CSE1729 - Introduction to Programming | February 2, 2017

Problem Set 1 Solutions

1. Re-write the following arithmetic expressions as SCHEME expressions and show the result of the SCHEME interpreter when invoked on your expressions.

   (a) $(22+42)\times(54\times 99)$.

   ```
   (* (+ 22 42)
      (* 54 99))
   ```

   (b) $((22+42)\times 54)\times 99$.

   ```
   (* (* (+ 22 42)
         54)
      99)
   ```

   (c) $64\times 102+16\times(44/22)$.

   ```
   (+ (* 64 102)
      (* 16
         (/ 44 22)))
   ```

   (d) Of course, the first two expressions evaluate to the same number. In what sense are they different? How is this reflected in the SCHEME expression?

   The order of operations is different. Observe how this changes the parenthetical structure of the SCHEME expression.

   (e) In a conventional unparenthesized (infix) arithmetic expression, like 3+4∗5, we rely on a *convention* to determine which operation we apply first (the convention is typically called the "rules of precedence"). Are rules of precedence necessary for evaluating arithmetic expressions in SCHEME? Explain your answer.

   No. Any such arithmetic expression determines unambiguously which operator to apply first; the same can be said of all subexpressions!

2. The company HP designed several families of scientific calculators which exploited a nice fact about *postfix* notation for arithmetic expressions: parentheses are unnecessary to uniquely define the expression.

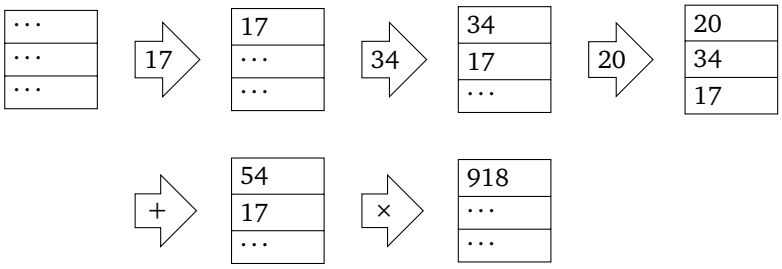
   The architecture of these calculators consisted of a simple memory structure called a "stack," which you may think of as a sequence of numbers which we will denote $(n_1,\ldots,n_k)$. Thus $(3,6,11)$ and $(.001,0,100000,45)$ are two different stacks; the first contains three numbers; the second contains four numbers. The stack structure is easy to reason about because there are only two different "operations" that are permitted on a stack: *push*, which adds a new element to the front of the stack, and *pop*, which removes the first element from the stack. In general, if the *push* operation is carried out with a number n and a stack $(n_1,\ldots,n_k)$, the result is the stack $(n,n_1,\ldots,n_k)$. On the other hand, if the *pop* operation is carried out on the stack $(n_1,\ldots,n_k)$, the result is the stack $(n_2,\ldots,n_k)$. Note that in order to carry out a *push* operation, one needs a new element to push onto the stack; note, also, that the *pop* operation only makes sense if the stack contains at least one number.

   The HP calculators used this stack architecture to provide the familiar functions of a calculator in the following specific fashion. At all times the calculator maintained a stack of numbers, and there were two ways that the user could interact with the stack:

(a) the user could *push* a new number onto the stack, or

(b) the user could apply a function (like $*$, /, +, and $-$) which would *pop* two elements from the stack, apply the operation to the two elements, and push the result back on to the stack.

The calculator typically showed you the first two elements of the stack, so you could read out the result of your computation. For example, a user who wished to compute $17 \times (34 + 20)$ would push the number 17 onto the stack, push the number 34 onto the stack, push the number 20 onto the stack, apply the + operator, and then apply the $\times$ operator. A diagram of this computation is shown below, which indicates the contents of the stack at various times (written vertically instead of horizontally, as above):



For convenience, we can denote this sequence of operations on the calculator as $[17][34][20] + \times$. (So a number in brackets indicates that the number is pushed on to the stack.) Note that, in general, the *order* of the operands for operations such as $-$ and / is important (e.g., $3-1$ is different from $1-3$). The convention on the HP calculators is that the first of the two values popped off the stack is the first argument; thus the $-$ operator applied to the stack $(1, 2, \ldots)$ will result in the stack $(-1, \ldots)$.

Give the sequence of operations that would compute the expressions of problem 1 above. Be sure that operations are applied with the same groupings as indicated in the original expressions.

(a) Give the sequence of operations to compute the expression $(22 + 42) \times (54 \times 99)$.
$[22][42] + [54][99] \times \times$

(b) Give the sequence of operations to compute the expression $((22 + 42) \times 54) \times 99$.
$[22][42] + [54] \times [99] \times$

(c) Give the sequence of operations to compute the expression $64 \times 102 + 16 \times (44/22)$.
$[64][102] \times [16] \times [44][22] / \times +$

3. Write SCHEME definitions for the functions below. Use the interpreter to try them out on a couple of test cases to check that they work, and include this output with your solutions.

(a) `inc`, the function $\texttt{inc}(x) = x + 1$. Once you have defined `inc` show how to use it to define the function $\texttt{inc2}(x) = x + 2$. (So, your definition of `inc2` should not use the the function +.)

```
(define (inc x) (+ x 1))
(define (inc2 x) (+ x 2))
```

(b) `square`, the function $\texttt{square}(x) = x^2$. As above, use the function `square` to define the function $\texttt{fourth}(x) = x^4$.

```
(define (square x) (* x x))
(define (fourth x) (square (square x)))
```

(c) p, the polynomial function $\texttt{p}(x) = (x^2 + 1)^4(16x^4 + 22)^2$.
(Hint: Of course it is possible to expand $(x^2 + 1)^4(16x^4 + 22)^2$ as a polynomial of degree 16 and write a SCHEME function to compute this by brute force. You can avoid all this computation by defining p in stages as you did above.)

```
(define (p x) (* (fourth (+ 1 (square x)))
                 (square (+ 22 (* 16 (fourth x))))))
```

(d) Using the function `fourth`, write the function `sixteenth`$(x) = x^{16}$. Similarly, using the functions `fourth` and `sixteenth`, write the function `sixty-fourth`$(x) = x^{64}$. Recall that $64 = 16 \times 4$. (You may want to test this on an input relatively close to 1, such as 1.01.)

```
(define (sixteenth x)
  (fourth (fourth x)))
(define (sixty-fourth x)
  (fourth (sixteenth x)))
```

(e) Reflect on your definition of `sixty-fourth` above. What would have been the difficulty of defining this merely in terms of $*$?

Without abstraction (that is, using a subordinate function), this would have required us to write 64 occurrences of the variable `x`.

Remark SCHEME provides built-in support for exponentiation (via the `expt` function, defined so that `(expt x y)` yields x^y). For the exercises above, however, please construct the functions $x \mapsto x^k$ using only $*$ and function application.

4. Write SCHEME definitions for the functions below. Use the interpreter to try them out on a couple of test cases to check that they work, and include this output with your solutions.

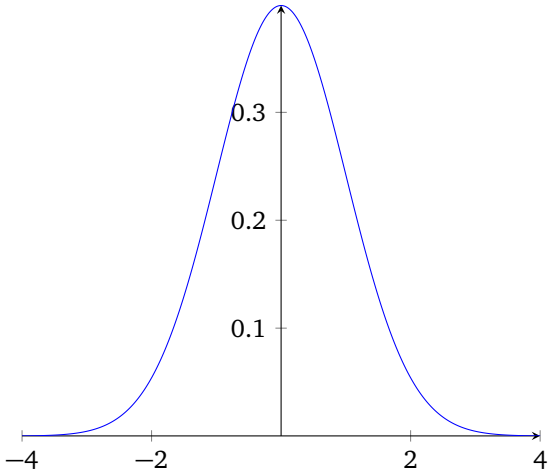
(a) The normal distribution with standard deviation σ at the point x is given by the function

$$N(x, \sigma) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{x^2}{2\sigma^2}}.$$

Write a SCHEME function `normal` so that `(normal x sig)` returns $N($`x`$,$`sig`$)$. You may use the built-in functions `sqrt` and `exp` which compute the square root and e^x functions, respectively. For simplicity, you may approximate π by the number 3.142.

```
(define (square x) (* x x))
(define (normal x sig)
  (* (/ 1 (sqrt (* 2 3.142 (square sig))))
     (exp (* -1 (/ (square x)
     (* 2 (square sig)))))))
```

The normal distribution is the single most important distribution is all of statistics. In general, it has a bell shape, as shown below:

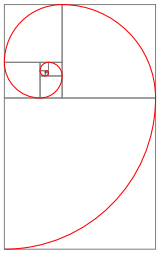


(b) The Fibonacci spiral, shown below, is a rather remarkable mathematical object which mysteriously appears in a number of places in biology. It is described by the polar equation

$$r(\theta) = \phi^{\theta \cdot (2/\pi)}.$$

Here ϕ is a constant called the *golden ratio* which you may approximate by 1.618. Likewise π is the familiar constant which you may approximate by 3.142. In order to carry out the exponentiation, please use the built-in function `expt` defined so that `(expt x y)` returns x^y. Using these, write a SCHEME function `fspiral` so that `(fspiral theta)` retuns the value r(`theta`).

```
(define (fspiral theta)
        (expt 1.618 (* theta (/ 2 3.142))))
```



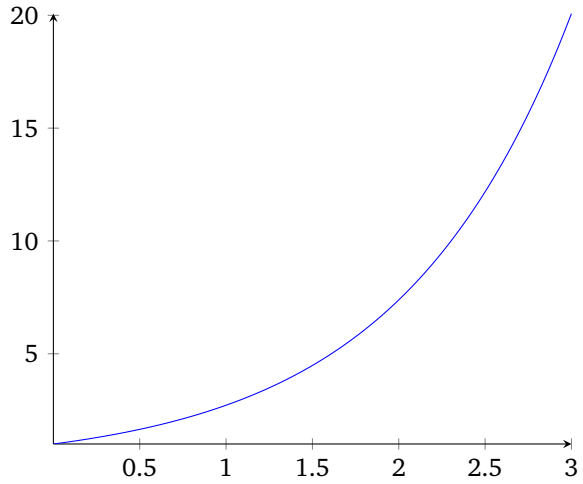
(c) The classical exponential (or "Malthusian") population growth model predicts the size of a population at a future time based on its current size. Specifically, if the population has size P_0 at time 0, the model predicts that the size of the population at time t will be

$$P_0 \cdot 2^{\alpha t},$$

where α is a parameter of the process. Write a scheme function `malth` so that `(malth t p a)` returns the value $p \cdot 2^{at}$.

```
(define (malth t p a)
        (* p (expt 2 (* t a))))
```

You can see the rough behavior of the model below.



(d) A more sophisticated population model can reflect a penalty term that attenuates growth when the population becomes large. (This can occur, for example, if a population has a food source of constrained size: when the population is small, food is relatively plentiful and the population grows; when the population is large, there is intense competition for food and the population does not grow as quickly.) A standard model to capture this is called the *single species model* and depends on three parameters: the initial population size P_0, a "stable population size" P_∞, and a rate $\alpha > 0$. Then, as a function of time t, the population has size

$$P(t) = \frac{P_\infty \cdot P_0}{P_0 + (P_\infty - P_0)e^{-\alpha t}}.$$

Write a SCHEME function `singlespecies` so that `(singlespecies Pi Ps alpha t)` returns the value $P(t)$ where $P_0 =$ `Pi`, $P_\infty =$ `Ps` and $\alpha =$ `alpha`.

```
(define (singlespecies Pi Ps alpha t)
        (/ (* Pi Ps)
           (+ Pi (* (- Ps Pi)
                    (exp (* -1 alpha t))))))
```

To visualize the behavior of such a model, the graph below shows the setting when $P_0 = 10$, $P_\infty = 1000$, and $\alpha = 1/25$.