SCHEME

COMPUTER SCIENCE MENTORS 61A

March 20 to March 24, 2017

Scheme is a programming language, much like Python. In fact, many of Python's design features were inspired by Scheme. The point of learning this language is twofold: one, we're looking into what parts of Python generalize to other languages. Two, we want to start thinking about how to design and build (an interpreter for) a programming language, and it turns out Scheme is a nice one to build. In fact, we'll show you enough of the language in this hour to write recursive procedures. This section covers the basics. You'll learn the rest in lab and discussion. It's pretty awesome that we'll be picking up a whole new programming language within an hour.

Useful Resources:

- CS61A Online Scheme Interpreter: `scheme.cs61a.org`
  - To see the box and pointer diagrams for lists, do `(demo 'autopair)`

1 What Would Scheme Print?

1. What will Scheme output? Draw box-and-pointer diagrams to help determine this.

```
scm> 3.14
```

Solution: `3.14`

```
scm> pi
```

Solution: `Error`

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

Solution: `pi`

```
scm> pi
```

Solution: `3.14`

```
scm> 'pi
```

Solution: `pi`

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

Solution: `3`

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

Solution: `3`

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

Solution: `4`

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

Solution: `3`

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

Solution: `hello`

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

Solution: `factorial`

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

Solution: `120`

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

Solution: `#f`

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

Solution: `Error`

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

Solution: `#t`

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

Solution: `#t`

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

Solution: `#t`

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

Solution: `#t`

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

Solution: `#f`

2 Code Writing in Scheme

2. **Hailstone once 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 to know:

1. `quotient`: floor divides, much like `//` in python

   `(quotient 103 10)` outputs 10

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

   `(remainder 103 10)` outputs 3

Solution:

```
(define (hailstone seed n)
    (if (= n 0)
        seed
        (if (= 0 (remainder seed 2))
            (hailstone
            (quotient seed 2)
            (- n 1)
           )
          (hailstone
          (+ 1 (* seed 3))
          (- n 1)
         )
    )
)
```

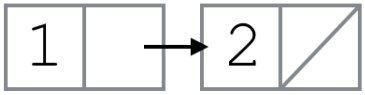
3 Scheme Lists

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

Solution: `(1 . 2)`


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

Solution: `(1 2)`

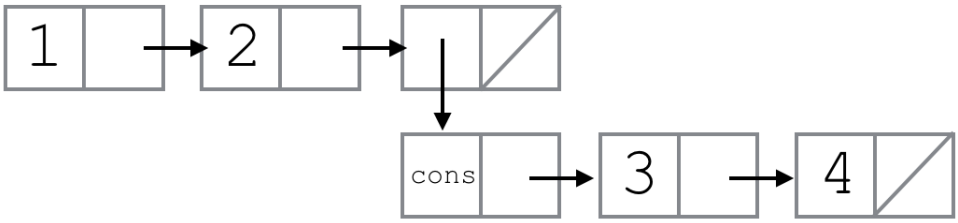



```
scm> (cons 1 '(2 3 4 5))
```

Solution: `(1 2 3 4 5)`


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

Solution: `(1 2 (cons 3 4))`




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

Solution:

```
eval: bad function in : (2 (cons 3 4))
```

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

Solution:
```
a
```

```
scm> a
```

Solution:
```
(1 2 . 3)
```

```
scm> (car a)
```

Solution:
```
1
```

```
scm> (cdr a)
```

Solution:
```
(2 . 3)
```

```
scm> (cadr a)
```

Solution:
```
2
```

How can we get the 3 out of a?

Solution:
```
(cddr a)
```

4 More Code Writing in Scheme

3. 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
```

Solution:

```
; well-formed with a nested if statement
(define (well-formed lst)
    (if (null? lst)
        #t
        (if (number? lst)
            #f
            (well-formed (cdr lst)))))

; well-form with a cond statement
(define (well-formed lst)
    (cond ((null? lst) #t)
        ((number? lst) #f)
        (else (well-formed (cdr lst)))))
```

4. 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
```

Solution:

```
; Same as below, but with cond
(define (is-prefix p lst)
    (cond ((null? p) #t)
        ((null? lst) #f)
        (else (and (= (car p) (car lst))
            (is-prefix (cdr p) (cdr lst))))))

; Solution that checks if lst is null for the last doctest
(define (is-prefix p lst)
    (if (null? p)
        #t
        (if (null? lst)
            #f
            (and
                (= (car p) (car lst))
                (is-prefix (cdr p) (cdr lst))))))
```