SCHEME, INTERPRETERS

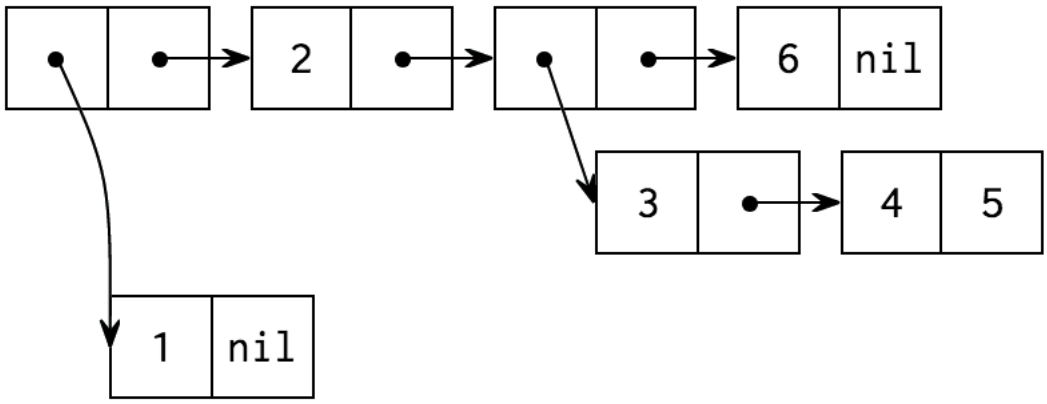
CS Scholars

April 4 and April 6, 2017

1 Introduction

1. What will Scheme output? Draw box-and-pointer diagrams where appropriate.

   (a) `(cons (cons 1 nil) (cons 2 (cons (cons 3 (cons 4 5)) (cons 6 nil))))`



   (b) `(define a 4)`

   a

   `((lambda (x y) (+ a)) 1 2)`

   4

   (c) `((lambda (x y z) (y x)) 2 / 2)`

   0.5

   (d) `((lambda (x) (x x)) (lambda (y) 4))`

   4

(e) `(define boom1 (/ 1 0))`

4

(f) `boom1`

4

(g) `(define boom2 (lambda () (/ 1 0)))`

4

(h) `(boom2)`

4

(i) Why/How are the two boom definitions above different? 4

(j) How can we rewrite boom2 without using the lambda operator? 4

2 Scheme Procedures

1. Write a procedure `blastoff` that takes in a number `n` and returns a list of all numbers from `n` and 1 followed by `BLASTOFF!`.

```
> (countdown 10)
(10 9 8 7 6 5 4 3 2 1 BLASTOFF!)

> (countdown 3)
(3 2 1 BLASTOFF!)

(define (countdown n)









)
(define (countdown n) (if (= n 0) (cons  blastoff! nil) (
   cons n (countdown (- n 1)))))
)
```

2. Write `before-in-list`, which takes a list, `lst` and two elements `a` and `b`. It should return `#t` if `a` appears in `lst` before `b`. Check the doctests for more details. Hint: Recall `contains?`.

```
> (before-in-list '(1 2 3) 1 3)
#t
> (before-in-list '(1 2 3) 3 1)
#f
> (before-in-list '(1 2 3) 1 4)
#f
> (before-in-list '(1 2 3) 0 3)
#f

(define (before-in-list lst a b)
```

```
)
(define (before-in-list lst a b) (cond ((null? lst) #f) ((
   equal? (car lst) a) (contains? b (cdr lst))) ((equal? (car
   lst) b) #f) (else (before-in-list (cdr lst) a b)) ) )
```

3. Describe the result of calling the following procedure with a list as its argument. What would

```
(mystery '(1 2 3))s
```

return?

```
(define (mystery lst)
   (mystery-helper lst '()))

(define (mystery-helper lst other)
   (if (null? lst)
      other
      (mystery-helper (cdr lst) (cons (car lst) other))))
```

It will reverse whatever is passed in. (3 2 1)

4. Write `wheres-waldo`, a Scheme procedure which takes in a scheme list and outputs the index of `waldo` if the symbol `waldo` exists in the list. Otherwise, it outputs the symbol `nowhere`.

```
> (wheres-waldo '(moe larry waldo curly))
2
> (wheres-waldo '(1 2))
nowhere

(define (wheres-waldo lst)
   (cond
      ((null? lst) ____________________)
      ((equal? ____________________________________________)
      (else
         (let ((found-him ____________________________________))
            (if (equal? ____________________________________)
               ____________________
               (+ 1 ____________________________________)
            )
```

```
            )
        )
    )
)
```

```
(define (wheres-waldo lst)
    (cond
        ((null? lst) 'nowhere)
        ((equal? (car lst) 'waldo) 0)
        (else
            (let ((found-him (wheres-waldo (cdr lst))))
                (if (equal? 'nowhere found-him)
                    found-him
                    (+ 1 found-him)
                )
            )
        )
    )
)
```

5. **To Binary**

Write a procedure that takes in a number `n` and returns a binary representation of `n`

```
> (to-binary 2)
(0 1 0)

> (to-binary 7)
(0 1 1 1)
```

Note: Here is an approach to finding the binary representation of a number.

1. What is the value of `n` % 2? Take note of this number.
2. Let `n` = `n` // 2
3. Repeat steps 1 and 2 until `n` becomes 0.
4. Reverse the order of the remainders you took note of in step 1.

Example:

$$\begin{aligned} n &= 9 \\ 9\%2 &= 1 \\ 4\%2 &= 0 \\ 2\%2 &= 0 \\ 1\%2 &= 1 \\ 0\%2 &= 0 \end{aligned}$$

So the binary representation of 9 is: 01001

```
(define (to-binary n)
```

Solution: (define (to-binary n) (if (= n 0) (cons 0 nil) (append (to-binary (floor (/ n 2))) (cons (remainder n 2) nil))))

3 Interpreters

3.1 Eval and Apply

There are three types of expressions in Scheme you should know:
(1) symbols
(2) atoms
(3) call expressions

Each type follows a set of rules. Lets take a look!

1. How many calls to `scheme_eval` are required for the following expressions? `scheme_apply`?

```
> 42
1, 0
> (define life 42)              #ignore this line
> life
1, 0
> (define (meaning life) 42)    #ignore this line (for now >:D
   )
> meaning
1, 0
> (meaning 0)
4, 1
> (+ 21 21)
4, 1
> (* 21 2)
4, 1
> (meaning (- 42 42))
7, 2
```

3.2 Special Forms

What makes special forms so special? List out as many special forms as you can. How are special forms evaluated in Scheme?

Special form	Rules for evaluation
`begin`	evaluate all expressions
`and`	evaluate expressions until one evaluates to a
`or`	evaluate expressions until one evaluates to a
`cond`	evaluate predicate expressions until one eval value, then evaluate the corresponding expre
`if`	evaluate the predicate, then evaluate the 2nd predicate is truth-y or the 3rd expression if th
`let`	evaluate expressions in bindings, then evalua body
`define`	no evaluation
`lambda`	no evaluation
`quote`	no evaluation

2. Same as above, but with special forms! How many calls must you make to `scheme_eval` and `scheme_apply`?

```
> (define life 42)
1, 0
> (define (meaning life) 42)
1, 0
> (let ((live +)) ((lambda (life) (live life)) 42))
8, 2
> (define (meaning) eq?)
1, 0
> meaning
1, 0
```

```
> (meaning)
3, 1
> (begin ((or #f #f (meaning) #f (/ 1 0)) 3 3))
10, 2
> (let ((legend #f) (of (+)) (zelda (* 3 3)) (if legend of
   zelda))
11, 2
```