# 6.MITx: Day 1 PM Project

This morning we started on an expression calculator and got as far as producing an array of tokens (strings representing numbers, operators, parentheses) from the text entered by the user. Now let's complete the implementation by writing code to evaluate the expression from left to right, computing the answer as we go.

First we'll need a `read_operand` function that attempts to interpret the first token in the array as a number. Here's a pseudo code description of what needs to be done:

```
read_operand(array of tokens) {
    num = first element of the array
    remove the first element using the .shift() method of array
    use built-in function parseInt() to convert num to an integer
    if result is the value NaN ("not a number"), throw an error "number expected"
    otherwise return the integer value
}
```

`throw` in Javascript is very much like `raise` in Python: it returns up the call tree until it reaches an enclosing `try` -- Google it to read more. Take a moment test your implementation using the Javascript console:

```
tokens = ['123','42','hi','25'];
    read_operand(tokens);   // call this 4 times: did you get the expected
results?
```

Now write the `evaluate` function that processes the token array element-by-element:

```
evaluation (array of tokens) {
   if array is empty, throw an error "missing operand"
   value = read_operand(array of tokens)
   while (array is not empty) {
      operator = next token, remove from array
      if operator isn't +, -, * or /, throw an error "unrecognized operator"
      if array is empty, throw an error "missing operand"
      temp = read_operand(array of tokens)
      value = value "operator" temp   // perform requested operation
   }
   return value
}
```

We'll test this code by calling it from inside of `calculate` (the function we wrote this morning) right after we have built the array of tokens:

```
try {
   val = evaluate(tokens)
  if tokens isn't empty, throw an error "ill-formed expression"
   return String(val)
} catch (err) {
   return err;   // error message will be printed as the answer
}
```

Try examples using your web page: enter expression, click calculate and see what get's printed. Test to make sure all the different error conditions are reported correctly.

Once this is working, spend your remaining time tackling the following extensions:

1. add code to deal with parenthesized subexpressions. Hint: if `read_operand` finds a "(" token, it should make a recursive call to `evaluate` and return the resulting value as the operand. Before returning, the code should check that the next token is ")" and remove it from the array, throwing an error if ")" isn't found.
2. add code to deal with negation. Hint: `read_operand` will have to check for a "-" token, then do the right thing!
3. change the token pattern to match floating point numbers instead of simply integers -- Google for "regex floating point". Note that you don't want to recognize a leading "-" as part of the number otherwise the text "3 - 4" will be tokenized as ["3","-4"] instead of ["3","-","4"].
4. add code to support operator precedence, i.e., the operators * and / take precedence over + and -. For example the expression "3 + 1/5" should evaluate to 3.2 not 0.8! Hint: change `evaluate` to only handle additions and subtractions, having it call a new function `read_term` that handles continuous sequences of multiplications and divisions (it stops when it runs out of tokens or see a "+" or "-" operator). `read_term` is now the function that calls `read_operand`.
5. Add support for built-in one-argument Math functions such as sin, log, sqrt, …

Good luck! Call us over and give a quick demo as you get the various pieces working...