CSE 390P: Problem Solving with Programming — Winter 2018

Day 1: Getting Familiar With Scheme

We will be using the site http://repl.it to run Scheme code. You and your team should use a *single computer* and work together.

Getting Familiar With Scheme

(a) Scheme uses *prefix notation* instead of the standard *infix notation*. For example, to call a procedure `foo` with arguments `argh1` and `argh1`, we would write `(foo argh1 argh2)` instead of the more standard `foo(argh1, argh2)`.

(b) In Scheme, we use the `define` procedure to define a variable and the `print` procedure to print strings.

```
(define hello "hello world")
(print "We want to print " hello)

> We want to print hello world
```

Notice the lack of commas between arguments to `print`.

(c) We define procedures in a very similar way:

```
(define (print-greeting greeting)
    (print greeting " World")
)
(print-greeting "Hello")
(print-greeting "Ahoy")

> Hello World
> Ahoy World
```

(d) In Scheme, the last value in order will be automatically returned. For example, here is a modification of the previous procedure to *return* the greeting instead of printing it, we simply remove the print:

```
(define (make-greeting greeting)
    (string-append greeting " World" "!")
)
(make-greeting "Hello")
(make-greeting "Ahoy")

=> "Ahoy World!"
```

(e) **Task 1.1**

Define a scheme procedure `prefix-to-infix` which takes in arguments op `arg1` and `arg2` in prefix notation and returns a String representing the infix notation of the arguments. Make sure your result has spaces. For example:

```
(print (prefix-to-infix "+" 1 2))
(print (prefix-to-infix "-" 3 4))

> 1 + 2
> 3 - 4
```

Hint: To convert a number to a string, use the procedure `number->string`.

(f) **Reading 1.2**

Read section 1.1.6 of SICP which can be found at the URL: http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html#%_sec_1.1.6.

Lists, Recursion, and Interviews

(a) In Scheme, the primary data structure used to work with data is the linked list. The following are examples of manipulations of linked lists:

```
; constructor
(define mylist (list 1 2 3))

; get the first element in the list
(print (car mylist))

; get the "rest" of the list
(print (cdr mylist))

; get the length of the list
(print (length mylist) ", " (length (cdr mylist)))

; add an element to the front of a list
(print (cons 0 mylist))

> 1
> (2 3)
> 3, 2
> (0 1 2 3)
```

(b) You may have noticed that `print`, `cond`, `+`, and `string-append` all take an arbitrary number of arguments. In Scheme, the way to specify an unknown number of arguments is to use ".". The argument after the "." will be a list that contains all the remaining arguments. For example:

```
(define (delimit-arguments delim . rest)
  (cond
    ((= 0 (length rest)) ())
    ((= 1 (length rest)) (list (car rest)))
    (else (cons
      (car rest)
      (cons delim
      (apply delimit-arguments (cons delim (cdr rest))))
    ))
  )
)
(print (delimit-arguments "hi" 1 2 3))

> (1 hi 2 hi 3)
```

N.B. `(apply fun args)` is a procedure that calls `fun` using the elements of `args` as the arguments to `fun`. I strongly recommend that you play around with this procedure before moving on.

(c) **Task 2.1** Write a procedure `prefix-to-infix-2` which generalizes `prefix-to-infix` to any number of arguments.

```
(print (prefix-to-infix-2 "+" 1 2 3 4 5))
(print (prefix-to-infix-2 "-" 3 4))
```

```
> 1 + 2 + 3 + 4 + 5
> 3 - 4
```

(d) **Task 2.2** Write a procedure `count-change` which returns the number of ways to make change for the first argument using the rest of the arguments as denominations.

```
(print (count-change 3 1 2 3))
(print (count-change 30 1))
(print (count-change 30 4))

> 3
> 1
> 0
```

(e) **Task 2.3** Write a procedure `pascal` which returns the nth row of pascal's triangle, where 1 is the 0th row, $(1\ 1)$ is the 1st row, etc.

```
(print (pascal 0))
(print (pascal 1))
(print (pascal 2))
(print (pascal 3))
(print (pascal 4))
(print (pascal 5))
(print (pascal 10))

> (1)
> (1 1)
> (1 2 1)
> (1 3 3 1)
> (1 4 6 4 1)
> (1 5 10 10 5 1)
> (1 10 45 120 210 252 210 120 45 10 1)
```

Number Theory

(a) **Reading 3.1**

Read sections 1.2.4 and 1.2.5 of SICP which can be found at the URL: http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html#%_sec_1.2.4.

(b) **Task 3.2**

Implement the Miller-Rabin Primality Test in Scheme: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test.