CS 135
Winter 2014
Troy Vasiga

# Assignment: 5

| | |
|---|---|
| Due: | Tuesday, February 25, 2014 at 9:00pm |
| Language level: | Beginning Student with List Abbreviations |
| Allowed recursion: | Pure structural |
| Files to submit: | `a05.rkt`, `a05bonus.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 |

The US Social Security administration publishes the 1,000 most popular male baby names and the 1,000 most popular female baby names for each decade since 1890. See `http://www.socialsecurity.gov/OACT/babynames/` for more information.

You may search for specific names and see graphs of that name's popularity through time at `http://www.student.cs.uwaterloo.ca/~cs135/assns/a05/nameSurfer/`. Use it to explore! For example, the name "Rock" has a spike right about the time Rock Hudson was a popular movie star. "Abdullah" appears very late in the American data. "Kelsey" shows a meteoric rise in popularity in the 1980's data. "Adolf" (as in Hitler) was already a dead name by 1920. One might find it surprising that "Adolph", practically the same name, did **not** take a plunge in popularity right after World War II.

The goal of this assignment is to produce the "guts" of a similar application.

Data on each name is recorded using the *Nameinfo* data type, defined as (*define-struct nameinfo* (*name decade rank gender*)) where

- *name* is a *String*
- *decade* is a *Nat* that is one of 1890, 1900, 1910, ... 2000. It indicates the decade for which the information applies
- *rank* is an *Int* in [1, 1000] indicating the popularity of *name* in the given *decade* where 1 is the most popular name for the decade and 1,000 is the least popular name
- *gender* is one of {'Male, 'Female} and indicates the gender of the babies given the name.

The *nameinfo* structure has already been defined for you and used to create two lists. The first, *name-list*, contains all 24,000 data points from the Social Security Administration data. The second, *short-name-list*, contains only the names beginning with "A" (2,057 data points). You may find it useful for experimenting. To access these definitions in DrRacket, you need to do the following:

1. Download the file `names.txt` from the course website. Save it in the same directory where your a05 Scheme programs will be saved.

2. Download the file `namelist.rkt` from the course website. Save it in the same directory as `names.txt`. **Do not cut and paste the `namelist.rkt` code into a blank file in DrRacket. Download it, perhaps using right-click then "Save As".**

3. In your Scheme program, include the line (*require* "namelist.rkt") at the beginning of your program. For example, save the following in `a05-demo.rkt`. It should display the first two names in the list.

   ```
   (require "namelist.rkt")

   (first name-list)
   (first (rest name-list))
   ```

You will **not** submit `namelist.rkt` nor `names.txt`. We already have copies; you won't be changing yours.

All of the required functions will be submitted in a single file, `a05.rkt`, because they build on each other for the last several problems. A syntax error in one will make all of them unmarkable. For this reason **you must use the public test to ensure your code can be tested.** Do not change your submitted code without reviewing the public test to verify that it (still) works correctly.

Here are the assignment questions you need to submit.

1. Write a full data definition and template for a *Namelist*, a list of *Nameinfo* structures.

2. Write the function *find-rank*. It consumes a *Namelist*, a name, one of {'Male, 'Female}, and a decade. It produces the name's popularity ranking for the given decade and gender. Produce *false* if the ranking can't be found.

3. Write the function *collect-name*. It consumes a *Namelist*, a name, and a gender. It produces a *Namelist* composed of all the *Nameinfo*s in the given list that match the given name and gender. The order of the produced list should be in the same order as they appear in the consumed list.

4. Write the function *first-n*. It consumes a list and a natural number, $n$, and produces the first $n$ items from the list. If there are fewer than $n$ items on the list produce 'NotEnoughItems. You should be able to do this without counting the number of items ahead of time. That is, do not use *length*.

5. Review the *sort* function on slide 06-29. It uses the *insert* function on slide 06-31. Adapt these two functions to sort a *Namelist* in increasing order by name (you will find the predicate *String<?* useful). If the names are the same, place females before males. If the name and gender are the same, sort in increasing order by decade. Call the functions *name-sort* and

*name-insert*. **Warning:** if you try sorting the entire *name-list* you will be waiting for a long time! Note that we will test both your *name-insert* and *name-sort* functions.

6. Write the function *bar-graph*. It consumes a list of natural numbers and returns a string such that each number $n$ in the list results in a sequence of $n$ "*" characters. Each such 'bar' should be separated from the next by a newline character ("\n"). For example, (*bar-graph* (*list* 5 10 0 3)) should produce "*****\n**********\n\n***\n".

   Notes for *bar-graph*:

   (a) You may not use string functions, in particular *make-string* or *string-append*. You may not use *append*, either (we haven't discussed it in class).

   (b) An asterisk is represented in scheme by #\\$*$ and the newline character is represented by #\newline.

   (c) Two functions you may find useful, especially in your testing, are *list→string* and *string→list*. The first function converts a list of characters into a string; the second converts a string into a list of characters. For example, both of the following tests pass:
   (*check-expect* (*list→string* (*list* #\\$*$ #\\$*$ #\\$*$)) "$***$")
   (*check-expect* (*list* #\\$*$ #\\$*$ #\\$*$) (*string→list* "$***$"))

   (d) Assuming you restrict yourself to pure structural recursion (as specified by the assignment header), your solution will look like (**define** (*bar-graph lon*) (*list→string* (*helper lon*))). For your helper function, consider "one step closer to the base case" of the list (*cons* 3 (*cons* 4 *empty*)) to be (*cons* 2 (*cons* 4 *empty*)) rather than (*cons* 4 *empty*).

   (e) The `namelist.rkt` module you *require* at the beginning of your code provides a *display* function. It will display your bar-graph in a visually pleasing fashion. *display* is not to be used in *bar-graph*. Use it to display the result of *bar-graph*.

7. Write *name-popularity-graph*. It consumes a *Namelist*, a name, and a gender, and produces a bar graph (as outlined in the previous part) of that name's popularity for each decade between 1890 and 2000. The length of each bar will be $67 - \lfloor \frac{rank}{15} \rfloor$. That is, top-ranking names will have long bars and low-ranking names will have short bars.

   Names that aren't ranked for a particular decade should have blank bars. There is a function in `namelist.rkt` that will insert dummy *Nameinfo* records into a *Namelist* sorted by decade where there are missing records. The contract is *insert-missing-decades: Nat Nat Namelist -> Namelist*. It takes the first decade to insert, the last decade, and a sorted *Namelist*.

An example of the result of (*display* (*name-popularity-graph name-list* "Kimberly" 'Female)) (Kimberly was not a popular name in the first part of the last century and thus has blank bars):

```
> (display (name-popularity-graph name-list "Kimberly" 'Female))





****************
*************************************************************
*******************************************************************
*******************************************************************
******************************************************************
*****************************************************************
****************************************************************
>
```

Recall that each row of stars must end with \n.

This concludes the list of questions for which you need to submit solutions. Don't forget to always review your public test after making a submission. Failing any portion of that test almost guarantees losing marks.

---

**Hints:**

- For testing purposes, define one or more short *Namelist*s containing 2-5 *Nameinfo*s. Don't test with *name-info*!
- Helper functions are your friends.

---

8. **5% Bonus**: Make a copy of `a05.rkt` named `a05bonus.rkt`. Modify it so that your popularity graph has each bar labeled with the appropriate decade. For example:

```
> (display (name-popularity-graph name-list "Kimberly" 'Female))
1890
1900
1910
1920
1930
1940 ****************
1950 *************************************************************
```

```
1960 *******************************************************************
1970 *******************************************************************
1980 ******************************************************************
1990 *****************************************************************
2000 ****************************************************************
>
```

You may use *number->string*. The restrictions on Q6 do not apply to the bonus. Notice that we expect exactly one space after the year, even if there are no stars on that line. In other words, the string above should begin ``` ``1890 \n1900 \n1910...''```.

---

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

There is a strong connection between recursion and induction. The proof technique of mathematical induction is what is used to prove programs using recursion correct, and 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 Scheme 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).