Assignment:	8
Due:	Tuesday, November 21, 9:00pm
Language level:	Intermediate Student with Lambda
Allowed recursion:	None (see notes below)
Files to submit:	`intro.rkt`, `nestlist.rkt`, `suggest.rkt`, `omit.rkt`
Warmup exercises:	HtDP (*Without using explicit recursion:* 9.5.2, 9.5.4), 19.1.5, 20.1.1, 20.1.2, 24.0.7, 24.0.8
Practice exercises:	HtDP 19.1.6, 20.1.3, 20.2.4, 21.2.3, 24.0.9

不用这个你用不了 lambda，没lambda
这个作业一半没法做

- **Make sure you read the OFFICIAL A8 post on Piazza** for the answers to frequently asked questions.

  map filter foldr build-list

- The Piazza post will also contain **a list of the allowed Abstract List Functions** (and other built-in functions).
- **You may not write explicitly recursive functions; that is, functions that involve an application of themselves, either directly or via mutual recursion, unless specifically permitted in the question.**

  除非题目允许，不可以写任何 recursion

- In this assignment your *Code Complexity/Quality* grade will be determined both by how clear your approach to solving the problem is, and how effectively you use Abstract List Functions and local constants/functions. You should include local definitions to avoid repetition of common subexpressions, to improve readability of expressions and to improve efficiency of code.
- You may reuse the provided examples, but you should ensure you have an appropriate number of examples and tests. When working with Abstract list Functions, an appropriate number of tests is often quite small (exhaustive testing is usually not necessary). For most functions, a few examples are usually sufficient.
- Your solutions must be entirely your own work.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.

注意，explicit Recursion 指的就是你直接用 recursion，你用 map filter foldr build-list 就是 implicit recursion
间接 recursion

Here are the assignment questions you need to submit.

1. Perform the assignment 8 questions using the online evaluation "Stepping Problems" tool linked to the course web page and available at

   https://www.student.cs.uwaterloo.ca/~cs135/stepping.

   The instructions are the same as those in assignment 3; check there for more information if necessary.

2. For this introductory question, you are to implement some straightforward functions using Abstract List Functions (ALFs). Place your solution in the file `intro.rkt`.

   REMINDER: As mentioned in the preamble, except as noted in questions 3(a) and 4(b) *you may not write explicitly recursive functions*. You must use abstract list functions.

   闭着眼睛都应该能做

   (a) Write `keep-ints` from Assignment 04. 看下一页

   (b) Write `contains?` from Assignment 04 (for this part, you are not allowed to use `member?` or `member`). 看下一页

   (c) Write `lookup-al` from Section 06 Slide 57 (you are not allowed to use `assoc` or `assq`). 看下一页

   (d) Write a function `extract-keys` which consumes an Association List (AL) and produces a list of all of the keys in the association list (in the same order they appear in the AL).

   (e) Write `sum-positive` from Assignment 04. 看下一页

   (f) Write `countup-to` from Section 06 Slide 23 (hint: use `build-list`) (for this part you are not allowed to use `range`). 看一下页

   (g) Write a function `shout` that consumes a list of strings and produces the same list of strings, but in all UPPERCASE. You **must** use the built-in function `char-upcase` to do the conversion. 每个string也要用abstract list function来变大写，先拆成listof char，再一个一个变，再拼起来

   ```
   (shout '("get" "off" "my" "lawn")) => '("GET" "OFF" "MY" "LAWN")
   ```

   (h) Write a function `make-validator` that consumes a `(listof Any)` and produces a predicate function (similar to the way that `make-adder` produces a function in Section 10). The produced function consumes a single item of type `Any` and produces a `Boolean` that determines if the item appears in the list that was consumed by `make-validator`.

   就是produce一个lambda，你都可以用(b)当helper

   ```
   (define primary-colour? (make-validator '(red blue green)))
   (primary-colour? 'red) => true
   ```

(a) (d) Write a function *keep-ints* which consumes a (*listof Any*). It produces a list that contains only the integers in the given list, in their original order. For example, (*keep-ints* (*cons* 'a (*cons* 1 (*cons* "b" (*cons* 2 *empty*))))) ⇒ (*cons* 1 (*cons* 2 *empty*)). You may find the predicate *integer?* useful. For full marks you may not build and then reverse a list.

(b) (b) Write a predicate *contains?* which consumes a value, *elem*, and a (*listof Any*). It produces *true* if *elem* is in the list and *false* otherwise. For example, (*contains?* 'fun (*cons* 'racket (*cons* 'is (*cons* 'fun *empty*)))) ⇒ *true*.

Note that Racket contains the built-in functions *member?* and *member* which are identical to *contains?* except for the name. Of course, **you can't use them on this assignment**. In class we've told you to use *equal?* sparingly. This is an appropriate place to use it because you don't know the types of what you are comparing.

(c)

```
;; (lookup-al k alst) produces the value corresponding to key k,
;;     or false if k not present
;; lookup-al: Num AL → (anyof Str false)

(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(equal? k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))]))
```

(e)

(a) Write a function *sum-positive* which consumes a list of integers and produces the sum of the positive integers in that list. The empty list sums to 0. For example, (*sum-positive* (*cons* 5 (*cons* −3 (*cons* 4 *empty*)))) ⇒ 9.

(f)

```
;; (countup-to n b) produces a list from n...b
;; countup-to: Int Int → (listof Int)
;; requires: n <= b
;; Example:
(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

(define (countup-to n b)
  (cond [(= n b) (cons b empty)]
        [else (cons n (countup-to (add1 n) b))]))
```

这个其实就是 General Tree 的另外一种表达方式，唯一的区别就是 nested list 可以有空

3. On Slide 76 of Section 08 we introduced the idea of **nested lists** (with arbitrary nesting). In this question we explore functions that abstract recursive behaviour on nested lists. In other words, we wish to write a higher-ordered function that generalizes the behaviour of functions such as `count-items` on Slide 82 and `flatten` on Slide 85. Also note the data definition of a `Nest-List-X` provided on Slide 80 and the corresponding template function on Slide 81 (you do not have to include them in your assignment). All of these slides will help guide your approach. However, note that instead of using the unknown predicate function `X?` as it appears on Slide 81 (or `number?` on Slides 82 & 85), it is much easier to switch the order of the two non-base cases and use the built-in function `list?` function to determine if the first element is a list. This allows us to write the function without knowing the type of `X`. Place your solution in the file `nestlist.rkt`. You do not have to provide a data definition for `Nest-List-X`.

让你写一个给 nested list 用的 foldr，就是把所有东西组合起来

   (a) Write a *nested* version of `foldr` named `nfoldr` that behaves very similar to `foldr`, except it consumes two “combine” functions instead of one. The first function is applied when the first element in the list is of type `X`, and the second is applied when the first element is a `list`. The third parameter is the base case and the fourth parameter is the nested list. The contract for `nfoldr` is as follows:

```
;; nfoldr: (X Y -> Y) (Y Y -> Y) Y Nested-Listof-X -> Y
```

仔细理解处理 nested list 的套路

The functions `count-items` and `flatten` (from Slides 82 & 85) can be implemented using `nfoldr` as follows:

看下一页，

```
(define (count-items nln) (nfoldr (lambda (x y) (add1 y)) + 0 nln))

(count-items '(1 (2 3) () ((4)))) => 4

(define (flatten lst) (nfoldr cons append empty lst))

(flatten '(1 (2 3) () ((4)))) => '(1 2 3 4)
```

你的 nfoldr 写好以后就可以这样来做 course note 的题

To help further illustrate how `nfoldr` works, consider the following applications:

```
(nfoldr f g b '()) => b

(nfoldr f g b '(1 2 3)) => (f 1 (f 2 (f 3 b)))

(nfoldr f g b '(1 (2 3) 4)) => (f 1 (g (f 2 (f 3 b)) (f 4 b)))

(nfoldr f g b '(1 (2 3) () ((4))))
=> (f 1 (g (f 2 (f 3 b)) (g b (g (g (f 4 b) b) b))))
```

做 nfoldr 肯定需要 recursion

You may use explicit recursion (pure structural) to `define` your `nfoldr` function. However, as stated at the start of this assignment, *all* of the remaining functions in this question must not use explicit recursion. Instead, they must be implemented by using your `nfoldr` function

The function count-items

```
;; count-items: Nest-List-Num → Nat
(define (count-items nln)
  (cond [(empty? nln) 0]
        [(number? (first nln))
         (+ 1 (count-items (rest nln)))]
        [else (+ (count-items (first nln))
                 (count-items (rest nln)))]))
```



```
;; flatten: Nest-List-Num → (listof Num)
(define (flatten lst)
  (cond [(empty? lst) empty]
        [(number? (first lst))
         (cons (first lst) (flatten (rest lst)))]
        [else (append (flatten (first lst))
                      (flatten (rest lst)))]))
```

任何一个问题如果做不出来，就先用 recursion写，然后再改成 nfoldr

b, c, def

都可以用 nfoldr 解决

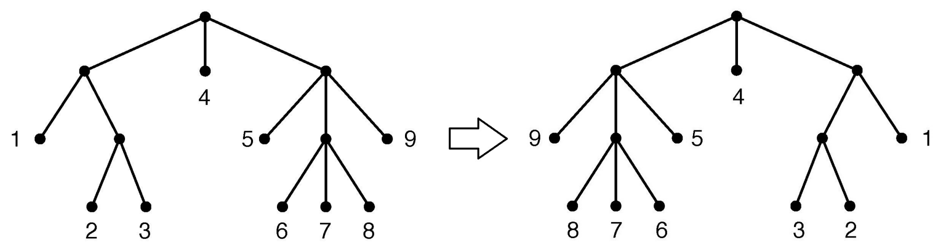
(b) Write the `nfilter` function.

```
;; nfilter: (X -> Bool) Nested-Listof-X -> Nested-Listof-X

(nfilter odd? '(1 (2 3) () ((4)))) => '(1 (3) () (()))
```

(c) Write the `nmap` function.

```
;; nmap: (X -> Y) Nested-Listof-X -> Nested-Listof-Y

(nmap sqr '(1 (2 3) () ((4)))) => '(1 (4 9) () ((16)))
```

(d) Write the `nreverse` function, which reverses every nested list. (note: you may not use `reverse` to solve this problem, but append will likely be necessary). We have shown how the nested list from Slide 77 in the notes would be reversed:




```
(nreverse '(1 (2 3) () ((4)))) => '(((4)) () (3 2) 1)
(nreverse '((1 (2 3)) 4 (5 (6 7 8) 9))) => '((9 (8 7 6) 5) 4 ((3 2) 1)) ;; Slide 77
```

(e) Write the `nheight` function, which determines the height of a nested list. The height of a regular (non-nested) list is 1, regardless if it is empty or not. A list that contains lists (such as a simple association list) would have a height of 2. The height corresponds to the maximum number of lists that have been nested. The nested list on Slide 77 (also shown above) has a height of 3. 找高度（空的 list 高度也是 1）

```
(nheight '()) => 1
(nheight '(a b c)) => 1
(nheight '((1 a) (2 b) (3 c))) => 2
(nheight '(1 (2 3) () ((4)))) => 3
(nheight '((1 (2 3)) 4 (5 (6 7 8) 9))) => 3 ;; Slide 77
```

(f) On Slide 77, we show how *leaf-labelled trees* can be represented by nested lists. While every leaf-labelled tree is a nested list, not every nested list is a leaf-labelled tree. For example, the nested list `'(1 (2 3) () ((4)))` contains an `empty` list, which is a "leaf" with no label. Furthermore, when `nfilter` is applied to a leaf-labelled tree, the result may no longer be a leaf-labelled tree. Write the `prune` function which removes all empty lists and any nested lists that only contain empty lists (i.e., they do not contain any elements of type `X`). Note that if there are no elements of type `X`, `prune` produces `empty`. 就是去掉所有空的 sublist

```
(prune '(1 (2 3 ()) ( (()) (4) ()(())))) => '(1 (2 3) ((4)))
(prune '(()((())())))) => '()
```

4. You are probably familiar with "spell checkers" that not only find misspelled words, but also *suggest* correctly spelled words. In this question, we will develop a function that suggests words. We use the following data definition of a `Word`:

```
;; A Word is a Str
;; requires: only lowercase letters appear in the word
;;           (no spaces, punctuation, etc.)
```

全小写的string，不包含标点符号

We have provided you with the file `suggest.rkt`. This file contains a partial solution that you may build upon. We strongly recommend you begin with this file, but if you want an extra challenge you may ignore this file and develop your solution independently.

(a) Write the function `remove-dups` that consumes a list of `Word`s that are sorted in non-decreasing order. `remove-dups` removes any duplicate `Word`s so that each `Word` in the produced list is unique (maintaining the original order).

去掉所有重复出现的strings

```
(remove-dups '("apple" "apple" "apples" "banana" "cherry" "cherry"))
 => '("apple" "apples" "banana" "cherry")
```

abstract function

(b) We desire a new higher-ordered function that abstracts lockstep recursion on both a list and a natural number (using a "countup" pattern). In other words, we wish to recurse on a list, but we also want to keep track of the *index* (or *"position"*) of the element we are processing. The very first element has an index of `0`, the second has an index of `1`, etc.

Write an *indexed* version of `foldr` named `ifoldr` that behaves very similar to `foldr`, except that it uses a combine function that has an extra argument, corresponding to the current index of the element being processed. For comparison, consider the contracts and sample applications for both `foldr` and `ifoldr`:

```
;; foldr: (X Y -> Y) Y (listof X) -> Y
;; (foldr f base '(a b c)) => (f a (f b (f c base)))
```

这个是普通的foldr

```
;; ifoldr: (Nat X Y -> Y) Y (listof X) -> Y
;; (ifoldr g base '(a b c)) => (g 0 a (g 1 b (g 2 c base)))
```

这个是让你写的ifoldr，就是consume的function里面多了一个Nat代表当前element的位置

The following example shows how `ifoldr` can be used:

0 1 2

```
(ifoldr (lambda (i x y) (cons (list i x) y)) empty '(a b c))
=> '((0 a) (1 b) (2 c))
```

这道题肯定要直接recursion

You may use explicit recursion (pure structural) to `define` your `ifoldr` function. As a challenge you can write `ifoldr` using built-in Abstract List Functions, but you are not required to do so – make sure you review the allowed functions and their usage in the Piazza Post (FAQ).

(c) The `suggest` function consumes a `Word` and a predicate function for determining if a `Word` is spelled correctly. The function produces a list of correctly spelled words that are "close" to the consumed word (we define "close" shortly). The predicate function consumes a `Word` and produces a `Boolean`.

suggest已经给你写好了，你要写的是suggest的四个helper functions

其实这页基本的废话，你要写的四个 helpers 都在下一页

```
;; suggest: Word (Word -> Bool) -> (listof Word)
```

For example, consider the following application of `suggest` that consumes a function `valid?` that determines if a string appears in a list of strings.

```
(define (valid? s)
  (member? s '("rights" "right" "fight" "aardvark" "fhqwhgads" "bright")))

(suggest "right" valid?) => '("bright" "fight" "rights")
```

A few things to note from this example:

- `suggest` does not have access to a list of correctly spelled words. `suggest` is only provided a predicate function (e.g., `valid?`) that determines if a word is correctly spelled.
- Instead, `suggest` generates many possible words that are close to the word (more on that later) and then `filter`s out any of the generated words that are not correctly spelled (according to the provided predicate function).
- In this example, `suggest` did not produce the word `"aardvark"` as it is not close to `"right"`.
- In this example, `suggest` could not produce the word `"light"`, because `(valid? "light")` produces `false`.
- The word list contained `"fhqwhgads"`, which is not a correctly spelled word (it does not appear in *most* dictionaries). It doesn't matter if a word is *actually* correctly spelled, only that the predicate function produces `true` (you may want to use misspelled words in your tests).
- It does not matter if the provided word is correctly spelled.

The list of words produced by `suggest` must be sorted in ascending order, may not contain any duplicates and must not contain the original word (if it is spelled correctly) or the empty string (`""`).

The `suggest` function generates every `Word` that has exactly one "change", as defined by the list below. In other words, `suggest` generates all `Word`s with an "Edit Distance" of one (including transpositions/swaps). If you wish, you can read more about Edit Distances online (but you are not required to do so).

For each change, we have shown what words would be generated for the word `"jklm"`.

- **single deletions:** each letter removed from the word.
  `"klm", "jlm", "jkm", "jkl"`
- **single insertions:** every possible letter (a..z) inserted before each letter in the word.
  `"ajklm", "bjklm", "cjklm", ... "zjklm", "jaklm", "jbklm", ...`
  `... "jzklm", "jkalm", ... "jkzlm", ... ... "jklym", "jklzm"`
- **trailing letter:** every possible letter inserted at the end of the word.
  `"jklma", "jklmb", "jklmc", ... "jklmz"`

```
> (replace-letters "ass")
(list
 (list "ass" "aas" "asa")
 (list "bss" "abs" "asb")
 (list "css" "acs" "asc")
 (list "dss" "ads" "asd")
 (list "ess" "aes" "ase")
 (list "fss" "afs" "asf")
 (list "gss" "ags" "asg")
 (list "hss" "ahs" "ash")
 (list "iss" "ais" "asi")
 (list "jss" "ajs" "asj")
 (list "kss" "aks" "ask")
 (list "lss" "als" "asl")
 (list "mss" "ams" "asm")
 (list "nss" "ans" "asn")
 (list "oss" "aos" "aso")
 (list "pss" "aps" "asp")
 (list "qss" "aqs" "asq")
 (list "rss" "ars" "asr")
 (list "sss" "ass" "ass")
 (list "tss" "ats" "ast")
 (list "uss" "aus" "asu")
 (list "vss" "avs" "asv")
 (list "wss" "aws" "asw")
 (list "xss" "axs" "asx")
 (list "yss" "ays" "asy")
 (list "zss" "azs" "asz"))
```

replace

```
> (trailing-letters "ass")
(list
 "assa"
 "assb"
 "assc"
 "assd"
 "asse"
 "assf"
 "assg"
 "assh"
 "assi"
 "assj"
 "assk"
 "assl"
 "assm"
 "assn"
 "asso"
 "assp"
 "assq"
 "assr"
 "asss"
 "asst"
 "assu"
 "assv"
 "assw"
 "assx"
 "assy"
 "assz")
```

trailing

```
> (insert-letters "ass")
(list
 (list "aass" "aass" "asas")
 (list "bass" "abss" "asbs")
 (list "cass" "acss" "ascs")
 (list "dass" "adss" "asds")
 (list "eass" "aess" "ases")
 (list "fass" "afss" "asfs")
 (list "gass" "agss" "asgs")
 (list "hass" "ahss" "ashs")
 (list "iass" "aiss" "asis")
 (list "jass" "ajss" "asjs")
 (list "kass" "akss" "asks")
 (list "lass" "alss" "asls")
 (list "mass" "amss" "asms")
 (list "nass" "anss" "asns")
 (list "oass" "aoss" "asos")
 (list "pass" "apss" "asps")
 (list "qass" "aqss" "asqs")
 (list "rass" "arss" "asrs")
 (list "sass" "asss" "asss")
 (list "tass" "atss" "asts")
 (list "uass" "auss" "asus")
 (list "vass" "avss" "asvs")
 (list "wass" "awss" "asws")
 (list "xass" "axss" "asxs")
 (list "yass" "ayss" "asys")
 (list "zass" "azss" "aszs"))
```

insert

swap

```
> (swap-letters "fuckyou")
(list "ufckyou" "fcukyou" "fukcyou" "fucykou" "fuckoyu" "fuckyuo")
```

- **single substitutions:** each letter replaced with every possible letter.
  `"aklm", "bklm", "cklm", ... "zklm", "jalm", "jblm", ...`
  `... "jzlm", "jkam", ... "jkzm", ... ... "jkly", "jklz"`
- **adjacent swaps (transpositions):** each adjacent pair of letters in the word swapped.
  `"kjlm", "jlkm", "jkml"`

We have provided an implementation for the `suggest` function in the file `suggest.rkt`. It is complete, but it relies on `remove-dups` (see above) which you are required to write and five helper functions to generate each of the possible changes above: `remove-letters, insert-letters, trailing-letters, replace-letters, swap-letters`.

We have also implemented one of the five helper functions (`remove-letters`) for you as an example. You are required to write the remaining four. If you do not complete all four, you will receive partial marks for the functions you did complete.

To implement `remove-letters` we first wrote a helper function `remove-at` which removes an element with index `i` from a list. `remove-at` depends on the `ifoldr` function you are required to write (see above). `remove-letters` then uses `build-list` to remove each letter. It is strongly recommended that you understand `remove-at` and `remove-letters` before attempting the remaining helper functions. The two functions will also be reviewed at the Tutorial this week.

You are not required to use our implementation of `suggest` or any of the helper functions it relies on. We will only test your `remove-dups`, `ifoldr` and `suggest` functions.

You may want to generate multiple predicate functions (similar to `valid?`) that have different word lists. The `make-validator` function from Question 2 above will be helpful. A predicate function that always produces `true` such as `(lambda (s) true)` (that thinks that all words are spelled correctly) may also be helpful.

After the deadline for A07, we will provide a file `spellcheck.rkt` that contains a (modified) Trie with almost 40,000 english words and provides a `spellcheck?` function that searches that Trie. You may use that file (add `(require "spellcheck.rkt")` at the top of your file) for your own experiments (and to show off your program to your friends & family) but you should not leave the `require` in your submitted file or use that file for your examples and tests. You are not required to use the `spellcheck.rkt` file and if your computer struggles with the large Trie then it is best to simply avoid it.

5. **8% Bonus**: Place your solution in the file `omit.rkt`. You do not need to include the design recipe for any of these bonus questions. However, you must provide a sentence or two that briefly explains your approach and demonstrates that you understand your code and that you are submitting your own work.

   (a) Write the Racket function `omit1`, which consumes a string and produces a list of all possible strings with all possible letters omitted. For example, `(omit1 "abc")` produces something like `(list "abc""ab""bc""ac""a""b""c""")`. The order of strings in the list may vary, but for a string of length n it must produce exactly 2^n strings. If it helps, you can assume the input string does not contain any duplicate characters. Write the function any way you want. (Value: 1%)

(b) Now write the Racket function `omit2`, which behaves exactly like `omit1` but which does not use any explicit recursion or helper functions. You must rely on abstract list functions and `lambda` (and potentially standard list functions like `string->list`, `cons`, `first`, `rest`, `append`, etc.). Your solution must not be more than three lines of code. Note that if you solve this question, you can also use it as a solution to the previous one—just have `omit1` apply `omit2`. (Value: 2%)

(c) For the ultimate challenge, write the Racket function `omit3`. Do not write any helper functions, and do not use any explicit recursion (i.e., your function cannot call itself by name). Do not use any abstract list functions. In fact, use only the following list of Racket functions, constants and special forms: `string->list`, `list->string`, `cons`, `first`, `rest`, `empty?`, `empty`, `lambda`, and `cond` (including `else`). You are permitted to use **define** exactly once, to define the function itself. (Value: 5%)

This concludes the list of questions for which you need to submit solutions. Don't forget to always check your email for the public test results after making a submission.

Enhancements: *Reminder—enhancements are for your interest and are not to be handed in.*

Professor Temple does not trust the built-in functions in Racket. In fact, Professor Temple does not trust constants, either. Here is the grammar for the programs Professor Temple trusts.

$\langle \text{exp} \rangle = \langle \text{var} \rangle |(\textbf{lambda } (\langle \text{var} \rangle)\ \langle \text{exp} \rangle\ )\ |\ (\langle \text{exp} \rangle \langle \text{exp} \rangle)$

Although Professor Temple does not trust **define**, we can use it ourselves as a shorthand for describing particular expressions constructed using this grammar.

It doesn't look as if Professor Temple believes in functions with more than one argument, but in fact Professor Temple is fine with this concept; it's just expressed in a different way. We can create a function with two arguments in the above grammar by creating a function which consumes the first argument and returns a function which, when applied to the second argument, returns the answer we want (this should be familiar from the *make-adder* example from class, Section 10). This generalizes to multiple arguments.

But what can Professor Temple do without constants? Quite a lot, actually. To start with, here is Professor Temple's definition of zero. It is the function which ignores its argument and returns the identity function.

$$(\textbf{define}\ \textit{my-zero}\ (\textbf{lambda}\ (f)\ (\textbf{lambda}\ (x)\ x)))$$

Another way of describing this representation of zero is that it is the function which takes a function f as its argument and returns a function which applies f to its argument zero times. Then "one" would be the function which takes a function f as its argument and returns a function which applies f to its argument once.

$$(\textbf{define}\ \mathit{my\text{-}one}\ (\textbf{lambda}\ (f)\ (\textbf{lambda}\ (x)\ (f\ x))))$$

Work out the definition of "two". How might Professor Temple define the function *add1*? Show that your definition of *add1* applied to the above representation of zero yields one, and applied to one yields two. Can you give a definition of the function which performs addition on its two arguments in this representation? What about multiplication?

Now we see that Professor Temple's representation can handle natural numbers. Can Professor Temple handle Boolean values? Sure. Here are Professor Temple's definitions of true and false.

$$(\textbf{define}\ \mathit{my\text{-}true}\ (\textbf{lambda}\ (x)\ (\textbf{lambda}\ (y)\ x)))$$
$$(\textbf{define}\ \mathit{my\text{-}false}\ (\textbf{lambda}\ (x)\ (\textbf{lambda}\ (y)\ y)))$$

Show that the expression $((c\ a)\ b)$, where c is one of the values *my-true* or *my-false* defined above, evaluates to a and b, respectively. Use this idea to define the functions *my-and*, *my-or*, and *my-not*.

What about *my-cons*, *my-first*, and *my-rest*? We can define the value of *my-cons* to be the function which, when applied to *my-true*, returns the first argument *my-cons* was called with, and when applied to the argument *my-false*, returns the second. Give precise definitions of *my-cons*, *my-first*, and *my-rest*, and verify that they satisfy the algebraic equations that the regular Scheme versions do. What should *my-empty* be?

The function *my-sub1* is quite tricky. What we need to do is create the pair $(0,0)$ by using *my-cons*. Then we consider the operation on such a pair of taking the "rest" and making it the "first", and making the "rest" be the old "rest" plus one (which we know how to do). So the tuple $(0,0)$ becomes $(0,1)$, then $(1,2)$, and so on. If we repeat this operation n times, we get $(n-1,n)$. We can then pick out the "first" of this tuple to be $n-1$. Since our representation of n has something to do with repeating things n times, this gives us a way of defining *my-sub1*. Make this more precise, and then figure out *my-zero?*.

If we don't have **define**, how can we do recursion, which we use in just about every function involving lists and many involving natural numbers? It is still possible, but this is beyond even the scope of this challenge; it involves a very ingenious (and difficult to understand) construction called the Y combinator. You can read more about it at the following URL (PostScript document):

http://www.ccs.neu.edu/home/matthias/BTLS/tls-sample.ps

Be warned that this is truly mindbending.

Professor Temple has been possessed by the spirit of Alonzo Church (1903–1995), who used this idea to define a model of computation based on the definition of functions and nothing else. This is called the lambda calculus, and he used it in 1936 to show a function which was definable but not computable (whether two lambda calculus expressions define the same function). Alan Turing later gave a simpler proof which we discussed in the enhancement to Assignment 7. The lambda calculus was the inspiration for LISP, a predecessor of Racket, and is the reason that the teaching languages retain the keyword **lambda** for use in defining anonymous functions.