CS 61A
Summer 2017

Scheme
Mentoring 11: July 26, 2017

1 Scheme

Visit scheme.cs61a.org to try the online interpreter. Type `(demo 'autopair)` and the interpreter will automatically draw box-and-pointer diagrams whenever an expression evalutes to a Scheme pair.

1.1 What Would Scheme Display?

(a)
```
scm> 3.14
```

(b)
```
scm> pi
```

(c)
```
scm> (define pi 3.14)
```

(d)
```
scm> pi
```

(e)
```
scm> 'pi
```

(f)
```
scm> (if 2 3 4)
```

(g)
```
scm> (if 0 3 4)
```

(h)
```
scm> (if #f 3 4)
```

(i)
```
scm> (if nil 3 4)
```

(j)
```
scm> (if (= 1 1) 'hello 'goodbye)
```

(k)
```
scm> (define (factorial n)
        (if (= n 0)
            1
            (* n (factorial (- n 1)) )))
```

(l)
```
scm> (factorial 5)
```

(m)
```
scm> (= 2 3)
```

(n)
```
scm> (= '() '())
```

(o)
```
scm> (eq? '() '())
```

(p)
```
scm> (eq? nil nil)
```

(q)
```
scm> (eq? '() nil)
```

(r)
```
scm> (pair? (cons 1 2))
```

(s)
```
scm> (list? (cons 1 2))
```

1.2 **Hailstone yet again** Define a program called `hailstone`, which takes in two numbers `seed` and `n`, and returns the `n`*th* hailstone number in the sequence starting at `seed`. Assume the hailstone sequence starting at `seed` is longer or equal to `n`. As a reminder, to get the next number in the sequence, if the number is even, divide by two. Else, multiply by 3 and add 1.

Useful procedures

- `quotient`: floor divides, much like `//` in python

  `(quotient 103 10)` outputs 10

- `remainder`: takes two numbers and computes the remainder of dividing the first number by the second

  `(remainder 103 10)` outputs 3

```
; The hailstone sequence starting at seed = 10 would be
; 10 => 5 => 16 => 8 => 4 => 2 => 1

; Doctests
> (hailstone 10 0)
10
> (hailstone 10 1)
5
> (hailstone 10 2)
16
> (hailstone 5 1)
16
> (hailstone 5 5)
1
```

Scheme lists are similar to the linked lists we've seen already in Python.

```
Link(1, Link.empty)                    (cons 1 nil)
a = Link(1, Link(2, Link.empty))       (define a (cons 1 (cons 2 nil)))
a.first                                (car a)
a.rest                                 (cdr a)
```

1.3 What Would Scheme Display? Draw box-and-pointer diagrams!

(a) `scm> (cons 1 2)`

(b) `scm> (cons 1 (cons 2 nil))`

(c) `scm> (cons 1 '(2 3 4 5))`

(d) `scm> (cons 1 '(2 (cons 3 4))`

(e) `scm> (cons 1 (2 (cons 3 4)))`

(f) `scm> (define a '(1 2 . 3))`

(g) `scm> a`

(h) `scm> (car a)`

(i) `scm> (cdr a)`

(j) `scm> (cadr a)`

(k) How can we get the 3 out of a?

1.4 Define `well-formed`, which determines whether `lst` is a well-formed list or not. Assume that `lst` only contains numbers.

```
; Doctests
> (well-formed '())
true
> (well-formed '(1 2 3))
true
; List doesn't end in nil
> (well-formed (cons 1 2))
false
; You do NOT need to check nested lists
> (well-formed (cons (cons 1 2) nil))
true
```

1.5 Define `is-prefix`, which takes in a list `p` and a list `lst` and determines if `p` is a prefix of `lst`.

```
; Doctests:
> (is-prefix '() '())
true
> (is-prefix '() '(1 2))
true
> (is-prefix '(1) '(1 2))
true
> (is-prefix '(2) '(1 2))
false
; Note here p is longer than lst
> (is-prefix '(1 2) '(1))
false
```