# CS 240 Project 2

Alex Vondrak

Fall 2011

## Overview

In Project 1, we wrote a *REPL* (read eval print loop) for a stack-based language. In this project, we will develop another REPL for a language based on linked lists—essentially, building a simple Lisp interpreter.

The "read" part of the REPL has been taken care of for us by Homework 3, which showed code that could parse a parenthetical syntax for linked lists. The "print" and "loop" parts are straightforward (they don't really change much from Project 1), so we'll just focus on interesting things we can do with "eval".

To that end, your task is to complete the partial implementation of a Lisp REPL given to you at `http://www.csupomona.edu/~ajvondrak/cs/240/11/fall/lecture/prj2.java`.

### Eval

Lisp expressions (`Exprs`) are composed of either `Atoms` or `Lists`. For our purposes, `Atoms` represent either integers or the standard arithmetic operators. `Lists` represent the application of an operator to zero or more operands (using *prefix* notation), thus also eventually evaluating to an integer. The basic syntax is

```
( operator operand_expr1 operand_expr2 ... )
```

For instance,

- the `Atom 2` evaluates to the `int 2`.
- the `Atom -2` evaluates to the `int -2`.
- the `Atom 15` evaluates to the `int 15`.
- the `List ( + 2 2 )` evaluates to `4`.
- the `List ( * 2 5 )` evaluates to `10`.
- the `List ( * ( + 1 2 ) 5 )` evaluates to `15`.

More examples can be seen in the Examples section at the end of the project description.

Since we eventually get an integer out of any expression, we add an `eval` method to the `Expr` interface from Homework 3. Its job is to return the `int` that the `Expr` represents, as in the below excerpt from `prj2.java`.

```
class EvalException extends Exception {
      public EvalException(String msg) { super(msg); }
}

interface Expr {
   public int eval() throws EvalException;
}
```

In `prj2.java`, you'll see several method "stubs" with `IMPLEMENT ME` comments inside of them. It is your job to do what those methods say to do:

- You must provide an `eval` method for the `Atom` class, in the case that the `Atom` is not some operator at the beginning of a `List`.
- The `List<E>` class also has an `eval` method, which must use the provided helper method (`evalArgs`) to first evaluate each operand of the list, then pass the results to the proper method based on the (presumed) `Atom` at the front of the list—either `add` for `"+"`, `sub` for `"-"`, `mul` for `"*"`, or `div` for `"/"`.

**Note:** The descriptions of the `add`, `mul`, `sub`, and `div` methods give you *specific* ways to implement them—ways that give you practice with different forms of recursion and iteration. Pay attention to the instructions in `prj2.java`.

## Examples

Here is a sample interaction with the Lisp REPL. Lines beginning with `lisp>` are what the user inputs at the prompt.

```
$ java Lisp
lisp> ( )
EvalException: Empty list.
lisp> 1 2 3
1
2
3
lisp> ( + ) ( + 1 ) ( + 1 2 ) ( + 1 2 3 )
0
1
3
6
lisp> ( - ) ( - 1 ) ( - 1 2 ) ( - 1 2 3 )
0
-1
-1
-4
lisp> ( * ) ( * 1 ) ( * 1 2 ) ( * 1 2 3 )
1
1
2
6
lisp> ( / ) ( / 20 ) ( / 20 10 ) ( / 20 10 2 )
EvalException: / expects at least 2 arguments
EvalException: / expects at least 2 arguments
2
1
lisp> ( - 96 23 )
73
lisp> ( - 96 20 1 1 1 )
73
lisp> ( / 200 10 )
20
lisp> ( / 200 5 2 )
20
lisp> ( / 200 ( * 5 2 ) )
20
lisp> ( + 8 ( * 5 ( - 10 ) ) )
-42
lisp> +
EvalException: Could not eval atom to an int: +
lisp> ( + a b )
EvalException: Could not eval atom to an int: a
lisp> (
Parsing error.
lisp> )
Parsing error.
lisp> Won’t you take me to funky town?
EvalException: Could not eval atom to an int: Won’t
EvalException: Could not eval atom to an int: you
EvalException: Could not eval atom to an int: take
EvalException: Could not eval atom to an int: me
EvalException: Could not eval atom to an int: to
EvalException: Could not eval atom to an int: funky
EvalException: Could not eval atom to an int: town?
lisp> ( 1 2 3 )
EvalException: Unknown oeprator: 1
lisp> ( + ( * 100 6 ) ( * 10 6 ) ( * 1 6 ) )
666
lisp> ( / 1 2 )
0
lisp> ( / 2 1 )
2
lisp>
```