SCHEME AND INTERPRETERS

COMPUTER SCIENCE 61A

July 31 to August 6, 2015

1 Scheme

1. What would Scheme print? Also, draw the box-and-pointer diagram for each of the following linked lists:

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

Solution: `(1 . 2)`

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

Solution: `Error`

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

Solution: `(1 2)`

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

Solution: `(1 cons 2)`

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

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

```
scm> (1 . 2 3)
```

Solution: `Error`

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

Solution: `a`

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

Solution: `1`

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

Solution: `2`

How can we access 3 from `a`? How can we access (3 . 4) from `a`?

Solution:
```
(car (cdr (cdr a)))
(cdr (cdr a))
```

```
scm> (define x 5)
```

Solution: `x`

```
scm> ((lambda (c d) (+ x)) 1 2)
```

Solution: `5`

```
scm> ((lambda (x y z) (y x)) 2 / 2)
```

Solution: `0.5`

```
scm> (define boom1 (/ 1 0))
```

Solution: `Error: Zero Division`

```
scm> boom1
```

Solution: `Error: boom1 not defined`

```
scm> (define boom2 (lambda () (/ 1 0)))
```

Solution:

```
scm> (boom2)
```

Solution: `Error: Zero Division`

Why are `boom1` and `boom2` (defined above) different?

Solution: The first line sets `boom1` equal to the value `(/ 1 0)`, which throws a `ZeroDivisionError`. On the other hand, `boom2` is equal to a lambda function that takes in no arguments that, when called, evaluates `(/ 1 0)` and throws a `ZeroDivisionError`.

How can we rewrite `boom2` without using the `lambda` operator?

Solution: `(define (boom2) (/ 1 0))`

2. Implement `contains`, which takes in a val and a well-formed list (lst). Returns `True` if the val is in lst or any of the lists within the lst, and `False` otherwise.

```
scm> (define a (cons 1 (cons 2 (cons 3 nil))))
a
scm> (contains 3 a)
True
scm> (contains 6 a)
False
scm> (define b  '(1 3 (9 8) 2))
b
scm> (contains 8 b)
True
scm> (contains 2 b)
True
scm> (contains 7 b)
False
```

```
(define (contains val lst)
```

Solution:

```
    (cond ((null? lst) False)
          ((list? (car lst)) (or (contains val (car lst))
                                 (contains val (cdr lst))))
          ((= (car lst) val) True)
          (else (contains val (cdr lst)))
    )
)
```

2 Interpreters

An interpreter is a computer program that **reads** your input, **evaluates** it, and **prints** the result. For each interpreter, there is an underlying language and an implemented language. The interpreter *understands* the implemented language and is *written* in the underlying language. For this question, we will be implementing a few functions using python for a very simple calculator language that behaves as follows:

```
> 1
1
> (+ 1 3)
4
```

The first step to implementing a language is **tokenizing**, which converts the input string to a list of tokens.

```
>>> tokens = tokenize("(+ (- 3 1) 4)")
>>> tokens
['(', '+', '(', '-', 3, 1, ')', 4, ')']
```

The next step is to **parse** the tokens and create a deep linked list that will expose the structure of the parentheses, so we can apply the operator to the operands in the correct order. We do this using a function called `calc_read`. `calc_read` takes in a list of tokens, and returns a linked list that represents the calculator expression.

```
>>> exp = calc_read(tokens)
>>> exp
Link('+', Link(Link('-', Link(3, Link(1))), Link(4)))
```

After parsing the input, we can **evaluate** the expression using a function called `calc_eval`. `calc_eval` evaluates the return value of `calc_read` by using `calc_apply` (defined below) to apply the operator (the first argument of the linked list returned by `calc_read`) to the operands (the rest of the linked list returned by `calc_read`). Your job is to implement `calc_eval`.

1. In order implement `calc_eval`, first implement `map_linked_list`, a function which takes in a function and a linked list, and applies the function to every element of the list.

```
def map_linked_list(f, lst):
    """Returns a list of the results produced by applying f to
    each element in lst.

    >>> my_list = Link(1, Link(2, Link(3, Link(4, empty))))
    >>> map_linked_list(lambda x: x * x, my_list)
    Link(1, Link(4, Link(9, Link(16))))
    """
```

Solution:

```
    if lst == Link.empty:
        return Link.empty
    return Link(f(lst.first), map_linked_list(f, lst.rest))
```

2. Now use `map_linked_list` and `calc_apply` in order to implement `calc_eval`. If the expression is a number (a primitive), then the expression just evaluates to itself. Otherwise, if it's a Link, the expression is a call expression.

```
def calc_eval(exp):
    """Evaluates a calculator expression.

    >>> calc_eval(5)
    5
    >>> calc_eval(link('+', link(12, link(3, empty))))
    15
    >>> subexp1 = link('*', link(3, link(4, empty)))
    >>> subexp2 = link('-', link(12, link(9, empty)))
    >>> exp = link('+', link(subexp1, link(subexp2, empty)))
    >>> print_linked_list(exp)
    < '+' < '*' 3 4 > < '-' 12 9 > >
    >>> calc_eval(exp)
    15
    """
```

Solution:

```
    if type(exp) in (int, float):
        return exp
    elif isinstance(exp, Link):
        operator = expr.first
        args = map_linked_list(calc_eval, expr.second)
        return calc_apply(operator, args)
```

3. Now, modify `calc_apply` so that it can handle the operator **, which is the power function. Assume a function `do_power` is defined for you. *Hint: The change is extremely similar to the implementation of * - + and /.*

```
def calc_apply(op, args):
    """Applies an operator to a linked list of arguments

    >>> calc_apply(   +   , Link(1, Link(3, Link(5))))
    9
    >>> calc_apply(   **   , Link(4, Link(3)))
    64
    """
    if op == '+':
        return do_addition(args)
    elif op == '*':
        return do_multiplication(args)
    elif op == '-':
        return do_subtraction(args)
    elif op == '/':
        return do_division(args)
```

Solution:

```
    elif op == '**':
        return do_power(args)
```