Appendix

This Appendix contains some of the content presented in tutorials, including additional examples and language syntax details.

You are still responsible for this content, even if it is not presented in the lectures.

A.1: CS 135 Review

In this Section we review some of the CS 135 content revisited in CS 136.

To be concise, we just refer to CS 135 instead of
“CS 135 or CS 145 or CS 115 and/or CS 116”.

Functions

Racket ***functions*** are defined with the `define` special form, which *binds* the function body to the name (identifier).

Racket uses *prefix* notation (instead of *infix* notation).

$$f(x, y) = (x + y)^2$$

```
(define (f x y)
  (sqr (+ x y)))
```

CS 135 terminology

```
(define (f x y)
  (sqr (+ x y)))

(f 3 4)     ; => 49
```

For the above example, we first *define* a function `f`, which has two ***parameters*** (`x` and `y`).

We then ***apply*** the function `f`, which ***consumes*** two ***arguments*** (the ***values*** `3` and `4`) and ***produces*** a single *value* (`49`).

In this course, we use different terminology: functions are *called* by *passing* argument values, and a value is *returned*.

Constants

Constants make your code easier to read and help you to avoid using “magic numbers” in your code.

Constants also give you *flexibility* to make changes in the future.

```
(define ontario-hst .13) ; effective July 1, 2010

(define (add-tax price)
  (* price (add1 ontario-hst)))
```

In this course, constants are often referred to as “variables”.

Running in Racket

A Racket program is a sequence of definitions and ***top-level expressions*** (expressions that are not inside of a definition).

When a program is "run" it starts at the top of the file, *binding* each definition and *evaluating* each expression. Racket also "outputs" the final value of each top-level expression.

```
(define (f x) (sqr x)) ; function definition
(define c (f 3))       ; variable definition

; top-level expressions:
(+ 2 3)       ; => 5
(f (+ 1 1))   ; => 4
(f c)         ; => 81
```

Conditionals

```
(cond
  [q1 a1]
  [q2 a2]
  [else a3])
```

The `cond` special form produces the first “answer” for which the “question” is true. The questions are evaluated in order until a true question or `else` is encountered.

Elementary data types

In addition to numbers, Racket also supports the elementary data types `'symbols` and `"strings"`.

- `'symbols` are atomic and useful when a small, fixed number of labels are needed. The only practical symbol function is comparison (`symbol=?`).

- `"strings"` are compound data and useful when the values are indeterminate or when computation on the contents of the string is required (*e.g.*, sorting).

Strings are composed of ***characters*** (*e.g.*, `#\c #\h #\a #\r`).

Structures

The `define-struct` special form defines a new compound structure with named ***fields***.

```
(define-struct my-posn (x y))
; defines posn, posn?, posn-x & posn-y

(define p (make-posn 3 4))
(posn-y p)    ; => 4
```

Lists

```
(define a1 (cons 1 (cons 2 (cons 3 empty))))
(define a2 (list 1 2 3))
(define a3 '(1 2 3))
```

You should be familiar with list functions introduced in CS 135, including: `cons`, `list`, `empty`, `first`, `rest`, `list-ref`, `length`, `append`, and `reverse`.

Binary search trees

In CS 135 we saw the Binary Search Tree (BST), where each node stores a `key` and a `value`

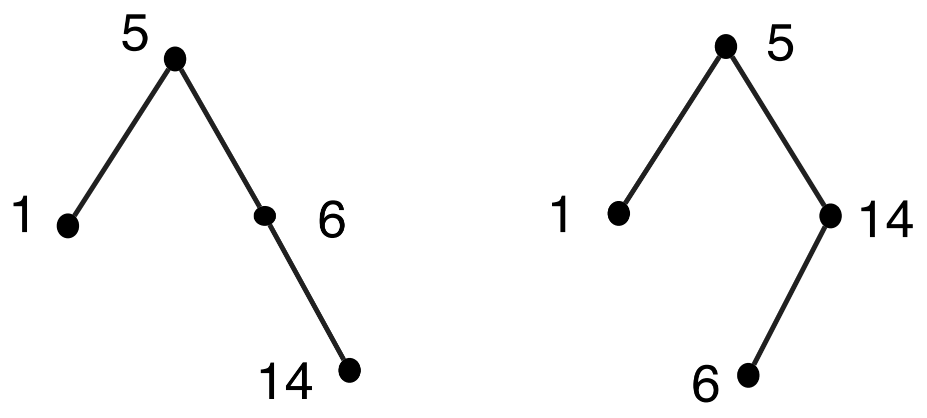
```
;; A Binary Search Tree (BST) is one of:
;; * empty
;; * a Node

(define-struct node (key val left right))
;; A Node is a (make-node Num Str BST BST)
;; requires: key > every key in left BST
;;           key < every key in right BST
```

We used the list function `empty` to represent an empty tree, but any ***sentinel value*** is fine. A popular alternative is `false`.

There can be several possible BSTs holding the same set of keys:

(we often only show the keys in a BST diagram)

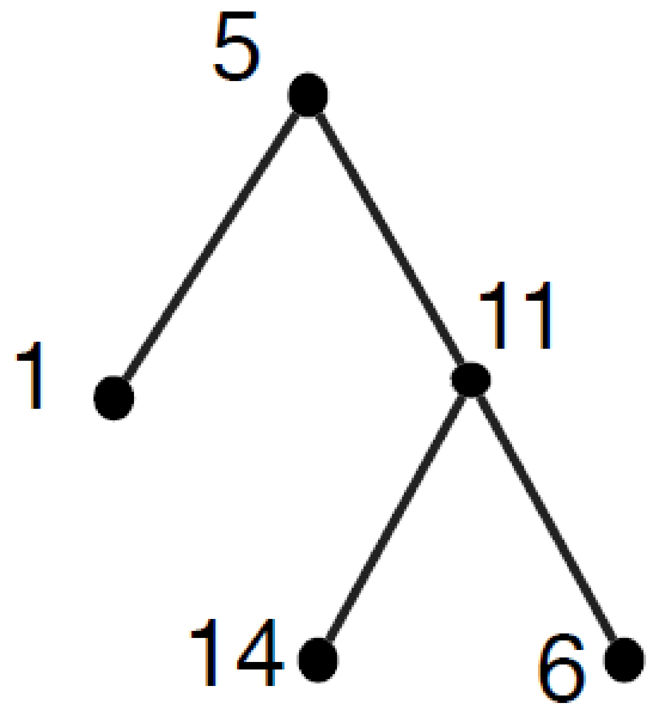


```
(define bst1 (make-node 5 "" (make-node 1 "" empty empty)
                 (make-node 6 "" empty (make-node 14 "" empty empty)))

(define bst2 (make-node 5 "" (make-node 1 "" empty empty)
                 (make-node 14 "" (make-node 6 "" empty empty) empty))
```

Remember that the left and right subtrees must also be BSTs and maintain the ordering property.

This is **not** a BST:



BST review

You should be comfortable inserting key/value pairs into a BST.

You should also be comfortable searching for a key in a BST.

You are *not* expected to know how to delete items from a BST.

You are *not* expected to know how to re-balance a BST.

Abstract list functions

In Racket, functions are also ***first-class*** values and can be provided as arguments to functions.

The built-in abstract list functions accept functions as parameters. You should be familiar with the abstract list functions `filter`, `map`, `foldr`, `foldl`, and `build-list`.

```
(define lst '(1 2 3 4 5))

(filter odd? lst) ; => '(1 3 5)
(map sqr lst)     ; => '(1 4 9 16 25)
(foldr + 5 lst)   ; => 20
```

Lambda

In Racket, `lambda` can be used to generate an anonymous function when needed.

```
(define lst '(1 2 3 4 5))

(filter (lambda (x) (> x 2)) lst) ; => '(3 4 5)

(build-list 7 (lambda (x) (sqr x)))
  ; => '(0 1 4 9 16 25 36)
```

A.2 Full Racket

In this course, we continue to use Racket, but we use the ***"full Racket"*** language (`#lang racket`), not one of the Racket "teaching languages".

There are some minor differences, which we highlight.

The first line of your Racket (`.rkt`) files must be:

`#lang racket`.

In DrRacket, you should also set your language to:

"Determine language from source".

To save space, we often omit `#lang racket` in these notes.

Even though you now have the full `#lang racket` available to you, you should not “go crazy” and start using every advanced function and language feature available to you.

For your assignments and exams, stick to the language features discussed in class.

If you find a Racket function that’s “too good to be true”, consult the course staff to see if you are allowed to use it on your assignments.

The objective of the assignments is **not** for you to go “hunting” for obscure built-in functions to do your work for you.

Functions without parameters

Functions can be defined without parameters. In contracts, use `Void` to indicate there are no parameters.

```
(define magic-variable 7)

; magic-function: Void -> Int
(define (magic-function) 42)

(define (use-magic x)
  (* x magic-variable (magic-function)))
```

Parameter-less functions might seem awkward now, but later we see how they can be quite useful.

Booleans

In full Racket, the values `#t` and `#f` are used to represent true and false. `true` and `false` are constants defined with those values.

Full Racket uses a wider interpretation of “true”:

Any value that is not `#f` is considered true.

Many computer languages consider *zero* (`0`) to be false and any *non-zero* value is considered true. This is how C behaves.

Because Racket uses `#f`, it is one of the few languages where zero is considered true.

Logical operators and & or

The special forms `and` and `or` behave a little differently in full Racket:

`and` produces `#f` if any of the arguments are `#f`, `#t` if there are no arguments, otherwise the *last* argument:

```
   (and 5 6 7)       ; => 7
```

`or` produces either `#f` or the *first* non-false argument:

```
   (or #f #f 5 6 7)  ; => 5
```

Conditionals

```
(cond
  [q1 a1]
  [q2 a2]
  [else a3])
```

In full Racket, the questions do not have to be Boolean values because any value that is not `#f` is considered true.

In full Racket, `cond` does not produce an error if all questions are false: it produces `#<void>`, which we discuss later.

The `if` special form can be used if there are only two possible answers:

```
(cond
  [q1 a1]
  [else a2])
```

is equivalent to:

```
(if q1 a1 a2)
```

`cond` is preferred over `if` because it is more flexible and easier to follow. We only demonstrate `if` because there is a C equivalent (the `?:` operator).

Structures

Full Racket provides a more compact `struct` syntax for convenience:

- `struct` can be used instead of `define-struct`
- the `make-` prefix can be omitted (`posn` instead of `make-posn`)

```
(struct posn (x y)) ; defines posn, posn?, posn-x & posn-y
(define p (posn 3 4))
(posn-y p) ; => 4
```

For now, you should include `#:transparent` in your `struct` definitions (this is discussed in Section 02).

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

Lists

Full Racket does not enforce that the second argument of `cons` is a list, so it allows you to `cons` any two values (*e.g.*, `(cons 1 2)`), but it's not a valid list, so don't do it!

member

In full Racket there is no `member?` function and `member` is not a predicate.

`(member v lst)` produces `#f` if `v` does not exist in `lst`. If `v` does exist in `lst`, it produces the tail of `lst`, starting with the first occurrence `v`.

```
(member 2 (list 1 2 3 4)) ; => '(2 3 4)
```

Recall, that `'(2 3 4)` is “true” (not false), so it still behaves the same in most contexts.

You can define your own `member?` predicate function:

```
(define (member? v lst) (not (false? (member v lst))))
```

Implicit local

The `local` special form creates a new local **scope**, so identifiers defined within the local are only available within the `local` body.

In full Racket, you do not need to explicitly use `local`, as there is an *implicit* ("built-in") `local` in every function body.

```
  (define (t-area a b c)
    (local
      [(define s (/ (+ a b c) 2))]
      (sqrt (* s (- s a) (- s b) (- s c)))))
```

Is equivalent to:

```
  (define (t-area a b c)
     (define s (/ (+ a b c) 2))
     (sqrt (* s (- s a) (- s b) (- s c))))
```

The variable `s` is implicitly `local`.

check-expect

The `check-expect` special form should not be used in full Racket.

In Section 07 we introduce more advanced testing methods.

For now, you can simply use `equal?` instead of `check-expect`.

```
(define (my-add x y) (+ x y))

;; instead of
;; (check-expect (my-add 1 1) 2)
;; (check-expect (my-add 1 -1) 0)

(equal? (my-add 1 1) 2)
(equal? (my-add 1 -1) 0)
```

A.3 Memory

In this section we briefly discuss number representations and computer memory.

Decimal notation

In ***decimal*** representation (also known as ***base 10***) there are **ten** distinct ***digits*** (0123456789).

When we write the number 7305 in base 10, we interpret it as

$$7305 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0.$$

4 decimal digits can represent 10^4 $(10,000)$ different possible values (*i.e.,* $0 \ldots 9999$).

The reason base 10 is popular is because we have 10 fingers.

Binary notation

In ***binary*** representation (also known as ***base 2***) there are **two** distinct digits (01).

When we write the number 1011010 in binary, we interpret it as

$1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
$= 64 + 16 + 8 + 2 = 90$ (in base 10).

4 binary digits, can represent 2^4 (16) different possible values $(0 \ldots 1111)$ or $(0 \ldots 15)$ in base 10.

In CS 251 / CS 230 you will learn more about binary notation.

Hexadecimal notation

In ***hexadecimal (hex)*** representation (also known as ***base 16***) there are **sixteen** distinct digits (0123456789ABCDEF).

When we write the number 2A9F in hex, we interpret it as

$$2 \times 16^3 + 10 \times 16^2 + 9 \times 16^1 + 15 \times 16^0$$
$= 8192 + 2560 + 144 + 15 = 10911$ (in base 10).

The reason *hex* is so popular is because it is easy to switch between binary and hex representation. A single hex digit corresponds to exactly 4 bits.

Conversion table

Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Binary/Hex conversion

To distinguish between 1234 (decimal) and 1234 (hex), we prefix hex numbers with “0x” (0x1234).

Here are some simple hex / binary conversions:

```
0x1234 = 0001 0010 0011 0100
0x2A9F = 0010 1010 1001 1111
0xFACE = 1111 1010 1100 1110
```

In C, a number with the prefix `0x` is interpreted as a hex number:

```c
int num = 0x2A9F; //same as num = 10911
```

A number prefixed with a zero `0` is interpreted as an *octal* numbers (base 8).

```c
int num = 025237; //same as num = 10911
```

This has been the source of some confusion.

Memory Capacity

To have a better understanding of the C memory model, we provide a *brief* introduction to working with *bits* and *bytes*.

You are probably aware that internally, computers work with **bits**.

A bit of *storage* (in the memory of a computer) is in one of two **states**: either 0 or 1.

A traditional light switch can be thought of as a bit of storage.

Early in computing it became obvious that working with individual bits was tedious and inefficient. It was decided to work with 8 bits of storage at a time, and a group of 8 bits became known as a ***byte***.

Each byte in memory is in one of 256 possible states.

With today's computers, we can have large memory capacities:

- 1 KB = 1 kilobyte = 1024 (2^{10}) bytes*
- 1 MB = 1 megabyte = 1024 KB = 1,048,576 (2^{20}) bytes*
- 1 GB = 1 gigabyte = 1024 MB = 1,073,741,824 (2^{30}) bytes*

$*$ The size of a kilobyte can be 1000 bytes or 1024 bytes, depending on the context. Similarly, a megabyte can be 10^6 or 2^{20} bytes, *etc.*.

Manufacturers often use the measurement that makes their product appear better. For example, A terabyte (TB) drive is almost always 10^{12} bytes instead of 2^{40}.

To avoid confusion in scientific use, a standard was established to use KB for 1000 bytes and KiB for 1024 bytes, *etc.*.

In general use, KB is still commonly used to represent both.

Primary Memory

Modern computers have ***primary memory*** in addition to ***secondary storage*** (hard drives, solid state drives, flash drives, DVDs, etc.).

The characteristics of primary memory and secondary storage devices vary, but *in general*:

Primary Memory:

- very fast (nanoseconds)
- medium capacity ($\approx$GB)
- high cost ($) per byte
- harder to remove
- erased on power down

Secondary Storage:

- ($\approx$20x-1000x) slower
- large capacity ($\approx$TB)
- low cost ($) per byte
- removable or portable
- persistent after power down

In practice, programs can only “run” in primary memory.

When you “launch” a program, it is copied from secondary storage to primary memory before it is “run”.

In this course, we are always referring to *primary* memory, which is also known as ***Random Access Memory (RAM)***. With RAM you can access any individual byte directly and you can access the memory in any order you desire (randomly).

Traditional secondary storage devices (hard drives) are faster if you access data *sequentially* (not randomly).

Primary memory became known as **RA**M to distinguish it from sequential access devices.

The term “RAM” is becoming outdated, as solid state drives and flash drives use random access. Also, modern RAM can be faster when accessed sequentially (in “bursts”).

Regardless, when you encounter the term “RAM”, you should interpret it as *“primary memory”*.