 2 (list + 3 4))
`(1 2 ,(+ 3 4))  ;; => (list 1 2 7) or '(1 2 7)
```

Racket also has special forms for these three operators:

```
(quote (1 2 (+ 3 4)))              ;; => (list 1 2 (list + 3 4))
(quasiquote (1 2 unquote(+ 3 4)))  ;; => (list 1 2 7) or '(1 2 7)
```

You can also splice in the contents of an expression which evaluates to a list with `@` (at):

```
`(1 2 ,@(build-list 3 add1)) ;; => '(1 2 1 2 3)
```

## 2.7 Importing and Exporting

By default, all bindings defined in a module are private to that module. To export a binding, use `(provide var-name)`. It is recommended to write `provide` at the top as a way to summarize the public interface of the module.

- To export every binding newly defined within a module: `(provide (all-defined-out))`.
- To export bindings imported from another module: `(provide (all-from-out <path>))`

Bindings can be imported from other modules with `require`.

## 2.8 Subtleties

### 2.8.1 Object Identity and Comparisons

There are four comparison operators in Racket:

- `equal?` is the safest but slowest comparison; it always gives the correct answer.
- `eq?` is faster as it does not inspect values but rather uses pointer comparison.
- `eqv?` is like `eq?` but handles numbers and characters consistently.
- `=?` is like `eqv?` but limited to numbers.

### 2.8.2 The `#lang` Header

Every Racket source file begins with a `#lang` (*hash-lang*) line, which specifies what *language* should be used to interpret the source code in the file.

The shorthand `#lang name` notation only works with langauges that have been installed as part of a Racket package. Alternatively, you can use the `#lang reader <path>` syntax to specify a local path to a language reader. Since we are not dealing with macros or building our own languages, this topic will not be further discussed.

### 2.8.3 Contracts

A *contract* operates at run time and restricts the values that can pass across the boundary between two parts of a program. They are primarily used with functions to check that input arguments and return values meet certain requirements. This makes debugging much easier.

```
(require racket/contract)
(define/contract (our-div num denom)
                 (number? (and/c number? (not/c zero?)) . -> . number?)
                 (/num denom)
```

# 3 Functions

A *function* is a set of expressions that's only evaluated when explicitly invoked at *run time*. Within a program, functions are used for three overlappting reasons:

1. To *delay* the evaluation of a set of expressions.
2. To allow *repeated* evaluation of a set of expressions.
3. To *generalize* a set of expressions by using input arguments.

If a function is created with `define`, it uses an *identifier* as its name. Alternatively, functions can be created using `lambda` with no explicit name provided; such functions are called *anonymous functions*.

**Remark** We won't be going into theoretical material such as lambda calculus here; this is a refresher on Racket language, not functional programming in general. You can take CS 145 if you are interested.

## 3.1 Argument

### 3.1.1 Positional Argument vs. Keyword Argument

By default, all function arguments are *positional arguments*, meaning they are bound to identifiers within the function according to where they appear in the argument list. Racket also supports named *keyword arguments* that can appear anywhere in the argument list:

```
> (define greet
    (lambda (given #:last surname)
      (string-append "Hello, " given " " surname)))
> (greet "John" #:last "Smith")
"Hello, John Smith"
> (greet #:last "Doe" "Jane")
"Hello, Jane Doe"
```

### 3.1.2 Setting Default Values

Any positional or keyword argument can have a default value by specifying that value in square brackets. As usual, arguments without default values must come first:

```
> (define (my-add x [y 10] #:z [z 1]) (+ x y z))
> (my-add 100) ;; x = 100, y = 10, z = 1
111
> (my-add 10 20) ;; x = 10, y = 20, z = 1
31
> (my-add 50 #:z 5) ;; x = 50, y = 10, z = 5
65
```

### 3.1.3 Using Rest Argument

Functions can also be defined to take any number of arguments with a *rest argument*, which means "put the rest of the arguments here". The rest argument always ends up holding a list. A rest argument can be combined with other arguments types, but must appear last and cannot have a default value.

```
> (define (my-add2 first-arg . rest-arg) (apply + first-arg rest-arg))
> (my-add2 1 2 3) ;; first-arg = 1, rest-arg = '(2 3)
6
```

## 3.2 Return Value

Every function has a return value, which is the last value appeared in the function. There is no explicit `return` statement in Racket. If the function does not have an explicit last value, it returns `#<void>`.

Occasionally, if you want to return multiple values, you can use `(values val1 val2)`. But consider first whether returning a `pair`, `list`, or customized `struct` is more convenient.

## 3.3 Naming Convention

- Predicate: end with `?`
- Mutating an existing object: end with `!`
- Type conversion: `type1->type2`
- If `func` handles one input or return one output, its variant which handles multiple inputs or returns multiple outputs should use the same name but suffixed with `*`.
- The `*` suffix can also denote a variant that treats multiple arguments as nested rather than parallel: `for` vs. `for*`, `let` vs. `let*`.

## 3.4 A List of Common Functions Dealing with Strings

### 3.4.1 Type Conversion

- List vs. String: `list->string` and `string->list`.
- Character vs. Integer (ASCII code): `integer->char` and `char->integer`.

### 3.4.2 String Formatting

- The function `format` is like `printf`, but instead of creating an output to STDOUT, it produces a formatted string as return value.

### 3.4.3 Other Utility Functions in `racket/base`

- `(string-length str)`: return `len(str)`.
- `(string-ref str i)`: return `str[i]`.
- `(substring str start end)`: return `str[start:end]`.
- `(string-append str1 str2)`: return `str1 + str2`.
- String comparison functions, e.g. `(string=? str1 str2)`.
- String predictates: `(string-contains? str substr)`, `(string-prefix? str prefix)`, `(string-suffix? str suffix)`.

### 3.4.4 Other Utility Functions in `racket/string`

- `(string-join strs [sep])`: return `sep.join(strs)`.
- `(string-replace str old new)`: return `str.replace(old, new)`.
- `(string-split str [sep])`: return `str.split(sep)`.
- `(string-trim str [sep])`: return `str.strip(sep)`.

### 3.4.5 String I/O

Print to STDOUT:

- `display`: print in a human-readable form.
- `write`: print in a machine-readable form (raw text).
- `printf`: C-style print function (Reference).

Taking inputs from STDIN:

- `read-char`: read a single character.

- read: `read an S-expression.`
- read-line: `read a single line and produce a string.`

**Remark** You can think of `write` as writing values in a format which `read` can read back in.

Print to STDERR:

- Writing to STDERR is accomplished in Racket by using the more general file output function, `fprintf`, which takes a file descriptor (or *port*, as they are called in Racket) as an additional argument.
- The port associated with STDERR can be obtained using the function `current-error-port` and can be passed into `fprintf`:

```
(fprintf (current-error-port) "Oops you launched a spaceship.\n")
```

## 3.5 A List of Common Functions Dealing with Binaries

### 3.5.1 Bitwise Operations

- To manipulate such values on a bit level, Racket provides
  - `bitwise-ior`: bitwise inclusive-OR
  - `bitwise-xor`: bitwise exclusive-OR
  - `bitwise-and`: bitwise AND
  - `bitwise-not`: bitwise NOT

### 3.5.2 Utility Functions

- The `printf` and `format` functions allows you to print integers in binary, octal, and hexadecimal forms. To convert a *positive* integer to its binary (String) form, try `(number->string n 2)`.
- The `(arithmetic-shiftn n m)` function returns the bitwise shift of `n` in its (semi-infinite) two's complement representation. If `m` is non-negative, the integer `n` is shifted left by `m` bits, i.e., `m` new zeros are introduced as rightmost digits; if `m` is negative, `n` is shifted right by -`m` bits, i.e., the rightmost `m` digits are dropped.
- The `(integer-length n)` function returns the number of bits in the (semi-infinite) two's complement representation of `n` after removing *all leading zeros* (for non-negative `n`) or *ones* (for negative `n`).

```
> (integer-length 8)
4 ;; 8 bits => 00001000 but leading zeros are trimmed
> (integer-length -8)
3 ;; 8 bits => 11111000 but leading ones are trimmed
```

### 3.5.3 Processing Bytes and Words

A *byte string* is a sequence of bytes, each being a value between 0 and 255 inclusive. You can specify a byte-string constant like a string constant with an additional `#` just before the first double quote.

There are a number fo byte-string functions that parallel the string functions:

- Type conversion: `bytes->list` and `list->bytes`, etc.
- I/O: `read-byte`, `read-bytes-line`, etc.
- Utility: `bytes-length`, `bytes-ref`, etc.

## 3.6 A List of Common Higher-Order Functions

### 3.6.1 Mapping

- `(map proc lst ...)`: Apply `proc` to every element in `lst`.
- `(for-each proc lst ...)`: Similar to `map` but `proc` is called for its side effect and the return value is ignored.

### 3.6.2 Filtering

- `(filter pred lst)`: Return `[x for x in lst if pred(x)]`
- `(remove val lst)`: Return a list with the first occurrence `val` removed.
- `(remove* vlst lst)`: Return a list with every occurrence of every element in `vlst` removed.

### 3.6.3 Folding

- `(foldl proc init lst ...)`: If `foldl` is called with `n` lists, then `proc` must take `n+1` arguments. The extra argument is the accumulative return value so far. The `proc` is internally invoked with the first item of each list, and the *final* argument is `init`. In subsequent invocation of `proc`, the last argument is the return value from the previous invocation of `proc`. Unlike `foldr`, `foldl` processes the `lsts` in constant space.

- `(foldr proc init lst ...)`: Like `foldl`, but the lists are traversed from right to left. Unlike `foldl`, `foldr` processes the `lsts` in space proportional to the length of `lsts`.

# 4 Data Structures

## 4.1 Struct

Racket official reference for struct

### 4.1.1 Struct Definition

```
(struct id (param1 param2) #:transparent)
```

- The `#:transparent` flag makes the fields of the structure visible in all circumstances.

### 4.1.2 Struct Built-in Functions

- The above expression, whene valuated, provides the following functions:
  - `(id p1 p2)`: a constructor (which takes two arguments and return an `id` instance).
  - `(id? obj)`: a predicate testing if the argument is an `id` instance.
  - `(id-param1 id-instance)`, `(id-param2 id-instance)`: getters for fields.
- Structures are immutable by default; adding `#:mutable` to the definition would further provides you field setters (mutators), i.e., `set-id-param1!` and `set-id-param2!`.

### 4.1.3 Advanced Flags

- `#:constructor-name constructor-id`: a *constructor* procedure that takes appropriate number of arguments and returns a new instance of the structure type, where `m` is the number of fields that do not include an `#:auto` option, defaults to `id`.

```
> (struct posn (x y) #:constructor-name def-posn #:transparent)
> (def-posn 1 2) ;; creates a posn instance with x=1 and y=2
(posn 1 2)
```

- `#:auto` and `#:auto-value`: If this flag is supplied as a *field-option*, then the constructor for the structure type does not accept an argument corresponding to the field. Instead, the structure type's automatic value is used for the kfield, as specified by teh `#:auto-value` option, or as defaults to `#f` when `#:auto-value` is not supplied. The field is mutable if `#:mutable` flag is supplied. If a *field* includes the `#:auto` option, then all fields after it must also include `#:auto`, otherwise a syntax error is reported.

```
> (struct posn (x y [z #:auto #:mutable])
          #:auto-value: 0
          #:transparent)
> (posn 1 2) ;; ctor does not accept argument for z
(posn 1 2 0) ;; z takes auto-value as its value
```

## 4.2 Vectors

### 4.2.1 Functions provided in `racket/base`

- *Constructor:* `(vector elem1 elem2 ...)`
- *Getter:* `(vector-ref vec i)`
- *Setter:* `(vector-set! vec i val)`
- *Length:* `(vector-length vec)`
- *Conversion*: `vector->list` and `list->vector`
- `(build-vector n proc)` creates a vector of `n` elements by applying `proc` to the integers from `0` to `(sub1 n)` in order.
    - If `vec` is the resulting vector, then `(vector-ref vec i) <= (proc i)`.

### 4.2.2 Functions provided in `racket/vector`

- `(vector-map proc vec ...)`: Return a fresh vector of `(map proc (vec ...))`.
    - The number of `vec`s supplied must equal to `proc`'s arity.
    - All `vec`s must have the same length.
- `(vector-map! proc vec ...)`: Like `vector-map`, the result is written to the first `vec`.
- `(vector-append vec ...)`: Return a fresh vector of vector concatenation.
- `(vector-take vec pos)`: Return a fresh vector containing `vec[0:pos]`.
- `(vector-take-right vec pos)`: Return a fresh vector containing `vec[-pos:]`.
- `(vector-drop vec pos)`: Return a fresh vector containing `vec[pos:]`.
- `(vector-drop-right vec pos)`: Return a fresh vector containing `vec[:-pos]`.
- `(vector-split-at vec pos)`: Return two vectors `vec[:pos]` and `vec[pos:]`, efficient.
- `(vector-split-at-right vec pos)`: Return `vec[:-pos]` and `vec[-pos:]`, efficient.
- `(vector-filter pred vec)`: Return `[x for x in vec if pred(x)]`.
- `(vector-count pred vec)`: Return `len([x for x in vec if pred(x)])`.

- `(vector-member val vec)`: Return `vec.index(val)`.
- `(vector-sort vec less-than?)`: Return a fresh, sorted vector.

## 4.3 Hashtables

### 4.3.1 Functions provided in `racket/base`

- *Constructor:* `make-hash` creates a new hashtable which values are to be compared with `equal?`, as is necessary for strings; `make-hasheq` creates a table for which `eq?` (pointer-wise equality) is used.
- *Getter:* `(hash-ref table key failure)`.
- *Setter:* `(hash-set! table key val)`: Insert `(key, val)` pair (or update if `key` exists).
- *Delete:* `(hash-remove table key)`.
- *Size:* `(hash-count table)`
- *Iterate:* `hash-map` or `(hash-for-each table proc)`.
- *Membership*: `(hash-has-key? table key)`
- *Return keys or values:* `(hash-keys table)` and `(hash-values table)`.
- *Conversion*: `hash->list` and `list->vector`

## 4.4 Association Lists

An *association list* is a list whose elements are all key-value *pairs*. As lists, association lists work with the usual list functions. The most important function for associated list is `(assoc v lst [is-equal?])`, which locates the first element of `lst` whose `car` is equal to `v` according to `is-equal?`. If such an element exists, the key-value pair is returned. Otherwise, the result is `#f`.

# 5 Advanced Topics

## 5.1 Loops

### 5.1.1 Basics

The basic `for` loops have two mandatory ingredients

1. An *iterator binding*, consisting of an *identifier* and a sequence of *values*,
2. A *body* of one or mroe expressions.

On each pass of the loop, the next value from the sequence is assigned to the identifier, and the body of the loop is evaluated. The loop exists when the iterator is exhausted.

### 5.1.2 Parallel vs. Nested

A loop can have more than one iterator binding. In that case, the `for` loop progresses in parallel whereas the `for*` loop progresses in a nested manner:

```
> (for ([i '(1 2 3)]
        [j '(4 5)])
       (println (list i j)))
'(1 4)
'(2 5)
```

```
> (for* ([i '(1 2 3)]
         [j '(4 5)])
        (println (list i j)))
'(1 4)
'(1 5)
'(2 4)
'(2 5)
'(3 4)
'(3 5)
```

### 5.1.3 Loops on Structures

A `for` loop discards the results of evaluating the body, which makes it mostly used for side-effects. Its return value is `#<void>`. By contrast, the `for/list` and `for/vector` variants gather these intermediate values into a new list or vector:

```
> (define xs
    (for/list ([i (list 1 2 3)])
              (* i i)))
'(1 4 9) ;; A list!
> (define xs
    (for/vector ([i (list 1 2 3)])
                (* i i)))
'#(1 4 9) ;; A vector!
```

`for/hash` follows the same idea, but expects the body to return two values on each pass, representing a key-value pair:

```
> (define h
    (for/hash ([i (list 1 2 3)])
              (values i (* i i))))
'#hash((1 . 1) (2 . 4) (3 . 9))
```

As a remark, though sequences can be used directly in an iterator binding, the best performance comes from using sequence constructors like `in-range`, `in-list`, and `in-string`.

### 5.1.4 Guards & Breaks

A *guard* expression limits the evaluation of the body based on a condition.

- `#:when <condition>`: execute the loop body when condition is satisfied.
- `#:when <condition>`: execute the loop body when condition is NOT satisfied.

```
> (for/list ([i (in-range 3)]
             #:when (odd? i))
            i)
'(1)
> (for/list ([i (in-range 3)]
             #:unless (odd? i))
            i)
'(0 2)
```

A *break* expression stops the loop early based on a condition.

- `#:break <condition>`: exits the loop immediately when the condition is satisfied.

- `#:final <condition>`: when condition is satisfied, execute loop body one last time before exiting the loop.

```
> (for/list ([i (in-naturals)]
             #:break (= i 3))
            i)
'(0 1 2)
> (for/list ([i (in-naturals)]
             #:final (= i 3))
            i)
'(0 1 2 3)
```

### 5.1.5 Loops vs. Maps

The following are equivalent:

```
(map
 (lambda (i) (* i i))
 (filter odd? (range 100))
```

```
(for/list
 ([i (in-range 100)]
  #:when (odd? i))
 (* i i))
```

## 5.2 Recursion

Every recursive function has two essential ingredients:

1. The task to be done is broken down into smaller versions of the same task.
2. The function reaches a terminating function eventually.

Recursion is a favored technique in functional programming because it helps avoid mutation of values.

### 5.2.1 Tail Recursion

*Tail recursion* refers to the special case where the return value of the recursion branch is nothing more than the value of the next recursive call (also known as a *tail call*).

Here is a non-tail-recursive factorial function:

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

Here is a tail-recursive version:

```
(define (tail-factorial n [acc 1])
  (if (= n 1)
      acc
      (tail-factorial (- n 1) (* n acc))))
```

### 5.2.2 Tail-Call Optimization

Some smart compilers can recognize tail recursive calls and optimize them. Ordinarily, each call to a function, including a recursive call, causes another set of arguments to be saved in the *call stack*. When compiler optimizes tail-calls, the same frame will be reused, i.e., the current arguments on the call stack are replaced by the arguments of the tail call. Thus, the stack never grows and no memory is wasted.

## 5.3 Exception Handling

### 5.3.1 Error Message Conventions

```
<srcloc>: <name>: <message>;
    <continued-message> ...
        <field>: <detail>
    ...
```

- `<srcloc>`: Source location.
- `<name>`: Identify the function/object throwing the exception.
- `<message>`: A concise error message.
- `<continued-message>`: More descriptive error message.
- `<field>: <detail>`: Document the specific value that triggered the error.

### 5.3.2 Throwing Exceptions

To raise an exception `v`, use `(raise v)`. The exception argument `v` can be anything and will be passed to the current exception handler. For a certain type of exception, we have:

- raise-user-error
- raise-argument-error
- raise-result-error
- raise-range-error
- raise-type-error
- raise-arity-error
- ...

### 5.3.3 Catching Exceptions

We wrap with-handlers around an expression to catch exceptions that might arise from the expression. The syntax is similar to cond.

```
(with-handlers
 ([<exception-type-1> <proc-handling-exception>]
  [<exception-type-2> <proc-handling-exception>]
  [<exception-type-3> <proc-handling-exception>])
 <function-body>)

(with-handlers
 ([exn:fail:contract? (lambda (exn) 'contract-error-occurs)])
 (car 42))
```

### 5.3.4 User-Defined Exceptions

We create a new exception type with struct that inherits from an exisiting exception type (either the root exn type or a subtype of it):

```
(struct exn:no-vowels exn:fail ()) ;; subtype of exn:fail
(define (raise-no-vowels-error word)
  (raise (exn:no-vowels
          (format "word ~v has no vowels" word) ;; ~v: prints next argument
          (current-continuation-marks))))
(define (f word)
  (unless (regexp-matc #rx"[aeiou]" word)
          (raise-no-vowels-error word))
  (displayln word))

> (f "strtd")
```

```
word "strtd" has no vowels

> (with-handlers
   ([exn:no-vowels? (lambda (exn) 'no-vowels)])
   (f "strtd"))
'no-vowels
```

## 5.4 Unit Testing

Racket includes a library called `rackunit` that allows us to create unit tests called *checks*. Each check is an assertion about the behavior of the program, usually consisting of a function call and the expected result. If the check is valid, it passes silented. If the check is invalid, an error is raised, along with the expected and actual values.

The special `(check-= <actual> <expected> <tolerance>)` can test whether a numerical result falls within a certain tolerance of a precise result, which is useful for inexact or probabilistic checks.

The function `(check-exn <exn-type> <actual>)` can test whether `<actual>` leads to a exception of type `exn-type`.

## 5.5 Local Variables

### 5.5.1 Implicit Begins

A `begin` expression sequences expressions: `(begin expr ...)`. The `expr`s are evaluated in order and the result of the last `expr` is ignored. Many forms, such as `lambda` or `cond`, support a sequence of expressions even without a `begin`, because Racket wraps an *implicit* `begin` around the body.

### 5.5.2 Local and Let

```
(let ([id val-expr] ...) body ...)
(let* ([id val-expr] ...) body ...)
(local [definition ...] body ...)
```

The `let` expression evaluates the `val-expr`s from left to right, creates a new local variable for each `id` taking the value of the result. It then evaluate the `body`s, in which the `id`s are bound. In this case, `id`s must be distinct.

The `let*` expression behaves the same as `let`, except the `id`s need not to be distinct; later bindings have access to and may shadow earlier bindings. This is because `let*` is implemented using nested `let`s.

The `local` expression accepts bindings expressed in the same way as in the top-level, i.e., using `define`, `struct`, etc.

There is a variable on `let` known as *named* `let` which facilitates the writing of loops:

```
(define (filter pred lst)
  (let myloop {[l lst][acc empty]}
    (cond
     [(empty? l)(reverse acc)]
     [(pred (first l))(myloop (rest l)(cons (first l) acc))]
     [else (myloop (rest l) acc)])))
```

The named `let` in this case essentially defines the local helper function `myloop`; the names in the list of bindings in the `let` become the parameters, and the values are the initial arguments.

## 5.6 Regular Expressions

### 5.6.1 Overview

- A string or byte string can be used directly as a regexp pattern, or it can be prefixed with `#rx` to form a *reading* regexp value or `#px` to form a *printing* regexp value.
- The `regexp-quote` function takes an arbitrary string and returns a string for a pattern that match exactly the original string. In particular, metacharacters are escaped with two backslashes. You can use this function as a sanity check.

### 5.6.2 Matching Functions

- `(regexp-match regexp-pattern text-string)`: Returns the matched substring if the pattern is found, or `#f` otherwise.
- `regexp-match?`: Same as above, but returns `#t` or `#f`.
- `regexp-match-positions`: Same as above, but returns a pair `(start, end)` denoting index of matched substring, or `#f` otherwise.
- `regexp-split`: Same as above, but performs a split based on the pattern and returns the splitted list.
- `(regexp-replace regexp-pattern text-string new-string)`: Replaces the first matched substring in `text-string` with `new-sring`.

- `(regexp-replace* regexp-pattern text-string new-string)`: Replaces every matched substring in `text-string` with `new-sring`.

### 5.6.3 Syntax

- `^` and `$` identifies the beginning and the end of the text string.
- `.` matches any character.
- `*` = zero or more; `+` = one or more; `?` = zero or one.
- `[abc]` matches one character from the characters enclosed within the brackets.
- `[^abc]` matches one character from the characters not enclosed within the brackets.
- For more advanced syntax, e.g. clustering, alternation, backtracking, try here.

### 5.6.4 `#px`-only Syntax

In general, `\x` and `\X` are the opposite of each other.

- `\b` matches a word-boundary and `\B` matches a non-word boundary.
- `\d` matches a digit and `\D` matches a non-digit.
- `\s` matches an ASCII space and `\S` matches a non-space.
- `\w` matches a character from `[A-Za-z0-9_]` and `\W` matches a non-"word" character.

You can use braces to specify much finer-tuned quantification than `*+?`:

- `{m}` matches exactly `m` instances of the preceding pattern, $m \geq 0$.
- `{m,n}` matches at least `m` and at most `n` instance, $n \geq m \geq 0$.

# 6 References


- Racket official guide
- Beautiful Racket
- UWaterloo CS 241 Racket refresher
- UWashington CSE341