Racket Essentials for CS241

This is a big document! The most important sections are 1, 2.2, 5, and 6. 4 is helpful but you can get by without most of it.

Contents

Overview

This guide summarizes additional Racket features not covered in CS135 / *How to Design Programs* (HtDP) but useful for the assignments in CS241. It is by no means exhaustive and you are not limited to the features introduced here!

1 Setup and Basics

1.1 Picking your Language

CS135's Racket dialects (Beginning Student, Intermediate Student) are a very small subset of the full Racket language. If you're using Dr. Racket, the language you select from the drop-down menu must be "Determine Language from Source". This should put `#lang racket` at the top of your file. If you prefer to use VS Code, vim, emacs, nano, etc. you'll need to make sure this line is here too, otherwise your file will not be runnable.

You might also consider using `#lang racket/base`, which is the same language, but doesn't include any of the standard libraries except the base libraries. (Sorry, in Racket lingo they are "packages" not "libraries"). If you do this you'll need to include the libraries you need yourself. It's perfectly fine to use `#lang racket` though.

Racket also skips the first line if it's a shebang, so your first line can be `#!/usr/bin/env racket` and you can now `chmod +x` the file and run it from the command line.

1.2 False Friends

Most of the code you wrote in CS135 will still work in `#lang racket` with a few exceptions noted below.

1.2.1 Structs

Racket structs are similar to those in the HtDP languages, but there are some minor changes. First, rather than using `define-struct` like in Beginning/Intermediate Student, you use **`struct`**. Second, the default constructor name is `struct-name` not `make-struct-name`.

```
;; Intermediate Student
(define-struct my-struct (x y z))
(define foo (make-my-struct 1 2 3))

;; #lang racket
(struct my-struct (x y z))
(define foo (my-struct 1 2 3))
```

The above snippets are nearly identical, but there's one other difference. In Intermediate Student, when the value of foo is displayed, it will be `(make-my-struct 1 2 3)` but in full Racket it will be `#<my-struct>`

In Racket, structs are *opaque*, which is a bit of terminology you might recognize from CS136. What it means in Racket is that the struct cannot be inspected. Its fields are ONLY accessible through the field selector functions (in this case, `my-struct-x` etc.). This includes the printing functions, so it is not possible for them to display the field values when printing an opaque structure[1].

[1] without learning about adding property functions to structs, which isn't necessary for the course so I will, with a heavy heart, omit it

To make a struct transparent, use the **#:transparent** keyword.

```
(struct my-struct (x y z) #:transparent)
(define foo (my-struct 1 2 3))
```

There! Now we have the equivalent to what we wrote in CS135. If you *really* want the constructor to be called make-<struct-name> you can use another keyword, **#:constructor-name** e.g.

```
(struct my-struct (x y z) #:transparent #:constructor-name make-my-struct)
(define foo (make-my-struct 1 2 3))
```

There are a number of new things that structs can do in full Racket, but most of them will not be relevant in this course. One that you might find useful is inheritance. When defining a struct you can put the name of an existing struct before the list of fields. I'll explain by example

```
> (struct fruit (weight shape colour) #:transparent)
> (struct banana fruit (length) #:transparent)
> (struct apple fruit (shininess) #:transparent)

> (define my-apple (make-apple 100 'round 'red 'very)) ; very shiny apple, 100g
> (fruit? my-apple)
#t
> (apple? my-apple)
#t
> (banana? my-apple)
#f
> (fruit-shape my-apple)
'round
> (apple-shininess my-apply)
'very
```

Note that fields are selected using the selector function of the ancestor struct.

1.2.2 true, false, and (void)

The Boolean constants `true` and `false` are just that, constants. Their values are `#t` and `#f`. So while you can continue to use the full word in your code, when Racket displays it, it will display the value.

Another important thing to note is that in full Racket, every value that's not `#f` is considered to be true. Many languages have something similar. In C and C++, anything that's not 0 (false) is considered to be true. Note that in C the NULL pointer is numerically 0 and considered false, but in Racket there's only one SINGLE false value, and that's `#f`.

Next, there's another special value, `#<void>`. This is a value you cannot just write in your code. The Racket syntax `#<any text>` like we saw with opaque structs is an "unreadable value". That means that Racket cannot convert the text back into the value that made it. If you want a void value you can get it using the `void` function, which takes any number of arguments and returns `#<void>`. This is useful because void is the only value that Racket will not print to the screen when it occurs as the result of a top level expression. That's useful for the assignments as sometimes you'll want to call a function for its side effects, but you don't want it putting its return value into standard out. `(void (my-function ...))` solves that issue.

Void comes up a few places. The first place you're likely to encounter it is **cond**. In Intermediate Student, if a **cond** expression is evaluated and all of the test expressions are false, it throws an

exception, halting the program. In full Racket, the **cond** expression returns void. The next place it comes up is functions that exist only for side effects (such as **printf**, a function we'll see shortly).

1.2.3 Numbers

The Racket number system is complicated, but essentially the same as it was in CS135. The one difference is that the full Racket reader will assume a number written in decimal expansion is an inexact value (a floating point number) where Intermediate Student would convert it to an exact rational. E.g. In CS135 if you wrote 1.5 you get the fraction 3/2. In full Racket you will get the same thing you'd get in C if you wrote `1.5` (i.e. a double). If you wanted it to be exact you'd write `#e1.5` - e for exact. You can similarly write `#i2` if you wanted to save an entire character vs writing 2.0. In this course we will not be using floating point numbers, but you might do so accidentally by writing `2.0` which was an integer in CS135, but a float in full Racket.

1.2.4 Lists and Pairs

Lists work more-or-less the same in full Racket as they did in HtDP, but there's one difference: You're allowed to write `(cons a b)` even if b is not a list. If b is not a list, this is what's called a "pair". You can also create pairs using the quote syntax

```
(define my-pair '(a . b))
```

You should not attempt to access these two values using `first` and `rest`. These functions are only for properly formed lists, which a pair is not. Instead, use the opaquely named `car` and `cdr` functions[2].

```
> (car my-pair)
'a
> (cdr my-pair)
'b
> (first my-pair)
first: contract violation
expected: (and/c list? (not/c empty?))
given: '(a . b)
```

If you see any stray dots in your lists, you likely used `cons` when the second argument wasn't a list. In CS135 you would have immediately got an error, now it's valid (but doesn't make a list).

You will also notice that full Racket does not have the `member?` function. It has `member`. There is no question mark as the `member` function does not return a Boolean value. It returns `#f` if the value isn't found, but if it is, it returns the list with that value as the first element, e.g. `(member 4 '(2 4 1)) => '(4 1)`. Since this isn't `#f` it's considered true, meaning the difference is a minor one.

1.3 Memory Model

In CS135 we learned about the Racket substitution model. In CS136 we learned about the C memory model. But what about the Racket memory model? How were those expressions stored in memory, and what really happens when we look up values? Great questions!

[2]These stand for "contents of address register" and "contents of decrement register", which is a bit of Lisp history we won't get into

Things like numbers, strings, Booleans, structs, these are values. The `define` syntax *binds* a value to an *identifier*. Binding is a bit of Racket jargon, which is why Dr. Racket will say things like "go to binding" and "next bound occurrence". The former means "go to definition" and the latter means "find the next time this variable is used. Not another one with the same name, but this one, right here".

Variables (including function parameters, struct fields, elements of lists) are all *references* to values. When we showed the substitution rule for function call syntax, this was not accurate (it's a simplification). Consider the following:

```
(define my-string "Hello, world!")
(define second-string my-string)
```

The above code has two variables, but only one string value is stored in memory. Both `my-string` and `second-string` reference the same string value. (If it helps to think in C/C++ terms, both are pointer pointing to the same address - in practice they probably ARE pointers, but the Racket standard doesn't say that they must be).

Does this matter? Yes, because of how the different generic quality functions work. It also matters when dealing with mutable objects. The above Racket string is immutable (just like it would be in C when using string literals) but Racket also supports mutable strings, so this distinction can be very important. Racket also supports other mutable data structures, though immutable varieties are preferred.

1.4 Equality

You saw `equal?` in CS135. There are actually two other generic equality functions. Another is `eq?`. The Racket documentation uses some fancy language-theory words here, you can read it if you like. I'll put it a bit more plainly. `equal?` is asking "Are these two values equivalent?" where `eq?` is asking "are these literally the same value?". If you're thinking in terms of C pointers, `(equal? a b)` is like the C `*a == *b` while `(eq? a b)` is like the C `a == b` Observe:

```
(define string1 "CS241")
(define string2 string1)
(define string3 (string-append "CS" "241"))


(equal? string1 string3) ; => #t
(eq? string1 string2)    ; => #t
(eq? string1 string3)    ; => #f
```

Savvy readers may wonder why I added the complexity of string concatenation. This is because, like C, Racket will uniquify string literals! The technical term is "intern". If I write "CS241" twice in my code, there will be only one string created, and both variables will be references to the same object. Strings returned by functions will not be interned, so be careful! This can cause odd behaviour when reading text input. Your code might work in tests because the strings are interned, but when reading from STDIN they won't be! (Sorry, this paragraph has too many exclamation marks). A symbol is basically just a string that is interned. That means that while `string=?` must inspect every character in each string in the worst case, `symbol=?` must only compare the addresses (and full Racket doesn't have it, since it's just the `eq?` function).

Now the number system is a bit wild here. One might observe that `eq?` works with integers and floats and conclude that it works for numbers in general. It does not! For floating point numbers

it does (but you should, in general, never[3] use strict equality for inexact values anyway), and for small integers as well (here small likely means "less than

$$2^{61}$$

"). Why? Don't worry about it[4]. Just remember that if you know both operands are numbers, use `=`. If not, `eqv?` is like `eq?` but it handles numbers as a special case.

Why not always use `equal?` then? It's slow. So slow. You also usually do not want to use any of the generic equality functions directly, but it's important to know for hash maps, which are discussed later.

1.5 I/O

In CS135 we didn't enter any input to our programs, and our output was printed by top-level expressions. That won't be enough for this course, where we need to read input from STDIN, and we need to write output more elegantly than by using top-level expressions. Let's start with output. There are three types of output: `print` `write` and `display`. These also come with "and newline" forms, e.g. `displayln`. Let's see some examples.

```
> (displayln "Hello, world!")
Hello, world!
> (writeln "Hello, world!")
"Hello, world!"
> (println "Hello, world!")
"Hello, world!"
```

So first things first, display doesn't include type information (no quotes around strings) and print and write do. So what's the difference between print and write? It's subtle and we don't need to get into it. You likely will be wanting to use **`printf`** for displaying text anyway. If you know Python you can think of display as using str() and write/print using repr().

`printf` is somewhat similar to other languages' but the place-holders are different. The Help Desk has them all, but you can get away with using ã and ṽ exclusively.

```
> (printf "Welcome to CS~v\n" "241")
Welcome to CS"241"
> (printf "Welcome to CS~a\n" "241")
Welcome to CS241
```

Of course 241 is a number so I should have passed it as an integer instead of a string, but if I did that then both place-holders would do the same thing. The v place-holder means "format including type information like quotation marks" where the a placeholder is equivalent to display "just the human readable contents please". (I like to remember them as standing for (v)alue and (a)nything but I don't actually know why the letters were picked).

Whew! For input you might try out `read`. You probably don't want to use `read`. Read reads the next datum, and already we're lost. What's that? Don't worry about it! (In fact what that is is sort of one of the topics of the course!) It reads one "thing" from input and is how Racket reads its own input files. That's not the kind of I/O we'll be doing in this course. We typically want to read an entire line of text at a time. That's **`read-line`**'s job

[3] I am aware of, and amused by, the contradiction of using "never" in a self-described generalization

[4] It bothers me to say this, but this document cannot turn into a 200 page textbook

```
(define the-line (read-line))
(printf "You entered the text ~v!\n" the-line)
```

I've used v here to put quotes around it. If we run that in Dr. Racket it'll show a fancy input box. From the command line it just sits there, waiting. Once I hit enter, things happen. What if I hit Ctrl+D for EOF?

```
You entered the text #<eof>!
```

Huh, another one of those unreadable values like void. You check if a value is EOF or not using `eof-object?` Why not just `eof?`? *Shrug.*

For `write` `display` `print` (and their "ln" equivalents) there's a second optional parameter specifying which output port (output stream) to write to, the default is STDOUT. `read` and **`read-line`** have one too, the port to read from (default is STDIN). If you want to write to STDERR, its name is the verbose `(current-error-port)`. This function[5], when called with no arguments, returns the current error port. If run from the command line, that will be STDERR. There's also `current-input-port` and `current-output-port` (which from the command line will be STDIN and STDOUT).

1.5.1 Byte-Level Output

If you want exact control over what bytes are output, you should use `(write-byte b)` which will write a single byte (an integer from 0 to 255 inclusive) to STDOUT.

```
> (write-byte 97)
a
```

Of course 97 is an ugly magic number, but in Racket the character c isn't an integer like it is in C and C++. It's a character, its own type. It's written `#\a`. If you want its value: `(char->integer #a) => 97`. This is giving you the Unicode Code Point for the given character. If it's an ASCII character this is the same as its ASCII value, so the difference shouldn't matter in this course.

2 Core Data Types

In CS135 we learned most of the core data types that Racket has at its disposal: exact numbers (integers and rationals), inexact numbers (floating point numbers), strings, Booleans, lists, and structs. Besides the differences already noted, these types are the same as they were in Intermediate Student. Now we get to the good stuff, all the core datatypes that we didn't use in CS135 (some of which we could have done, and some of which are not available in the HtDP languages).

Oh, one thing to note if you ignored the "this is like Racket" asides in CS136: Racket lists are linked lists. `cons` is sort of like a struct with two fields, first and rest. And empty is just a null pointer. (In fact, empty is the HtDP friendly name for the Racket value `null`). This isn't new, just something we didn't mention in CS135 since we hadn't seen linked lists before.

2.1 Vectors

A Racket vector is an array. Like in C, they have a fixed length once defined. In Racket lists are usually preferred, but as we all know, arrays have constant-time random access, where linked lists

[5] parameter, but who's counting?

do not. So if you have a lot of data and plan to jump around in it rather than doing start-to-end iteration, a vector is preferred.

To define a vector:

```
(define v1 (vector 1 2 3 4 5)) ; vector constructor function, equivalent to list
(define v2 '#(1 2 3 4 5)) ; quoted vector format
```

To access a value:

```
(vector-ref v 2) ; => 3 -- note that it's 0 indexed like in C
```

Many (perhaps even most) list functions have a vector equivalent. There is build-vector, equivalent to build-list, for example.

Also, Racket vectors are mutable. In addition to **vector-ref** there is also **vector-set!** that allows you to change the value at a given index. This cannot be used to grow the vector though. Now we see why it matters that Racket variables are references.

```
> (define my-vector (vector 1 2 3 4))
> (define (start-with-zero v)
    (vector-set! v 0 0)) ; vector-set! returns #<void> so so does my function
> (start-with-zero my-vector)
> my-vector
'#(0 2 3 4)
```

When `start-with-zero` is called, the parameter v will not by a copy of my-vector, but an *alias*. So, just like a C function with a pointer parameter, or a C++ function with a `var` parameter, my function can alter the object[6]

2.2 Hash Maps

Racket also has hash maps to give (amortized) constant-time read and write access for key-value pairs.

There are three flavours, based on which equality function they use. `hash` uses `equal?`, `hasheq` uses `eq?` and `hasheqv` uses `eqv?`. If your key is a symbol you should use `hasheq`, if your key is a number, you should use `hasheqv`, otherwise you should probably use `hash`. `hasheq` will not work with strings, but likely will work with numbers. Nevertheless, `hasheqv` is the correct hash map to use if your keys are numbers!

To create a hash map, use `hash`. This requires an even number of arguments. The even arguments are keys, and the odd arguments the corresponding values.

```
(define hash1 (hasheq 'a 1
                      'b 2))
(define hash2 (hasheqv 'a 1
                       'b 2))
(define hash3 (hash 'a 1
                    'b 2))
```

[6]CS136 for the longest time had a slide saying that this means Racket is pass-by-reference and I never managed to convince the course coordinator to delete it as inaccurate! It's pass by value, but all the values are references. This is how Python works too. Python calls it "pass by assignment" where the Racket documentation doesn't call it anything.

As my keys are symbols, hash1 is the hash with the most appropriate equality function.

To access values, use `hash-ref`. This does not come in multiple flavours, it will use whichever equality function the hash map uses.

```
  > (hash-ref hash1 'a)
  1
  > (hash-ref hash2 'a)
  1
  > (hash-ref hash3 'a)
  1
  > (hash-ref hash1 'c)
hash-ref: no value found for key
  key: 'c
```

As you can see, if you try to get the value for a key that's not present, you get an exception. To avoid this, you can use `hash-has-key?` to first check whether the key is present, or you can use the second optional parameter to `hash-ref`

```
  > (hash-ref hash1 'c #f)
  #f
```

This argument can also be a function, in which case it is called (with no arguments) when the key is not found (and only then). The value this function call returns will be returned by `hash-ref`.

To update a hash map, use `hash-set`. This is *functional*, meaning it returns a new hash map rather than modifying the existing one.

```
  > (define new-ht (hash-set hash1 'c 3))
  > (hash-ref new-ht 'c)
  3
  > (hash-ref hash1 'c #f)
  #f
```

Another useful hash map function is `hash-update`, a higher-order function that uses an updater function to modify a given value.

```
  > (hash-update hash2 'a add1)
  '#hasheqv((a . 2) (b . 2))
```

As with `hash-ref`, `hash-update` has an optional parameter used when the key is missing. This value is fed to the updater when the key is not already in the table.

```
  > (hash-update hash2 'c add1 0)
  '#hasheqv((a . 2) (b . 2) (c . 1))
```

2.2.1 Mutable Hash Maps

Believe it or not, the above immutable hash maps still have O(1) amortized access, including updates like key insertion or removal. (It's magic[7]). They are not quite as efficient as an imperative (or "mutable") hash, but I suggest avoiding mutable data structures whenever possible as the performance penalty is rarely noticeable. If you want to use one anyway then use `make-hash` (also available in hasheq and hasheqv flavours). These constructors take one parameter, a list of key-value pairs (pair in the `(cons k v)` sense). Mutable tables have their own modification functions, ending with an exclamation mark (pronounced "bang" e.g. `hash-update!` is pronounced "hash update bang", if you care).

[7] A hash trie, not a hash table

```
> (define mutable-ht (make-hasheq '((a . 1) (b . 2))))
> (hash-set! mutable-ht 'c 3)
> (hash-ref mutable-ht 'c #f)
3
```

`hash-ref` also has a bang form `hash-ref!` for which the "if not present" parameter is mandatory. If the key is missing, it will insert it with the default value, as well as returning the default value.

3 I/O

4 New Syntax

There is a lot of new syntax supported in full Racket. Let's start with "how did `hash-ref` do that?".

4.1 Optional and Keyword Arguments

Racket does not support function overloading like C++ does, but does support optional parameters. To make a parameter optional, you provide a default value by putting brackets around the name of the parameter and an expression for the default. Optional parameters must all be at the end of the parameter list.

```
> (define (my-function mandatory [optional #f])
    (cond [optional (+ mandatory optional)]
          [else (* 2 mandatory)]))
> (my-function 3)
6
> (my-function 3 2)
5
```

Racket also supports keyword parameters. These are like optional parameters, but their arguments must be prefixed with a keyword, special syntax that starts with #: (like **`#:transparent`**). Keyword arguments let you have many optional parameters with default values without needing to worry about what order they're in.

```
> (define (my-fancy-function mandatory [optional #f] #:kw1 [kw1 #f] #:kw2 [
  key-word-parameter-2 'none])
...)
> (my-fancy-function 1 1 #:kw1 #t)
> (my-fancy-function 4 #:kw2 'yes #:kw1 #t)
```

Note that the name of the parameter does not need to be the same as its keyword (like I did with kw2) but style-wise it's a good idea to use the same name.

4.2 Local Scope

In CS135 we saw **`local`** to create local definitions. This syntax exists in full Racket as well, but is only used rarely because function definitions (and **`cond`**, and many other syntax forms) have an implicit local scope already. Sorry for the jargon, let's look.

```
;; CS135 style
(define (my-function x y)
  (local [(define sum (+ x y))]
...))
;; Full Racket
(define (my-function x y)
  (define sum (+ x y))
  ...)
```

At this point we could talk about let, let*, letrec, etc.. But we won't. **local** is all you need (if you need anything explicit at all).

4.3 mutation / assignment

Racket supports assigning to variables. Functional programmers call this "mutation" and imperative programmers call this "you have a name for that?". Assignment is done using **set!**.

```
> (define x 3)
> x
3
> (set! x 4)
> x
4
```

Note: Like in C++ you can modify the value of a parameter. As Racket is not pass-by-reference, this has no effect outside the function. But **vector-set!** and `hash-set!` do? Yes. `(set! variable value)` is making the variable point to a different object, where things like **vector-set!** leave the variable alone, and modify the state of the object itself.

```
> (define (foo x)
    (printf "x is ~a\n" x)
    (set! x 3)
    (printf "x is now ~a\n" x))
> (define x 4)
> (foo x)
x is 4
x is now 3
> x
4
```

4.4 begin

Oops, that's something else new. Functions can have more than one expression. The value returned by the function is the last expression's value. In this case, that means printf, which returns nothing, so foo also returns nothing. I would have brought this up sooner, but it doesn't make any sense without side effects like from **printf** and **set!**. Many places besides function bodies allow you to have multiple parts. The "result" portion of **cond** clauses also lets you do this and follows the same "value is the value of the last expression" rule. For syntaxes that do not do this, you can use `(begin expr ...)` which evaluates its subexpressions left to right, and produces the value of the rightmost expression.

There's also the less-often-used `(begin0 expr ...)` which also evaluates its subexpressions left to right, but takes on the value of the first (leftmost) expression instead of the last (rightmost) expression.

4.5 if, when, unless

The cond syntax is pretty handy, but there are a few other conditional syntax forms that can be more concise. The first is **`if`** which has exactly 3 parts. `(if test true-value false-value)` evaluates test. If it is true then it evaluates the true-value expression, otherwise the false-value. Note that I said "syntax" not "function". Like **`cond`** this is not a function! The true-value and false-value expressions are left un-evaluated until the test tells Racket which expression to use.

Because of the lack of bracketing the **`if`** syntax is short, but it does mean it can't have local definitions inside (this is one of those cases where **`local`** is still useful, though at that point just use **`cond`** I say).

There is also **`when`** and **`unless`**. `(when test expr...)` evaluates test, and if true, evaluates the remaining expressions left to right. **`unless`** is the same but only evaluates the other expressions when the test is false. This is useful if you have side effects. As with **`begin`** these two take on the value of their last expression (if evaluated). If not evaluated, the **`when`** / **`unless`** expression has the special `#<void>` value.

4.6 Exceptions

Those runtime errors you (probably) got in CS135 were all exceptions. Isn't that neat? No? Okay... Well, we can catch them using something called **`with-handlers`**. It's sort of like try...catch but you put the catch at the top.

```
> (with-handlers ([exn:fail:contract?
                   (lambda (exn)
                     (displayln "You broke the contract my friend!"))]
                  [exn:fail? (lambda (exn) (displayln "Fallback case..."))])
   (/ 5 0))
You broke the contract my friend!
```

Explanation: in the first block you put cases, sort of like **`cond`** cases. The first part is a function that checks whether an exception is one we're interested in. Above my first handler is interested in contract violations (like divide by zero). These are all structs and they use inheritance. All exceptions descend from the base `exn`. "Failure" exceptions all descend from `exn:fail`. I've added a second handler that catches all failure exceptions as well.

If you want to throw your own exceptions you can using **`raise`** but it's a bit tricky to build one. There are some helpful utility functions. One you might find useful is `(error label message)` which will raise a base `exn:fail` exception. The error text is "label: message". You can also provide additional values in which case the message is assumed to be a format string with the same format as **`printf`**. A similar function more useful for this course is `raise-user-error` which has the same parameters. The difference is this will make a `exn:fail:user`. This exception is similar to the base `exn:fail` but when printed it only prints its error message and not a stack trace. This is handy because on the assignments we will sometimes want to print an error and abort the program. If you just use **`error`** you get a stack trace, but this is the internals of your compiler and the user doesn't need to know that. If anything it would greatly confuse them!

4.7 for loops

Is "CS students don't learn for loops until 1B" still a meme? Racket actually does have for loops.

```
> (for ([i '(1 2 3 4 5)])
    (displayln i))
1
2
3
4
5
```

It's more efficient if you use a sequence instead of a list. Most collections have a sequence constructor of the form `in-<collection type>` e.g. `in-list`

```
> (for ([i (in-list '(1 2 3 4 5))])
    (displayln i))
1
2
3
4
5
```

You can provide multiple clauses in your for loop, in which case they are processed in lockstep. The loop stops if any of the clauses has exhausted its sequence.

```
> (for ([value (in-list '(a b c d))]
        [index (in-naturals)])
    (println "~a -> ~v\n" index value)
0 -> 'a
1 -> 'b
2 -> 'c
3 -> 'd
```

Note: in-naturals never stops producing values! It's very easy to make an infinite loop. It should only be used along side finite sequences.

For loops do not have a value (or rather, their value is `#<void>`). But there are variants that do. For example `for/list` returns a list containing the values returned by its body, `for/hasheq` returns an immutable `hasheq` value. To use this, the body of the for loop must be an expression of the form `(values key-expr value-expr)`.

For loops also have some keywords `#:when` and `#:unless` that let you only process certain iterations. This isn't that useful for regular `for` but it is very handy for things like `for/list` where it lets you filter elements out as you go. There are also `#:break` and `#:final` clauses that let you end early. `#:break expr` means "If expr is true, stop the loop now." where `#:final expr` means "If expr is true, then stop after doing the next iteration."

For loops also have a `*` variant like `for*` which nests the clauses instead of doing them in lockstep. E.g. if you have a list of lists:

```
(for* ([sublist (in-list my-list-of-lists)]
       [value (in-list sublist)]) ...)
```

`for/fold` is another fun for loop syntax, but I'll leave that to the Racket help desk to explain.

4.8 named `let` loops

A more general form of loop is something called a "let loop". The syntax looks like

```
(let function-name ([param1 arg1]
                    [param2 arg2] ...)
  body ...)
```

This is equivalent to

```
(local ([(define (function-name param1 param2 ...) body ...)])
  (function-name arg1 arg2 ...))
```

The former is more concise and expressive. If you don't agree then you're free not to use it! HtDP renamed this syntax `recr` which, admittedly, is a way better name than `let` for what it's doing. Much in the way that `first` is a way better name than `car`.

4.9 Quasiquote

Have you ever felt that quoted lists are great, but it's too bad you cannot randomly unquote to put a variable's value in? Good news! Quasiquoting does exactly this. To start a quasiquote, use a backtick instead of an apostrophe. It's the one the tilde (~) is above. When you are quasiquoting, you can put an unquote (a comma) in front of something to evaluate it. E.g.

```
> (define x 3)
> `(1 2 x)
'(1 2 x)
> `(1 2 ,x)
'(1 2 3)
```

The unquoted item can be a more complex expression.

There's also a "splicing" unquote, `,@`. What it does is evaluated the splice-unquoted expression (which must produce a list). Then that list is spliced into the quasi-quoted list. Hmmm, not too happy with that sentence, let's see an example.

```
> (define my-list '(1 2 3))
> `(a b ,my-list c d)
'(a b (1 2 3) c d)
> `(a b ,@my-list c d)
'(a b 1 2 3 c d)
```

5 Pattern Matching

This is technically more syntax but SO useful I felt it needed its own section.

Match syntax looks like the following

```
(match <expr>
  [pattern1 expr1 ...]
  [pattern2 expr2 ...]
  ...)
```

A simple example:

```
(match foo
  [(? string?) (displayln "foo is a string")]
  [(? symbol?) (displayln "foo is a symbol")]
  [_ (displayln "foo is something else")])
```

So is this just **cond** that's a bit more concise? Far from it! Let's see some other things.

```
(match foo
  [(list a b c) (printf "Foo is a list of 3 values: ~a, ~a, and ~a\n" a b c)]
  [(list _ _ _ r ...) (printf "Foo had more than 3 things, the extras are ~a\n" r)])
```

In the first pattern, the list has 3 elements and we bind each of them to a unique name. In the second pattern, the list has 4 or more things. The first 3 are the pattern _ which means "match anything, don't give it a name" and the last one, named r, has an ellipses after which means match 0 or more of them. If not for the first pattern the message would be wrong, this pattern matches lists of 3 or more things, not 4 or more.

Remember arithmetic expression trees from CS135?

```
(define (eval-exrp e)
(match e
  [(? number?) e] ; leaf, return value
  [`(+ ,l ,r) (+ (eval-expr l) (eval-expr r))]
  [`(- ,l ,r) (- (eval-expr l) (eval-expr r))]
  [`(* ,l ,r) (* (eval-expr l) (eval-expr r))]
  [`(/ ,l ,r) (/ (eval-expr l) (eval-expr r))]))
```

I've used the quasiquote pattern so I can unquote l and r. I could have used `(list '+ l r)` instead.

The latter half of the course has a lot of tree traversals, and match makes these both concise and expressive.

It's also possible to use a struct as the pattern, and bind names to the values of the fields of the struct. So I could have a pattern like `(apple size _ _ _)` binding size to the size of the apple, and binding nothing to the other fields.

Patterns are recursive, so when matching a list, struct, etc. you can put a sub-pattern for an element of the list, field of a struct, or whatever else.

6 Bit-Level Manipulation

(Note: This might not make sense until the second lecture).

In this course we'll be manipulating individual bits within integer values. Racket is verbose here. Rather than single-character operators, the bitwise functions have the names `bitwise-and` `bitwise-ior`, and `bitwise-xor`.

Racket also has a bit-shift function `arithmetic-shift`. This takes 2 arguments. The value, and how far to shift it left. You can accomplish right shifts by using a negative shift amount.

```
> (arithmetic-shift 5 2)
20
> (arithmetic-shift 241 -3)
30
```

Explanation: 5 is 101 in binary. Shifting 2 bits left gives 10100, which is 20. 241 is 1111 0111 in binary. Shifting 3 bits right gives 1 1110, which is 30.

If you want hexadecimal literals in your Racket code, they start with `#x` e.g. `(= #xF1 241)` is true. You can also use `#o` for octal (that's an "oh" not a zero) and `#b` for binary.

7 Modules

Each Racket file is a module. By default a module does not export any variables. To export, use the **provide** syntax. To import another module, use **require**

```
;; module.rkt

(provide foo)
(define foo 3)

;; client.rkt
(require "module.rkt")
(displayln foo)
```

In my experience most students put everything in one file, both in their C++ code and Racket code, so maybe you don't need to use this? Up to you. You can also **require** built-in packages, in which case you don't put their name in quotes.

E.g. `(require racket/list)` imports the list helper functions. This is already part of `#lang racket` so it's not necessary, but if you're using `#lang racket/base` then it is.

7.1 Sub Modules

Racket supports submodules. This degree of modularization is overkill for the course, except for two standard submodules, `main` and `test`.

When a Racket module is imported, its sub-modules are not. When a Racket file is run from the command line by `racket file.rkt` then the main module *is* evaluated. The same happens when you hit the "run" button in Dr. Racket.

Dr. Racket also will run the `test` sub-module when you hit run (this can be switched on and off in the Language menu). From the command line to run the tests for a file use `raco test file.rkt`.

```
#lang racket

(define (my-function ...) ...)

(module+ main
  (displayln "Usage information -- ...")
  ...
)

(module+ test
  (unless (equal? (my-function <test case>) expected-value)
    (error 'my-function "Test case 1 failed: Expected ... saw ~v" (my-function ...)))

  ... other tests ...
)
```

The syntax **module+** means "create the named sub-module, and if it already exists, append to it". That's very useful for tests as you can put them beside the function being tested instead of at the end.

8 Testing

Of course that style of testing isn't ideal. Why didn't I use `check-expect`? It's not part of full Racket, it's specific to the HtDP languages. But `rackunit` is a package that has similar functions.

```
#lang racket
(module+ test
  (require rackunit))

(define (function1 ...) ...)
(module+ test
  (check-equal? (function1 arg1) 1)
  (check-within (function1 arg2) 3.1415 1e-3) ; check it matches to 3 decimal places
)

(define (function 2 ...) ...)
(module+ test
  (check-eq? (function2 arg1) 'expected))
```

9 Example Programs

9.1 Reading Input with Recursion

This is an example we use in C for CS136.

```
#lang racket

;;; Simple functions that reads user input and prints it forwards, then backwards.

;; Let's start with basic CS135 style with the addition of I/O

(define (read-and-print-v1)
  (local [(define next-value (read))]
    (cond [(eof-object? next-value) (void)] ; do nothing
          [else
           (displayln next-value)     ;; print before recursion, this is the "forward"
           (read-and-print-v1)
           (displayln next-value)])))  ;; print after recursion, this is the "backward"


;; In full Racket we don't need local

(define (read-and-print-v2)
  (define next-value (read))
  (cond [(eof-object? next-value) (void)]
        [else
         (displayln next-value)
         (read-and-print-v2)
         (displayln next-value)]))

;; We could also use "unless" instead of cond to make it more concise

(define (read-and-print-v3)
  (define next-value (read))
  (unless (eof-object? next-value)
    (displayln next-value)
    (read-and-print-v3)
    (displayln next-value)))
```

9.2 Generating word counts

V1, using recursion

```
#!/usr/bin/env racket
#lang racket

;; (add-words-to-counter list-of-words counter) increments the count of each word
;;   from list-of-words by 1, inserting it with a count of 1 if it's a new word
;; add-words-to-counter: (listof Str) (hashof Str Nat) -> (hashof Str Nat)
(define (add-words-to-counter list-of-words counter)
  (cond [(empty? list-of-words) counter]
        [else (add-words-to-counter
                (rest list-of-words)
                (hash-update
                  counter ; table to update
                  (first list-of-words) ; key
                  add1 ; what to do to the value
                  0))])) ; default value

;; (read-input counts) adds the count for each word read from STDIN to counts
;; read-inputs: (hashof Str Nat) -> (hashof Str Nat)
(define (read-input counts)
  (define next-line (read-line))
  (cond [(eof-object? next-line) counts]
        [else
          (define words (string-split (string-downcase line)))
          (read-input (add-words-to-counter words counts))]))

(read-input (hash))
```

V2, using for loops

```
#!/usr/bin/env racket
#lang racket

(for*/fold ([counts (hash)]) ; accumulator and initial value
           ([line (in-port read-line (current-input-port))]
            [word (in-list (string-split (string-downcase line)))])
           (hash-update counts word add1 0))
```