CS 135 Fall 2015
Akinyemi, Heinle, Holtby, Istead, Roegiest, Tompkins

# Assignment: 5

| | |
|---|---|
| Due: | Tuesday, October 20th, 9:00 pm |
| Language level: | Beginning Student with list abbreviations |
| Allowed recursion: | Pure structural recursion |
| Files to submit: | `prime.rkt`, `store.rkt`, `applicants.rkt` |
| Warmup exercises: | HtDP 10.1.4, 10.1.5, 11.2.1, 11.2.2, 11.4.3, 11.5.1, 11.5.2, 11.5.3 |
| Practise exercises: | HtDP 10.1.6, 10.1.8, 10.2.4, 10.2.6, 10.2.9, 11.4.5, 11.4.7 |

- You should note a heading at the top of the assignment: "Allowed recursion." In this case the heading restricts you to **pure structural recursion**. That is, the recursion that follows the data definition of the data it consumes, and parameters of the recursive calls are either unchanged or one step closer to the base case.
- Provide your own examples that are distinct from the examples given in the question description of each function.
- Of the built-in list functions, you may only use *cons*, *first*, *second*, *rest*, *empty?*, *cons?*, *(list ...)*, *member?*, and *append*.
- Do **not** use quoted lists for this assignment. Use *(list...)* instead.
- Your solutions must be entirely your own work.
- For this and all subsequent assignments, you should include the design recipe for all functions, including helper functions, as discussed in class.
- Solutions will be marked for both correctness and good style as outlined in the Style Guide.
- When naming the list-of-X parameters in your function contracts, follow the *(listof X)* notation that is given on Slide 34 of Module 05 (e.g., include a list of numbers as *(listof Num)*).
- You must use the **cond** special form, and are not allowed to use **if** in any of your solutions.
- It is very important that the function names and number of parameters match ours. You must use the basic tests to be sure. The names of the functions must be written exactly. The names of the parameters are up to you, but should be meaningful. The order and meaning of the parameters are carefully specified in each problem.

Here are the assignment questions you need to submit.

1. This question involves recursion on the natural numbers.

   (a) Write a function *prime?* that consumes a positive natural number and produces true if that number is prime, and false otherwise. A prime number is a number divisible by only 1 and itself, and 1 is not a prime. Put another way, a number is prime if the largest factor of that number (other than itself) is 1.

   You may find it useful to observe that natural number *b* is a factor of natural number *a* if and only if (*remainder a b*) is 0 (zero).

(b) Write a function *next-prime* that consumes a natural number and produces the next prime strictly greater than that number. The consumed number can be any natural number. That is, it does not need to be prime itself. Examples:
(*next-prime* 7) => 11
(*next-prime* 8) => 11

Note: This function requires a slight change from the count-up template because it does not know what it is counting up to until it gets there. This is OK.

(c) Write a function *prime-range* that consumes two natural numbers and produces the list of all prime numbers in the interval that starts with the first number and ends with the second number (inclusive). In other words, if given the natural numbers $a$ and $b$ (in that order) it produces all primes $p$ such that $a \leq p \leq b$. The list produced is in ascending order (the first value is the smallest value).
Examples:
(*prime-range* 1 10) => (*list* 2 3 5 7)
(*prime-range* 10 1) => *empty*

Place you solutions in the file `prime.rkt`

2. In this question we will be dealing with the CS135 store, where students can buy CS135-themed items such as delicious doughnuts, BraveRats playing cards, and Rocketdyne F-1 engines. We will use the following data definitions

```
(define-struct product (name price taxable?))
;; A Product is a (make-product Sym Num Bool)
;; requires: price > 0,
;; price cannot have fractional cents.
```

Taxable items are charged 13% sales tax. "Fractional cents" would mean that the number of cents is not a whole number, e.g. 3.01 is an allowed price, while 3.005 is not. For all functions that consume a list of products, you can assume there are no duplicate product names.

(a) Write a function *have-product?* that consumes a symbol and a list of products, and produces true if the symbol is the name of one of the products in the list. The function produces false otherwise.

(b) Write a function *valid-order?* that consumes a list of symbols and a list of products, and produces true if all symbols in the list of symbols are the names of items in the list of products, and false otherwise. An empty order is always valid.

(c) Write a function *budget-items* that consumes a list of products and a price limit (a number), and produces the list of products with a final price that is less than or equal to the price limit. In other words, (*budget-items my-products* 100) removes all products in the list that are greater than $100 **after tax**. For example with a price limit of $100.00 one could buy a tax-free $100.00 item, but could not buy a taxable $88.50 item (after

taxes this item is $100.005). A final price of $100.0001 is greater than $100 (i.e. do not round the prices).

(d) Write a function *total-order* that consumes a list of symbols and a list of products, and produces the final cost (after tax) of buying all of the items in the list of symbols. The price must be **rounded down** (i.e. *floor*) to a whole cent value (e.g. if the total cost is $99.999 the function will produce 99.99. Make sure you do not round until the final step! If the list of symbols contains a product name not listed in the list of products, the *total-order* function cannot compute the total, so it produces the symbol 'cannot-fulfill instead.

Hint: To round something (down) to the cents instead of to dollars, use (/ (*floor* (∗ *value* 100)) 100).

Place your solutions in the file `store.rkt`

3. Large companies usually have many applicants to choose from, and many different positions they are hiring for. One way to assist the decision making process is with math! An employer decides on *N* skills they are interested in, such as "coding", "program design", "being able to follow a style guide", and "people skills". The employer then assesses these skills in the applicants through a series of tests (or they get the applicants to self-assess). An applicant's skill ranks are a list of *N* natural numbers, one per skill. A 0 would mean the employee has none of that particular skill. If the employer was using the 4 skills above, a skill ranking might look like (*list* 9 6 1 1), they're a strong coder, an acceptable designer, but not great at following guidelines or interacting with people.

   The employer would also assign *N* weights to each position. A weight of 0 would mean that position does not require that skill at all, and a high weight means that skill is very important. For the position "Style Enforcer" the employer might have weights (*list* 2 0 9 6). They need to at least understand how to code, but don't need to know how to design. The most important skill is being able to follow a style guide, and they also need people skills so they provide more helpful feedback than "your code is bad and you should feel bad".

   An applicant's suitability score for a given position is the sum of their skills, weighted by the corresponding weights for that position. Using the above two lists, the suitability score is $9 \cdot 2 + 6 \cdot 0 + 1 \cdot 9 + 1 \cdot 6 = 33$.

   These scores can be used to pick the most suitable positions for each applicant, and/or the most suitable applicants for each position.

   Because it is easier to talk about a position by name, rather than as a list of numbers, a position is represented using a structure:

   (*define-struct pprof* (*title wts*))
   ;; A position profile (PProf) is a (make-pprof Sym (listof Nat)).

   Note: *N* could be 0, in which case all lists are empty and all applicant scores are 0.

(a) Write a function *applicant-score* which consumes the applicant's skill ranking list and the employer's position profile, and produces the applicant's score for that position. The list of skill rankings must contain the same number of elements as the profile's *wts* field.

(b) An employer may have several positions that match an applicant's skill. In this situation, the employer may wish to match the applicant with the most suitable position. We can start with the case where there are only two positions to choose from.

Write a function *position-max* which consumes an applicant's skill rankings (a list of natural numbers) and two position profiles. The function produces the profile that has the greatest *applicant-score* for the given skill rankings. If both profiles yield the same score, the function picks the first profile over the second.

(c) Now we can use *position-max* to find the most suitable position out of a list of any number of position profiles.

Write a function *position-list-max* which consumes the applicant's skill rankings (a list of natural numbers), and a non-empty list of position profiles. The function will produce the position that the applicant is most suited for. If there is a tie, the function goes with the first profile it encountered with this score (the one earliest in the list of profiles).

(d) As positions get filled, the employer will have to remove them from the list.

Write a function *remove-position* which consumes a position title (a symbol) and a list of position profiles, and produces a new list where the profile with that name has been removed. If no position had that name, the list is unchanged.

Example:
(*remove-position* 'manager (*list* (*make-pprof* 'coder (*list* 2 0)) (*make-pprof* 'manager (*list* 1 2)))) => (*list* (*make-pprof* 'coder (*list* 2 0)))

Place your solutions in the file `applicants.rkt`

This concludes the list of questions for which you need to submit solutions. Do not forget to always check your email for the basic test results after making a submission.

---

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

There is a strong connection between recursion and induction. Mathematical induction is the proof technique often used to prove the correctness of programs that use recursion; the structure of the induction parallels the structure of the function. As an example, consider the following function, which computes the sum of the first *n* natural numbers.

```
(define (sum-first n)
  (cond
    [(zero? n) 0]
    [else (+ n (sum-first (sub1 n)))]))
```

To prove this program correct, we need to show that, for all natural numbers $n$, the result of evaluating *(sum-first n)* is $\sum_{i=0}^{n} i$. We prove this by induction on $n$.

**Base case:** $n = 0$. When $n = 0$, we can use the semantics of Racket to evaluate *(sum-first 0)* as follows:

(*sum-first* 0)
*yields* (**cond** [(*zero?* 0) 0][**else** ...])
*yields* (**cond** [*true* 0][**else** ...])
*yields* 0

Since $0 = \sum_{i=0}^{0} i$, we have proved the base case.

**Inductive step:** Given $n > 0$, we assume that the program is correct for the input $n-1$, that is, *(sum-first (sub1 n))* evaluates to $\sum_{i=0}^{n-1} i$. The evaluation of *(sum-first n)* proceeds as follows:

(*sum-first n*)
*yields* (**cond** [(*zero? n*) 0][**else** ...]) ;(we know $n > 0$)
*yields* (**cond** [*false* 0][**else** ...])
*yields* (**cond** [**else** (+ *n* (*sum-first* (*sub1 n*)))])
*yields* (+ *n* (*sum-first* (*sub1 n*)))

Now we use the inductive hypothesis to assert that *(sum-first (sub1 n))* evaluates to $s = \sum_{i=0}^{n-1} i$. Then *(+ n s)* evaluates to $n + \sum_{i=0}^{n-1} i$, or $\sum_{i=0}^{n} i$, as required. This completes the proof by induction.

Use a similar proof to show that, for all natural numbers $n$, *(sum-first n)* evaluates to $(n^2 + n)/2$.

**Note:** Summing the first $n$ natural numbers in imperative languages such as C++ or Java would be done using a `for` or `while` loop. But proving such a loop correct, even such a simple loop, is considerably more complicated, because typically some variable is accumulating the sum, and its value keeps changing. Thus the induction needs to be done over time, or number of statements executed, or number of iterations of the loop, and it is messier because the semantic model in these languages is so far-removed from the language itself. Special temporal logics have been developed to deal with the problem of proving larger imperative programs correct.

The general problem of being confident, whether through a mathematical proof or some other formal process, that the specification of a program matches its implementation is of great importance in *safety-critical* software, where the consequences of a mismatch might be quite severe (for instance, when it occurs with software to control an airplane, or a nuclear power plant).