Using Racket in CS 241

1 Purpose

This is a guide to using Racket 5.3 for assignments in CS 241. We will assume that students know the basics of Racket from CS 135 and CS 136. Useful references include the Guide and Reference in DrRacket's Help Desk, the R5RS (Scheme standard) standard document (in DrRacket's Help Desk or at `http://www.schemers.org`), the SRFI documents describing standard libraries (in DrRacket's Help Desk, or at `http://srfi.schemers.org`) and the book, *The Scheme Programming Language* (TSPL), by Kent Dybvig (available in paperback or at `http://www.tspl.org`).

2 Racket 5.3

Racket can be thought of as R5RS Scheme plus many useful extensions, but there are also some changes, meaning you cannot just paste in Scheme code and expect it to work. Your programs should start with the line

```
#lang racket
```

which wraps your entire file in a module declaration and uses a language with many helpful libraries already preloaded.

To develop your programs in DrRacket, use the Racket language, not the teaching languages used in CS 135. To run your programs on the command line, type:

```
racket myfile.rkt
```

The `racket` binary is included with the DrRacket distribution, but you may need to set up your shell so that its search path includes the directory it is in (or an alias to it). Since Marmoset uses `racket` to test your programs, at least your final round of testing should use this. Given that most programs involve I/O, you may choose to do more testing on the command line.

There are many optional command-line flags for `racket`, including one which gives you a basic read-evaluate-print loop (REPL) after running your program. See the Racket Reference for more details. Any additional command-line arguments are put into a vector (see below) that is the value of the expression (*current-command-line-arguments*).

Note that the values of top-level expressions are printed after evaluation if they are not `<void>`. If you don't want this to happen, you can wrap the expression with (*void* ...).

3 Immutable lists

One major change in Racket, with respect to Scheme, is that lists constructed in the normal fashion are immutable. The R5RS functions *set-car!* and *set-cdr!* are not provided.

Mutable lists can be constructed using the *mcons* and *mlist* functions, as described in section `3.10` of the Racket Reference. An alternative is to use immutable lists that contain *boxes*, mutable cells holding a single value, as described in section 3.12 of the Racket Reference.

4 Text I/O

You will use redirection of standard input and output to read from and write to files in 241, with one exception discussed below.

The basic Racket output function is *display*. This will write a single value to standard output in "human-readable" form. *write* does the same thing in "machine-readable" form; for example, it prints quotes around strings and uses escape sequences. (*newline*) will begin a new line in the output. For finer control, you can use *write-char*, which writes a single character to standard output, or *write-string*.

Racket provides the function *printf*, which works in a fashion similar to the function of the same name in C or C++. *printf* consumes a format string followed by several arguments:

```
>(printf "The values are ~a and ~a\n" (add1 2) "test")
```

```
The values are 3 and test
```

The notation "\n" in a string indicates a new line. Other such escaped characters include "\space", "\return", and "\\" to insert a backslash in a string. The formatting escape "~a" prints in the style of *display*; see the Racket Reference for other options.

There are two basic input functions. *read-char* will read a single character. *read* will read an S-expression, that is, something which resembles one expression in a Racket program, and produce the corresponding list. (You can think of *write* as writing values in a format which *read* can read back in.) *read-line* is an Racket extension which will read a single line and produce a string. There are many other I/O functions detailed in Chapter 12 of the Racket Reference.

The following code will read all input lines into a list of strings.

```
(define (read-all-input)
  (local ((define item (read-line)))
    (if (eof-object? item)
        empty
        (cons item (read-all-input)))))
```

Note the use of *eof-object?*, which tests whether the item read is a special "end of file" object. This is obviously not the best strategy if the input is very large, because it will use too much memory. In this case, you want to use a tail-recursive function to read a line at a time (or several lines, if necessary) and process them, possibly generating output, before repeating the action.

The function *for-each* works like *map*, but it is guaranteed to process the list in left-to-right order, and it throws away the result of applying the function to each element of the list. It is useful for generating output from a list.

It is occasionally useful/required in 241 to write to "standard error", which is a separate output stream that can be separately redirected. This is accomplished in Racket by using the more general file output functions which take a file descriptor (or *port*, as they are known in Racket) as an additional argument. The port associated with standard error can be obtained using the function *current-error-port*, and the function *fprintf* can use this port.

```
(fprintf (current-error-port) "Goodbye, world.\n")
```

5 Working with strings and characters

Converting a string to a list of characters allows one to use Racket's various list-processing functions. The functions *string->list* and *list->string* convert in either direction. The 'a' character is specified by the Racket constant `#\a`. The functions *char->integer* and *integer->char* will convert to and from the numerical ASCII representation (actually, these functions work with the Unicode encoding UTF-8, but this coincides with ASCII in the 0-127 range). The function *format* is like *printf*, but instead of creating output, it produces a string. This is a quick way to create complicated strings containing computed results.

There are a small number of functions which can be applied to strings such as *string-length*, *string-ref*, and *substring*, and there is a larger string library which can be required, as discussed below. Note also that Racket provides string matching using regular expressions, also described below. This is useful for breaking up a string into a list of shorter strings according to various criteria. On a more mundane level, *string->integer* and *integer->string* may be handy.

6 Binary I/O and processing

Much of CS 241 involves representation and manipulation of bit strings.

Racket allows integer values to be specified not only as arbitrary-length strings of decimal digits, as you are familiar with, but in binary, octal, and hexadecimal as well. The Racket values `10`, `#d10`, `#b1010`, `#o12`, and `#xa` are all equal. The *printf* and *format* functions permit one to print integers in binary, octal, and hexadecimal; see the Racket Reference for details.

To manipulate such values on the bit level, there are a few functions described in section 3.2.2.6 of the Racket reference manual. *bitwise-and*, *bitwise-ior* ("inclusive OR"), *bitwise-xor*, and *bitwise-not* should be obvious, except that they can be given arguments of arbitrary length. To figure out what they should do, consider their arguments to be padded on the left. Nonnegative integers are padded with an infinite number of 0's, and negative integers are padded with an infinite number of 1's (as happens with 2's complement numbers). Thus (*bitwise-not* `0`) is `-1`. The function *integer-length* will return the number of bits in a number without such padding, and *arithmetic-shift* will shift such sequences of bits a specified number of places left or right.

CS 241 also deals with bits grouped into bytes and words. Racket provides the data type "byte string" (described in section 3.4 of the Racket Reference) to deal with such groupings effectively. A byte string is a sequence of bytes, each byte being a value between 0 and 255 inclusive. You can specify a byte-string constant like a string constant but putting a # just before the first double quote. If you need to specify an eight-bit value that doesn't correspond to a character on the keyboard or a convenient escape sequence like `"\n"`, you can use a backslash followed by the octal representation; thus the byte strings `#"a"` and `#"\141"` are equal.

There are a number of byte-string functions that parallel the string functions, described in section 3.4.1 of the Racket Reference. For example, there are functions *bytes->list* and *list->bytes*; the lists produced are not lists of characters but lists of numbers between 0 and 255. Sections 12.2 and 12.3 of the Racket Reference describe byte I/O, again parallel to the character-based I/O described above. For example, there are functions *read-bytes-line*, *read-bytes* (with the number of bytes to be read as a parameter), *read-byte*, and the corresponding write procedures.

7 Structures

Reader of the textbook, "How to Design Programs" (HtDP) will be familiar with structures, which are an extension found in full Racket and the teaching languages. Here is the definition of a structure type holding the coordinates of a point:

(**define-struct** *posn* (*x y*) #:transparent)

HtDP readers will not be familiar with the third argument, which makes the fields of the structure visible in all circumstances. (This is best described in the Racket Reference, section 5.)

The above expression, when evaluated, provides the constructor *make-posn* (taking two arguments in this case), the selectors *posn-x* and *posn-y*, and the type predicate *posn?*. Structures are immutable by default; adding the keyword #:mutable also provides the field mutators *set-posn-x!* and *set-posn-y!*. Structures are not part of standard Scheme (R5RS); lists or vectors (next section) are used instead.

Racket provides an alternate way of defining structures which has some advantage. With this definition:

(**struct** *posn* (*x y*) #:transparent)

the constructor is simply called *posn*. This is now the preferred way of defining structures in full Racket.

8 Vectors

Most languages provide arrays, which implement constant-time insertion and lookup into a fixed-sized table. In Racket, these are known as *vectors*. The only advantage of vectors over lists is speed. However, they are less flexible than lists. Imperative languages tend to emphasize arrays for historical reasons, and many "pseudocode" algorithms are described using them. These algorithms can often be implemented using lists or hash tables (described below). There are times when vectors are appropriate, but this has to be determined by a careful examination of their intended use.

A vector can be created with *vector*. The expression (*vector* 1 'blue *true*) returns a vector containing the three given elements. *build-vector* consumes an integer k and an optional initial value, and creates a vector of size k. *vector-length* produces the length of its vector argument.

Vectors are indexed starting with 0. (*vector-ref v i*) produces the ith element of vector v, and (*vector-set! v i val*) sets the ith element of v to val. The functions *vector->list* and *list->vector* do what they suggest.

Here is an implementation of binary search using vectors. Note that we are using the Racket construct (**if** *test true-exp false-exp*), which shortens single-test **cond**s (Racket also provides one-armed **when** and **unless**). You should know that #f and *false* are the only things that **if** and **cond** tests consider to be false, but any other value is considered to be true. This can simplify some tests. We are also using the **local** construct from 135, which provides a convenient alternative to the Racket constructs **let**, **let***, and **letrec**. Below we will discuss all of these constructs.

```
(define (contains? svec key)
  (local (
    (define (bin-search lower upper)
      (if (= lower upper)
        lower
        (local (
          (define (mid (quotient (+ lower upper) 2))))
         (if (> key (vector-ref svec mid))
           (bin-search (add1 mid) upper)
           (bin-search lower mid)))))
    (define answer (bin-search 0 (sub1 (vector-length svec)))))
   (if (= key (vector-ref svec answer))
     answer
     false)))

(define example (vector 1 3 5 6 7 9 10 12))

(= (contains? example 3) 1)
(not (contains? example 8))
(= (contains? example 9) 5)
```

9 Hash Tables

Racket provides both mutable and immutable hash tables, described in section 3.13 of the Racket Reference. These implement what one might call "dictionaries", providing nearly constant-time insertion, lookup, modification, and deletion of (key,value) pairs. The only advantage of a hash table over a list of (key,value) pairs (or an unbalanced binary search tree) is speed. Hash tables retain some of the flexibility of lists, but there is no implicit ordering. Hash tables will be useful in CS 241 for implementing symbol tables and other lookup tables. There will be times, however, when a list of (key,value) pairs will suffice.

(*make-hash*) creates a new hash table in which values are to be compared with *equal?*, as is necessary for strings; (*make-hasheq*) creates a table for which *eq?* (pointer comparison) is used). (*hash-set! table key value*) puts the given (key,value) pair into the given table (removing any other pair with the same key), and (*hash-ref table key failure*) retrieves the value associated with the given key. If there is no such value, then if *failure* is a procedure with zero arguments, it is called and it provides the value; if *failure* is not a procedure, it is returned by *hash-ref*. (*hash-remove table key*) will remove any (key,value) pair without replacing it.

(*hash-count table*) gives the number of elements in the hash table. You can iterate over hash tables with *hash-map* (which accumulates the result of applying the given function into a list) or *hash-for-each*. Details on these and other hash table functions are in section 3.13 of the Racket Reference manual. Racket also provides immutable hash tables, which can be updated in a purely functional fashion (the update function produces a new table).

Here is an example of removing duplicates from a list in (essentially) linear time. We process the list accumulatively, checking whether each element is a key in the hash table. If it is, it's a duplicate. If it isn't, we add it (with value *true*), and also add it to the accumulator.

```
(define (remove-duplicates lst)
  (local (
     (define ht (make-hash))
     (define (rd-helper lst acc)
       (cond
         [(empty? lst) (reverse acc)]
         [else
          (cond
            [(hash-ref ht (first lst) false)
               (rd-helper (rest lst) acc)]
            [else
              (hash-set! ht (first lst) true)
              (rd-helper (rest lst) (cons (first lst) acc))])])))
    (rd-helper lst empty)))

(remove-duplicates (list 2 3 4 3 2 1 2 3 4 2 3)) => (2 3 4 1)
```

10 Higher-Order Functions, Eval, Apply

The common presentation of *map* gives it two arguments (a function with one parameter, and a list). In fact, it can take k arguments, where the first argument is a function accepting $k-1$ parameters. The expression (*map* + *list1 list2 list3*) will produce a list which is the element-wise sum of the three argument lists. This is also true for *for-each*, and for the higher-order functions provided by the `list.ss` library, such as *foldr* and *foldl* (which are discussed below).

The Racket function *apply* consumes a function and a list, and applies the function with the elements of the list as an argument; (*apply* + (*list* 1 2 3)) produces 6. This is occasionally useful.

11 Implicit Begins and Internal Defines

There are implicit **begin** statements wrapped around the body of every **lambda** expression (including the implicit ones in function definition), and every **local**, **let**, and the other similar constructs discussed in the next section. There are also implicit **begin** statements wrapped around every **cond** answer (though not **if**, for obvious reasons). We used this in the "remove duplicates" example above.

Before the implicit **begin** in a **lambda** or function definition, Racket allows internal **define**s. This is an alternative to immediately using one of the constructs in the next section (internal definitions are converted to a use of **letrec**).

12 Local and Let

The single local binding construct **local** used in 135 is also available in the full Racket language, but Racket contains many similar constructs.

let takes a number of name-value bindings without using the keyword **define**, plus any number of body expressions. (**let** ([*x* 1][*y* 2]) (+ *x* *y*)) yields 3. None of the names are in the scope of any of the value-expressions.

let∗ is like **let**, but later bindings can use names defined in earlier bindings. **let**∗ is implemented using nested **let**s.

letrec adds the feature that all of the names are in the scope of all the value-expressions. It can be used to define recursive and mutually-recursive procedures.

The semantics of **local** is that the list of definitions at the beginning are lifted to the top level after being rewritten using unique names, with a similar rewriting of the body.

```
(local ((define x 1)
        (define y (+ x 1)))
    (+ x y)) =>
(define x_unique 1)
(define y_unique (+ x_unique 1))
(+ x_unique y_unique)
```

There is a variation on **let** known as "named **let**" which facilitates the writing of loops. 135 teaches how to do this using an accumulatively-recursive helper function. The following code does this using a named **let**, whose syntax 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.

```
(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)])))

(filter even? '(1 2 3 4 5 6)) => '(2 4 6)
```

This saves a little bit of typing and indentation.

Racket provides iterations and comprehensions (see section 11 of the Racket Reference), which are basically ways of writing nested loops in a very terse fashion. These may save you a lot of typing, at the cost of some time spent initially to learn to use them properly.

13 List Utilities

CS 135 students will be familar with the renamed basic list functions *first*, *rest*, and *empty?*, which make code easier to read. Then there is the higher-order function *filter*, as well as *foldr* which abstracts structural recursion on lists and *foldl* which abstracts accumulative recursion. Section 3.9 of the Racket Reference lists several more useful utilities, such as *assf* for working on association lists (lists of two-element lists), *findf*, *last-pair*, *sort*, *take*, *drop*, *append-map*, *filter-not*, *remove*, and *remove∗*. Knowing about these can save a lot of time coding small helper functions.

14 String Utilities

CS 135 and CS 136 spent little time on strings in Racket. There are some useful string functions defined in section 3.3 of the Racket Reference manual, but the really useful ones concern regular expressions.

Racket provides some functions which use regular expressions to manipulate strings. A regular expression is a way of specifying a string which is actually a pattern intended to match a portion of another string called the text string. The simplest regular expressions are strings like "abc", which just matches an occurrence of "abc" in the text string. But some characters are special in patterns. A period matches any character, so "a.c" will match an occurrence of "abc" but also "aqc". If you really want to match a period, you must escape it with a backslash: "a\\.bc". You can match zero or more occurrences of a character by putting an asterisk after it, and one or more by using a plus. So "a*bc" will match "bc", "abc", "aabc", and so on. There are more special characters, which you can read about in Chapter 9 of the Racket Guide manual and section 3.7 of the Racket Reference manual.

Playing with various patterns and texts, using *regexp-match*, is the best way to understand these. Note that you will learn about regular expressions in CS 241, though the syntax will be different, as it will be for regular expression libraries in other languages. *regexp-match*, given a pattern and text, produces a list of all substrings of the text that match the pattern. *regexp-split* is the opposite: given a pattern and text, it produces a list of strings which are all the parts that don't match. This is very useful.

```
(define test "abc,def,gh,i,jk,lmn")
(regexp-split "," test) => ("abc" "def" "gh" "i" "jk" "lmn")
```

You know from CS 135 and CS 136 that a typical way of working with a string in Racket is to convert it to a list of characters with *string->list*, process it in some fashion (perhaps making use of abstract list functions) and convert the result back with *list->string*. Many efficient implementations of this type of processing are provided by the SRFI 13 string library, which can be loaded by including the following line of code.

```
(require srfi/13)
```

This provides functions such as *string-append*, *string-take*, *string-drop*, *string-map*, and *string-fold*. It also provides search and replace functions. *string-join* and *string-tokenize* may be useful for simple ways of putting together or taking apart strings. (Some of these are available in Racket as well, so check the documentation.)

String processing is tricky in many languages. Suppose you wish to write code to create an answer list incrementally by tacking one new element at a time onto the end, using *append*. This will, as we saw in CS 135, take time proportional to the square of the length of the final result. The same thing will be true if we use *string-append* to tack short strings on to the end of an answer string one at a time. The solution in CS 135 was to accumulate the answer list in reverse order (just using *cons* to add each new element to the front of the accumulator) and reverse it once it was complete. The string solution, using SRFI 13, is to accumulate the answer string as a list of strings in reverse order and then use *string-concatenate-reverse* (a function you can easily write yourself) to concatenate the strings from the end of the list to the front, in time proportional to the length of the final result. There is also a function *string-concatenate* in case you have the list of strings in the correct order already.

15 Quasiquote and Matching

You probably know quote as a way of quickly writing lists. The lists '((1 2) (3 4)) and (*list* (*list* 1 2) (*list* 3 4)) are structurally equal.

Quasiquote is a way of using quote notation but interpolating expressions which are not quoted, but evaluated. To do this, an open-single-quote is used to start the list, and a comma to start an expression that is evaluated.

‘(1 2 ,(+ 3 4)) => '(1 2 7)

This avoids big expressions using *list*. You can also splice in the contents of an expression which evaluates to a list, using ,@.

‘(1 2 ,@(*build-list* 3 *add1*)) => '(1 2 1 2 3)

Trees are an important data structure in CS 241. From CS 135, you know that you can represent trees using either structures (example: binary search trees) or hierarchical lists (example: expression trees). The advantage of using structures is that code is more readable and you will get an understandable error if your code assumes a node is of one type but you provide it a node of a different type. The advantage of using lists is that the notation described above makes it easy to describe data, you have all the abstract list functions available to you, and you don't need to replicate code for similar but different types of structures. You will have to think carefully about how you represent data.

Some of the differences between these two representations can be erased with the use of pattern matching. Racket provides a number of pattern-matching functions that are very useful when you have to destructure and work with lists or structures. Instead of using list or structure selector functions to pick values out, and then using constructor functions to put together a related computed value, you can specify a pattern with variables that match parts of lists or structures, and then use those variables to specify the result.

The example below shows how a value is removed from a binary search tree using conventional syntax, and then using the *match* special form.

```
(define (remove-from-bst n bst)
  (cond
    [(boolean? bst) false]
    [(< (node-ssn bst) n)
      (make-node
        (node-ssn bst)
        (node-name bst)
        (node-left bst)
        (remove-from-bst n (node-right bst)))]
    [(> (node-ssn bst) n)
      (make-node
       (node-ssn bst)
       (node-name bst)
       (remove-from-bst n (node-left bst))
       (node-right bst))]
    [(boolean? (node-left bst)) ; (node-ssn bst) is n
      (node-right bst)]
    [(boolean? (node-right bst))
      (node-left bst)]
    [else
      (make-node
        (node-ssn (largest-in-bst (node-left bst)))
        (node-name (largest-in-bst (node-left bst)))
        (remove-largest-from-bst (node-left bst))
        (node-right bst))]))

(define (remove2 n bst)
  (match bst
    [#f false]
    [(struct node s nm l r)
      (cond
        [(< s n) (make-node s nm l (remove2 n r))]
        [(> s n) (make-node s nm (remove2 n l) r)]
        [(boolean? l) r]
        [(boolean? r) l]
        [else (match-let ([(struct node s1 nm1 _ _) (largest-in-bst l)])
                (make-node s1 nm1 (remove-largest-from-bst l) r))])]))
```

You can see how *match* takes an argument and a series of pattern-expression pairs; the first pattern that matches binds a number of pattern variables which are used in evaluating the corresponding expression. The "struct" indicates a structure (whose name follows); changing that to "list" would match a five-element list, binding the five names that follow (*node* is the first) to each of the values in it. An underscore or "_" will match anything. Note the use of *match-let* in the last case. This type of pattern matching is built into the languages ML and Haskell, but is not standard in regular Racket.

There are many more options for patterns described in chapter 8 of the Racket Reference. The example below demonstrates the use of literal symbols and computed predicates in simplifying an interpreter for arithmetic expressions such as '(+ (∗ 2 3) (− 4 1)). First, we present the version written without pattern matching.

```
(define (interp exp)
  (cond
    [(number? exp) exp]
    [(symbol=? (first exp) '+) (+ (interp (second exp)) (interp (third exp)))]
    [(symbol=? (first exp) '−) (− (interp (second exp)) (interp (third exp)))]
    [(symbol=? (first exp) '∗) (∗ (interp (second exp)) (interp (third exp)))]
    [(symbol=? (first exp) '/) (/ (interp (second exp)) (interp (third exp)))]))
```

This would be even less readable if we used *car*, *cadr*, and *caddr* instead of *first*, *second*, and *third*. It's not too bad as is, but if we want to add more syntax (keywords, variables, etc.) it would get complicated fast. Here's the pattern-matching version.

```
(define (interp exp)
  (match exp
    [(list '+ l r) (+ (interp l) (interp r))]
    [(list '− l r) (− (interp l) (interp r))]
    [(list '∗ l r) (∗ (interp l) (interp r))]
    [(list '/ l r) (/ (interp l) (interp r))]
    [(? number? n) n]))
```

These techniques can pay off when manipulating abstract syntax trees in CS 241.

16 Unnecessary Complications

The programs you have to write in CS 241 are fairly short and specialized, and there are some features of Racket that are overkill. You probably don't need to use modules (Chapter 6 of the Racket Guide manual), or the object/class system (Chapter 13). `class.ss` in MzLib. Modules facilitate code maintenance and reuse, as well as some of the optimizations that Racket performs, and it is necessary to master `class.ss` in order to use GUI components.

Continuations in general are overkill for CS 241, though Racket's exception-handling mechanism (10.1 in the Racket Guide manual) may come in handy for error recovery or for quickly returning from a deep recursion without having to check for errors or success flags at each step on the way back up. Macros can make your life easier at times, but the effort required to master them may not be worth the payoff in this course.