Appendix A

This appendix contains additional content, examples and language syntax details that may not be covered in the lectures.

Some of this content will be covered in tutorials.

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)))
```
(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)

```
5
/   \
/     \
1      6
|      |
14     14
```

```
(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.

```racket
(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:

\[
(\text{and } 5\ 6\ 7) \quad ; \quad \Rightarrow\ 7
\]

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

\[
(\text{or } \#f\ \#f\ 5\ 6\ 7) \quad ; \quad \Rightarrow\ 5
\]
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 Racket Modules

There are two Racket special forms that allow us to work with modules: provide and require.

provide is used in a module to specify the identifiers or “bindings” (e.g., function names) that are available to clients.

require is used to identify the module (a file name) that the current file depends upon.

There is also a module special form in Racket and many other module support functions that we do not discuss in this course.
Creating a module

In full Racket, adding a `provide` special form automatically makes a `.rkt` file a module.

Conceptually, the `provide` special form can be seen as the “opposite” of the `local` special form: `local` makes definitions “invisible” to the outside, whereas `provide` makes definitions “visible” to the outside.

Any private functions you wish to hide should not be provided.
example: provide

(provide function-a function-b)

(define (function-a p) ...)

(define (function-b p1 p2) ...)

(define (hidden-helper n) ...) ; not provided

In this example, the function hidden-helper is private and not visible outside of the module.
The require special form

When the require special form is evaluated, it “runs” all of the code in the required module and makes the provided identifiers or “bindings” available.

require also “outputs” the final value of any of the top-level expressions in the module.

Modules should only have definitions, not any top-level expressions.
A.4 C Modules

In this appendix we further explore modularization in C modules. In particular, we focus on global scoping and the behaviour of \#include.

To better understand C’s modularization behaviour, it helps to understand what happens when Seashell “runs” your program.
Running a program

In Racket, require temporarily suspends the execution of the main (client) program to run (“process”) the module implementation.

C has no mechanism to automatically process module implementations.

To build a C program you must specify all of the files you wish to package together. For example:

- main.c
- module1.c
- module2.c
In the “real world” (outside of Seashell), you must *manually* specify all of the files you wish to package together to make the program.

Seashell *automatically* scans your “run file” to determine which module implementations should be packaged together to build your program.

We are presenting a simplified view of *linking* in C.
For this example, Seashell automatically detects that main.c also requires module1.c and module2.c and packages them all together:

```c
// main.c: set as the "run file"

#include "module1.h"
#include "module2.h"

int main(void) {
    //...
}
```

Seashell can also use binary (.o) implementation files instead of source code (.c) implementation files.
In other words, when you

```
#include "module1.h"
```

it is **Seashell** that adds the implementation `module1.c` to the program, not C.

Remember, the `#include` *preprocessor directive* simply “inserts” the contents of one file into another and does nothing else.
So while it may appear that placing a function **declaration** in an **interface** (.h file) “provides” the function, it is not actually the case. The function is provided when Seashell packages the two files together. This works because, by default, all functions have **program scope**, and are available to all other files once they are packaged together.
If two different modules both have a function (or a global variable) with the same name, C will encounter an error when trying to package them together. This is known as a “collision”.

There are two key strategies to help avoid collisions:

- For *provided* functions (and global variables), use meaningful identifiers that reflect the name of the module. This will reduce the chance of a collision.

- For functions (and global variables) that are *not provided*, use the static keyword to give them *module scope*. 
A.5 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 . . . 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

\[
1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0
\]

\[
= 64 + 16 + 8 + 2 = 90 \text{ (in base 10)}.
\]

4 binary digits, can represent \(2^4 \) (16) different possible values (0 . . . 1111) or (0 . . . 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

\[
2 \times 16^3 + 10 \times 16^2 + 9 \times 16^1 + 15 \times 16^0 \\
= 8192 + 2560 + 144 + 15 = 10911 \text{ (in base 10)}.
\]

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

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

\[
\begin{align*}
0\times1234 & = 0001 \ 0010 \ 0011 \ 0100 \\
0\times2A9F & = 0010 \ 1010 \ 1001 \ 1111 \\
0\timesFACE & = 1111 \ 1010 \ 1100 \ 1110
\end{align*}
\]
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 (≈GB)
- high cost ($) per byte
- harder to remove
- erased on power down

**Secondary Storage:**
- (≈20x–1000x) slower
- large capacity (≈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 **RAM** 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”.

A.6 I/O & Testing

In this section we briefly discuss additional I/O and testing methods
Advanced Tracing

You can use different **tracing levels** to indicate how much detail you want in your tracing output. Once you have debugged your code, you can simply set the level to zero and turn off all tracing.

```c
int TRACELEVEL = 2;

int main(void) {
    if (TRACELEVEL >= 3) printf("starting main\n");
    // ...

    for (int i=0; i < 10; ++i) {
        sum += i;
        if (TRACELEVEL >= 2) printf("i = %d, sum = %d\n", i, sum);
    }
    //...
}
```
Input

The Racket read function attempts to read (or “get”) a value from the keyboard. If there is no value available, read pauses the program and waits until there is.

(define my-value (read))

read may produce a special value (#<eof>) to indicate that the End Of File (EOF) has been reached.

EOF is a special value to indicate that there is no more input.
The read function is quite complicated, so we present a simplified overview that is sufficient for our needs.

read interprets the input as if a single quote ’ has been inserted before each “value” (again, not really but close enough).

If your value begins with an open parenthesis (, Racket reads until a corresponding closing parenthesis ) is reached, interpreting the input as one value (a list).

Text is interpreted as symbols, not a string (unless it starts with a double-quote "). The symbol->string function is often quite handy when working with read.
example: read

(define (read-to-eof)
  (define r (read))
  (printf "~v\n" r)
  (cond [(not (eof-object? r)) (read-to-eof)]))

1
two
"three"
(1 two "three")
Ctrl-D

1
'two
"three"
'(1 two "three")
#<eof>
User interaction

With the combination of input & output, we can make *interactive* programs that change their behaviour based on the input.

```scheme
(define (get-name)
  (printf "Please enter your first name:\n")
  (define name (read))
  (printf "Welcome, to our program, ~a!\n" name)
name)
```
(define (madlib)
  (printf "Let’s play Mad Libs! Enter 4 words :
")
  (printf "a Verb, Noun, Adverb & Adjective :
")
  (define verb (read))
  (define noun (read))
  (define adverb (read))
  (define adj (read))
  (printf "The two ~as were too ~a to ~a ~a.
" noun adj verb adverb))

(madlib)
Interactive testing

In DrRacket, the *interactions window* was quite a useful tool for debugging our programs.

In Seashell, we can create **interactive I/O testing clients**.

Consider an example with a simple arithmetic module.

```c
// addsqr.h

// sqr(x) returns x*x
int sqr(int x);

// add(x,y) returns x+y
int add(int x, int y);
```
example: interactive I/O testing in C

#include "addsqr.h"

int main(void) {
    char func;
    int x,y;
    while (1) {
        if (scanf(" %c", &func) != 1) break;
        if (func == 'x') break;
        if (func == 'a') {
            scanf("%d", &x); // you should check scanf return values
            scanf("%d", &y);
            printf("add %d %d = %d\n", x, y, add(x,y));
        } else if (func == 's') {
            scanf("%d", &x);
            printf("sqr %d = %d\n", x, sqr(x));
        }
    }
}
With this *interactive* i/o testing module, tests are entered via the keyboard.

**Input:**

- a 3 4
- a -1 0
- a 999 -1000
- s 5
- s -5
- s 0
- x

**Output:**

- add 3 4 = 7
- add -1 0 = -1
- add 999 -1000 = -1
- sqr 5 = 25
- sqr -5 = 25
- sqr 0 = 0
One big advantage of this i/o interactive testing approach is that we can experiment with our module without having to program (code) each possible test.

It’s also possible that someone could test the code without even knowing how to program.

A disadvantage of this approach is that it can become quite tedious to rely on human input at the keyboard.

Fortunately, the Seashell environment has support to automate interactive testing with .in and .expect files.
A.7 Efficiency

In this section we provide more efficiency examples.
Examples: recurrence relations

For simplicity and convenience (and to avoid any equal? issues) we use lists of integers in these examples.

(define (member1 e lon)
  (cond [(empty? lon) #f]
        [ (= e (first lon)) #t]
        [else (member1 e (rest lon))])))

\[ T(n) = O(1) + T(n - 1) = O(n) \]

(define (member2 e lon)
  (cond [(zero? (length lon)) #f]
        [ (= e (first lon)) #t]
        [else (member2 e (rest lon))])))

\[ T(n) = O(n) + T(n - 1) = O(n^2) \]
(define (has-duplicates? lon)
  (cond [(empty? lon) #f]
        [(member (first lon) (rest lon)) #t]
        [else (has-duplicates? (rest lon))]))

\[ T(n) = O(n) + T(n-1) = O(n^2) \]

(define (has-same-adjacent? lon) ;; O(n)
  (cond [(or (empty? lon) (empty? (rest lon))) #f]
        [(= (first lon) (second lon)) #t]
        [else (has-same-adjacent? (rest lon))]))

(define (faster-has-duplicates? lon)
  (has-same-adjacent? (sort lon <)))

\[ T(n) = O(n \log n) + O(n) = O(n \log n) \]
(define (find-max lon)
  (cond
    [(empty? (rest lon)) (first lon)]
    [(> (first lon) (find-max (rest lon))) (first lon)]
    [else (find-max (rest lon))]))

\[ T(n) = O(1) + 2T(n - 1) = O(2^n) \]

(define (fast-max lon)
  (cond
    [(empty? (rest lon)) (first lon)]
    [else (max (first lon) (fast-max (rest lon)))])

\[ T(n) = O(1) + T(n - 1) = O(n) \]

(define (clean-max lon)
  (apply max lon))

\[ T(n) = O(n) \]
A.8 Linked lists

In Section 11 we present a linked list with a wrapper strategy.

We also intentionally avoid following a very “functional” approach to help prevent mixing paradigms.

In this Section we present additional examples, still using the same llnode structure:

```
struct llnode {
    int item;
    struct llnode *next;
};
```
Functional approach

We can create helper functions to create a “functional” atmosphere:

```c
int first(struct llnode *lst) {
    assert (lst != NULL);
    return lst->item;
}

struct llnode *rest(struct llnode *lst) {
    assert (lst != NULL);
    return lst->next;
}

struct llnode *empty(void) {
    return NULL;
}

bool is_empty(struct llnode *lst) {
    return lst == empty();
}
```
At the heart of Racket’s functional list approach is the cons function.

This C cons returns a **new** node that *links* to the rest.

```c
struct llnode *cons(int f, struct llnode *r) {
    struct llnode *new = malloc(sizeof(struct llnode));
    new->item = f;
    new->next = r;
    return new;
}
```

This is very similar to how Racket’s cons is implemented.
We can use our new C cons function the same way we use the Racket cons function.

```
struct llnode *my_list =
    cons(10, cons(3, cons(5, cons(7, empty()))));
```

![Diagram of linked list](image)

We can also use cons in different ways (e.g., with mutation).

```
struct llnode *my_list = empty();
my_list = cons(7, my_list);
my_list = cons(5, my_list);
my_list = cons(3, my_list);
my_list = cons(10, my_list);
```
Using this approach, the code to make a *duplicate* of a list is very straightforward.

```c
struct llnode *list_dup(struct llnode *lst) {
    if (is_empty(lst)) return empty();
    return cons(first(lst), list_dup(rest(lst)));
}
```

In Section 11 we present a *list_dup* function that is more awkward.
A C function written with cons will return a **new list** (a “functional” approach).

Consider the following “square list” function:

```c
struct llnode *sqr_list(struct llnode *lst) {
    if (is_empty(lst)) return empty();
    return cons(first(lst) * first(lst),
                sqr_list(rest(lst)));
}
```
To correctly use the `sqr_list` function, the result should be stored in a separate variable.

```
struct llnode *a = cons(10, cons(3, cons(5, cons(7, empty()))));
struct llnode *b = sqr_list(a);
```

Unfortunately, if the function is misunderstood or used incorrectly, it can create a memory leak.
The following two statements each create a memory leak.

```c
a = sqr_list(a); // original list is lost
sqr_list(a); // new list is lost
```

![Diagram of list operations](image)

The best strategy to avoid a memory leak is to provide a clear contract.
The cons function can cause a similar problem:

```c
struct llnode *a = cons(7, empty());
cons(5, a);  // memory leak
```

This is not a concern in Racket because it uses garbage collection and automatically frees memory.
We also have to worry about node sharing.

```c
struct llnode *a = cons(4, cons(5, cons(6, empty())));
struct llnode *b = cons(3, a);
struct llnode *c = cons(1, cons(2, b));
```
To properly use cons in an imperative environment, it helps to imagine that the second parameter (the “rest”) is to be considered “inaccessible” or an “invalid” list.

Only the returned value should be used as a list.

```
// cons(f, r) produces a new list with f added to the front of the list r
// effects: r is no longer a valid list pointer
struct llnode *cons(int f, struct llnode *r);
```
Working without a wrapper

In Section 11 we wrote an add_front function that worked with the wrapper structure.

Without the wrapper, we can simply use cons to add to the front of the list.

However, we have to ensure that the caller uses cons properly (storing the result and considering the second parameter invalid).
Instead, we want to write an `add_front` function where we pass the list as a parameter.

For example, we would like to have the following code sequence.

```c
struct llnode *lst = NULL;

add_front(7, lst);  // won't work
add_front(5, lst);  // won't work
```

Remember, to have a function change a variable (i.e., `lst`), we need to pass a `pointer` to the variable.

```c
add_front(7, &lst);
add_front(5, &lst);
```

Since `lst` is already a pointer, we need to pass a `pointer to a pointer`.
void add_front(int n, struct llnode **ptr_front) {
    struct llnode *new = malloc(sizeof(struct llnode));
    new->item = n;
    new->next = *ptr_front;
    *ptr_front = new;
}

Example:

struct llnode *lst = NULL;
add_front( 7, &lst);
add_front( 5, &lst);
add_front( 3, &lst);
add_front(10, &lst);

10 → 3 → 5 → 7
This *pointer-to-a-pointer* approach is also necessary if the first item may be removed.

```c
int remove_from_front(struct llnode **ptr_front) {
    struct llnode *front = *ptr_front;
    int retval = front->item;
    *ptr_front = front->next;
    free(front);
    return retval;
}
```

Instead of returning nothing (*void*), it is more useful to return the value of the item being removed.

Avoiding this *pointer-to-a-pointer* is one of the advantages of using a wrapper strategy.
An alternative “destructive” approach uses a Racket-like programming interface (functions produce new lists), but each list passed to a function may be destroyed (freed).

```c
struct llnode *insert(int n, struct llnode *slst) {
    if (is_empty(slst)) {
        return cons(n, empty());
    } else if (n <= first(slst)) {
        return cons(n, slst);
    } else {
        int f_backup = first(slst);
        struct llnode *r_backup = rest(slst);
        free(slst);  
        return cons(f_backup, insert(n, r_backup));
    }
}
```

This approach has been taught in previous offerings of CS 136.