Comp 210
Lab 2
Scheme Examples, and Using Donkey

1 Data Description → Templates → Functions

The recipe for writing functions presented in class is simple: describe the data precisely, make a template function which handles any data of this kind, and then fill in the template based on the particular task of the task at hand. The shape of a function follows from the shape of the data is uses!

1.1 Data Description

Write down a precise definition of what type of data you'll be using. Actually deciding how to represent the data is the first phase of writing a program, before you ever write a big of the code. Along with deciding exactly how to decompose large programs into manageable chunks, this phase is perhaps the most creative and the most critical to getting a good program.

As an example, consider credentials. A person either is self-taught, or they went to a school and had an academic advisor; our value on the credential will want to include the advisor's credential. One data description (out of many possibilities) might be:

A credential is either

- *name*, where *name* is a symbol, or
- (list name school advisor-credential), where *name* and *school* are symbols, and *advisor-credential* is a credential.

Of course, to work with these data objects, we need to have a good idea of what they look like, so let's give a few concrete examples: 'edison and (*list* 'alexGreat 'tutored (*list* 'aristotle 'stoaU (*list* 'plato 'acadamy 'socrates))).

1.2 Template

A template is what you can start writing about a function knowing only what type of data it processes, but nothing about what the particular function does. Create the template in three steps:

1. Case Analysis (use **cond**), if any. For every different possible form of the data (from your data description), you'll have a corresponding choice about how to handle it. If there are no cases—the data is always of one predictable form—then you won't have any **cond** at all.

   For the case of credentials, the two cases are whether the person is self-taught or not. As mentioned in lecture, we can use the Scheme predicate *cons?* to ask if somthing is a constructed list. (Alternatively we can ask *symbol?* to identify symbols.) Thus given a credential *cr*, are two **cond** conditions are *symbol? cr*)} **and** {*scheme* cr) and **else**cr) (where we can get by with just one of these conditions and *car*).

2. Data Extraction (use *cdr*, *cr*, etc.) Within each case, extract the useful individual data from any lists they happen to reside in. Just mention each possibly-important datum separated by ellipses ("...").

   For a credential (*car cr*) which is a symbol, there is no data to extract. In the other case, (*car* (*cdr cr*)) is the name field, (*car* (*cdr* (*cdr cr*))) is the school field, and (*fun-for-creds* (*car* (*cdr* (*cdr cr*)))) is the credential.

3. Natural Recursion: note how it follows from the natural recursion in the data description! If the data has no natural recursion, then your template won't either.

   In our case, the function would recursively be called on the credential inside the credential: *count-advisors*.

Putting this all together for our example, we have a template for credentials:

```
(define fun-for-creds
  (lambda (cr)
    (cond ((symbol? cr)  ...cr...)
          (else       ...(car cr)...(car (cdr cr))...
                           ...(fun-for-creds (car (cdr (cdr cr))))))))
```

1.3 Function

Once you have the template, you need to process the different chunks of data you have in each case. Naturally, the details of doing this depend entirely on what the function is supposed to do.

You may not always use every possible piece of information from the Data Extraction step; that's okay.

For instance, let's make a particular function, *name*, which returns how many "generations" somebody is from a self-taught mentor. We fill in the blanks of the template, which isn't too hard. If we're in the first condition, of a self-taught mentor, then they have 0 advisors. In the other case, we realize that the name and school fields don't matter. The number of advisors is exactly the number of ther advisor's advisors, plus one.

```
(define count-advisors
  (lambda (cr)
     (cond ((symbol? cr)  0)
           (else          (+ 1 (fun-for-creds (car (cdr (cdr cr)))))))))
```

Try it out!

2 Family Tree

With your lab leader, work through the following: How would you give a data structure for a family tree, that is, a list of peoples parent's names, and their parent's parent's names, and ...?

A family tree is either:

- (*list mom dad name*), a symbol, or
- *name*, where *mom* is a symbol, and *dad* and *length* are family trees.

```
(define fun-for-families
   (lambda (t)
      (cond ((symbol? t) ...)
            (else    ...(car (cdr (cdr t)))
                     ...(fun-for-families (car t))...
                     ...(fun-for-families (car (cdr t))...)))))
```

Given a familty tree, how many known people are in it (assuming everybody in the tree is distinct)? Well, if the tree is just a symbol, then there is only 1. The other possibility is that with parents, there is 1 (the individually named person), plus however many people are in each parent's part of the tree. (Note that the parents themselves get counted in their own section of the tree.)

We follow the template to get

```
(define fun-for-families
   (lambda (t)
      (cond ((symbol? t) 1)
            (else    (+ 1 (fun-for-families (car t))
                          (fun-for-families (car (cdr t))))))))
```

Bonus exercise: write a function that, given a family tree, returns how many ancestors a person has *with the same name*. Hint: first write a function that takes two arguments: a name and a family tree, and returns how many time that name occurs in the tree.

3 Hand-evaluation

In lecture we've been talking about the rules of evaluation. You should be able to look at the programs we're writing, and determine what they'll do. You need only repeatedly use the rule of "replace an expression A with an equal expression B" (being careful to note which parentheses are part of A, and which aren't).

For example, consider the function (*length* (*list* 'a 'b 'c 'd)), to compute the length of a list:

```
(define length
  (lambda (lst)
    (cond [(null? lst) 0]
          [else       (+ 1 (length (cdr lst)))])))
```

Now, we hand evaluate *product*, showing only the recursive function calls:

```
(length (list 'a 'b 'c))
=
(+ 1 (length (list 'b 'c)))
=
(+ 1 (+ 1 (length (list 'c))))
=
(+ 1 (+ 1 (+ 1 (length null)))
=
(+ 1 (+ 1 (+ 1 0)))
=
(+ 1 (+ 1 1))
=
```

```
(+ 1 2)
=
3
```

Ta-dum! Try writing (*product* `4 5 3`), a function which takes a list of numbers and returns their product. (What should you do if the list is empty?!) Then, show the recursive applications calls made when you evaluate (*length* (*list* 'a 'b 'c 'd)).

4 The UNIX Corner: the Filesystem

The handout for Lab 1 gave an introduction to the UNIX file system. We'll build on that here by mentioning a few more commands in more detail.

- `ls` Some useful flags to `ls` are `-a`, which lists *all* files. Normally, files whose name begins with the character "`.`" are ignored by `ls` as uninteresting.

  The `-l` flag (as mentioned in the Lab 1 handout) gives the *long* listing of files, including the size, owner's name, and permission privileges for that file. The meaning of those permission codes, and a more full explanation of how to use `chmod` to change them, will be talked about in lab, and is also covered in Owlnet's UNIX handout distributed in Lab 1.

- `mkdir, rmdir` As mentioned in the Lab 1 handout, `mkdir` command *makes* a new *directory*. You can remove a directory with `rmdir`, but you must remove all the directory's files first (with `rm`). Of course, you can make directories within directories; for instance after changing directories to `~/comp210`, you can then `mkdir hw2` and `mkdir hw3` to further categorize your files.

- `pwd` To find out where you currently are in the filesystem, `pwd` will *print* your *working directory*.

- `cp, mv` These commands *copy* and *move* files, respectively. Note that moving a file is the same as renaming it; there isn't a separate rename command.

  Both `cp` and `mv` share a similar syntax; we use `cp` to illustrate, but the below holds for `mv` as well.

You can copy one file to another by saying `cp oneFile another`. (Try making a copy of your homework file, naming the copy `test.ss`.) If the file `another` already exists, it will be overwritten, and lost forever. (One special case is that if `another` already exists but is a directory, then the directory is *not* overwritten, and instead a copy of `oneFile` is put *inside* `another/`.) You can also copy many files all to the same directory at once: `cp file1 file2 file3 file4 targetDir`.

The `-i` flag ("interactive"?) provides some protection for you; it causes `cp` or `mv` to ask before deleting a file. You may have already seen this behavior; type `which cp` for a clue as to why!

In order to copy an entire directory to a new name, you must use the `-r` flag to *recursively* copy all the files inside that directory (including any of those files that are themselves directories). (This does not apply to `mv`.)

- `rm` To *remove* files, use `rm`, but with caution: once a file is deleted, it is gone from the disk! You might be able to get the file backed up from tape, but only if the file has been around for a while. Fortunately, your `rm` is initially set up to use the `-i` flag, which acts as it does for `cp` and `mv`. The `-r` (recursive) flag works for `rm` as well, but be very careful–`rm -r ~` will speedily delete *all* your files!

- `man` If you've forgetten exactly which arcane flag to (say) `ls` causes it to indicate directories by appending a slash to the name, you can always look it up in the *manual*: try `man ls`. (The answer is `-F`.) The on-line manual pages are famous for not being particularly helpful in learning about UNIX overall, and it has more obscure details than any sane person would want to know, but it *is* handy to remind yourself of what flags a program takes, and what they do.

  Will `man man` cause an infinite loop?

- `alias` You can use `alias` to give a new (shorter) name to another existing command. For instance, I like `alias ls ls -F`. For the rest of the time I am logged in, whenever I type (say) `ls ~comp210`, the command that really gets issued is `ls -F ~comp210`. To cancel an alias you've made, use `unalias`: For instance, `unalias ls`.

  This concept may not seem to have too much potential, but some idiots go beserk over it. For instance, check out the file `ian/.aliases`.

For aliases that you like to use all the time, you can actually put that above alias command at the end of your file `~/.cshrc`. This is a special file whose commands are executed every time you log in. (In fact, it is through this file that a system-wide file is executed for you; this system-wide file contains the command `alias cp cp -i`. Aha!)

Self-exercises:

1. What is the absolute pathname for comp210? For your home directory?

2. From your home directory, what is the relative pathname of the comp210 directory? Use `ls` to verify your guess.

3. List the files of the comp210's homework directory, find what the file is named which has the hw2 solutions, and copy it to a file named `my-copy.ss` in your `comp210` directory.

4. Find exactly the line in the system-wide file which aliases `cp` for you. If you were to decide you didn't want this behaviour of `cp`, how would you nullify it? (I don't recommend actually doing this, but it's empowering to know how.)