University of Virginia **Out: 5 December 2011**
cs1120: Introduction of Computing **Due: 1:00pm, Monday, 12 December 2011**
Explorations in Language, Logic, and Machines

Final Exam - Solutions

Language

1. Write a small BNF replacement grammar that produces *exactly* 27 strings. For full credit, your grammar should use only two different non-terminals and no more than three terminals.

$S \rightarrow T\ T\ T$
$T \rightarrow$ **a** | **b** | **c**

This produces the 27 = 3×3×3 strings: aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc

2. Write a BNF replacement grammar that produces all strings made up of **0**s and **1**s that end in a **0** and no other strings. For example, your grammar should produce the strings **01001010** and **0**, but not **10101** or the empty string.

$S \rightarrow T$ **0**
$T \rightarrow \varepsilon \mid B\ T$
$B \rightarrow$ **0** | **1**

Defining Procedures

A famous unsolved problem in mathematics is the Collatz conjecture:

1. Start with any natural number, *n*.
2. If *n* is even, divide *n* by two. If *n* is odd, multiply it by 3 and add 1.
3. Keep going until you reach one.

For example, starting from $n = 1120$, we would go through this sequence:

560, 280, 140, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

The Collatz conjecture speculates that no matter what number you start with, this process eventually reaches 1. It is not known if this is true (but no one has yet found a natural number for which it fails).

3. Define a procedure using either Scheme or Python (your choice) that takes as input a natural number *n* and tests the Collatz conjecture for *n*. Your procedure should output the sequence of values on the path to reaching one as a list. If the Collatz conjecture is false (that is, the value never reaches 1), your procedure may run forever.

For example, in Scheme:

```
> (collatz 1120)
(1120 560 280 140 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1)
```

in Python:

```
>>> collatz(1120)
[1120, 560, 280, 140, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

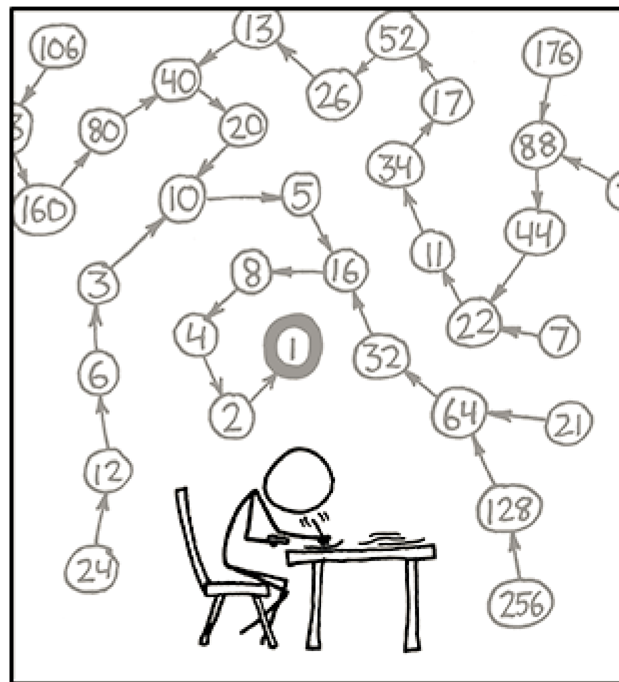
In Python:

```
def collatz(n):
   output = []
   while n != 1:
      output.append(n)
      if n % 2 == 0:
         n = n / 2
      else:
         n = 3 * n + 1
   output.append(n)
   return output
```

[Bonus] Explain why determining if a correct collatz procedure (that solves the previous question) is an *algorithm* would be worth at least a quadruple-gold-star.

As mentioned in the question, it is not known if the Collatz process always reaches 1. Our collatz procedure will always terminate if the Collatz conjecture is true, but will run forever on some natural number input if the Collatz conjecture is not true. So, our procedure is an algorithm if and only if the Collatz conjecture is true, which requires solving an open mathematical problem.

(I should have worded this question more carefully. If we allow inputs that are not natural numbers, it is easy to answer the question since the procedure (as we defined it) runs forever on input 0. Hence, it is obvious that it is not an algorithm, but since the Collatz procedure is only defined for natural numbers, it doesn't matter for this question what happens on other inputs.)



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

xkcd.com

Running Time Analysis

4. For each procedure below, give the worst-case asymptotic running time. Remember to define all variables you use in your answer clearly, and state any assumptions.

   a.

```
(define (mlist-copy p)
   (if (null? p)
      null
      (mcons (mcar p) (mlist-copy (mcdr p)))))
```

The worst-case asymptotic running time is in $\Theta(N)$ where N is the number of elements in p (which is the size of the input).

There are N recursive calls, and each call involves constant work.

b.

```
def isSquare(n):
   for i in range(1, n):
      if i * i == n:
         return True
   return False
```

Assuming the * operations are constant time, the worst-case asymptotic running time is in $\Theta(V)$ where V is the value of input n, or $\Theta(2^N)$ where N is the size (length) of the input. The worst-case input is when n is not a square and its value is the maximum value that can be written in the input space ($2^N - 1$ using binary notation). Then, there number of loop iterations is n, and each iteration is constant time (with the aggressive assumption about *).

c. A magic square is a square where each row, column, and both diagonals sums to the same value. Give the asymptotic running time of this Python procedure that tests if a square passed in as a list of lists is a magic square. (You may assume all arithmetic operations used are constant time, and that the input p is a square.)

```
def isMagicSquare(s):
  target = 0
  for e in s[0]:
    target = target + e
  diagsum = 0
  revdiagsum = 0
  for i in range(0, len(s)):
    colsum = 0
    rowsum = 0
    diagsum = diagsum + s[i][i]
    revdiagsum = revdiagsum + s[i][len(s) - i - 1]
    for j in range(0, len(s)):
      rowsum = rowsum + s[j][i]
      colsum = colsum + s[i][j]
    if colsum != target or rowsum != target: return False
  return diagsum == target and revdiagsum == target
```

Suppose S is an $M \times M$ square. Then, the running time is in $\Theta(M^2)$.
In terms of the size of the input, N, the running time is in $\Theta(N)$. The reason for this is that the value of len(s) is $\Theta(\text{sqrt}(N))$ since *s* is a square matrix. Thus the inner loop (**for j in range(0, len(s))** involves $\Theta(\text{sqrt}(N))$ iterations, and its body is constant time (with the assumption about arithmetic and indexing being constant). There are $\Theta(\text{sqrt}(N))$ iterations of this loop from the outer loop (**for i in range(0, len(s)))**, so the total running time is in $\Theta(\text{sqrt}(N)) \times \Theta(\text{sqrt}(N)) = \Theta(N)$.

Mutation

5. Define a procedure that takes as input a mutable list of numbers, and modifies the list so that every other element is replaced with its negation. So, the first element should be negated, but not the second, etc. You may use *either* Scheme or Python to define your procedure (your choice). For example, in Scheme you would define the **mlist-negate-every-other!** procedure that behaves like this:

```
> (define p (mlist 1 -2 3 0 -17))
> (mlist-negate-every-other! p)
> p
{-1 -2 -3 0 17}
```

In Python you would define the **listNegateEveryOther** procedure that behaves like this:

```
>>> p = [1, -2, 3, 0, -17]
>>> listNegateEveryOther(p)
>>> p
[-1, -2, -3, 0, 17]
```

```
def listNegateEveryOther(p):
  odd = True
  for i in range(0, len(p)):
    if odd: p[i] = -p[i]
    odd = not odd
```

6. Define a Scheme or Python procedure that takes as input a mutable list, and modifies the list so that each element is the maximum value of all elements up to and including itself. Your procedure must modify the input list.

   For example, in Scheme:

   ```
   > (define p (mlist 1 2 3 2 4 6 3))
   > (make-maxilative! p)
   > p
   {1 2 3 3 4 6 6}
   ```

   For example, in Python:

   ```
   >>> p = [1, 2, 3, 2, 4, 6, 3]
   >>> makeMaxilative(p)
   >>> p
   [1, 2, 3, 3, 4, 6, 6]
   ```

```
def makeMaxilative(p):
  max = False
  for i in range(0, len(p)):
    if not max or p[i] > max:
      max = p[i]
    else:
      p[i] = max
```

Interpreters

Suppose we want to add a new special form to Charme similar to the **for** construct in Python. The grammar for the **for** special form is:

Expression → *ForExpression*
ForExpression → **(for** *Name* **in** *Expression*$_1$ **do** *Expression*$_2$**)**

The first expression should evaluate to a list; the second expression can be any expression. The evaluation rule is:

> To evaluate the **for** expression, **(for *N* in *E*$_1$ do *E*$_2$)**, evaluate the expression ***E*$_1$**, which must evaluate to a list. Then, for each element in the list evaluate the expression ***E*$_2$** in a new environment where the name ***N*** is bound to the value of that element. The value of the for expression is a list containing the results of each evaluation of ***E*$_2$**.

Here are a few examples:

```
> (for x in (list 1 2 3 4) (+ x 1))
(2 3 4 5)
> (for x in (list 2 3) (* x x))
(4 9)
```

7. Define an **evalFor(expr, env)** procedure that implements the evaluation rule for the **for** special form as defined above. You may assume the value passed in as **expr** is parsed Charme expression and that corresponds to a syntactically valid **for** expression.

```
def evalFor(expr, env):
  var = expr[1]
  lst = meval(expr[3], env)
  body = expr[4]
  res = []
  for el in lst:
    newenv = new Environment(env)
    newenv.addVariable(var, el)
    res.append(meval(body, newenv))
  return res
```

8. Would it be a good idea to add **for** to Charme (or Scheme)? Provide a convincing argument supporting your answer.

Not really. There is no need for a special construct for this since we could just define map as a regular procedure that does essentially the same thing.

Computability

9. Is the *Runs-Forever* problem defined below computable? For full credit, your answer must include a clear and convincing proof supporting your answer.

   Input: A string *s* that defines a Python program.

   Output: True if the program defined by *s* runs forever. False if the program defined by s will eventually finish.

No. RUNS-FOREVER is not computable. To prove this, we show how we could define an algorithm **halts(s)** that solves the HALTING-PROBLEM using an algorithm **runsForever(s)** that solves the RUNS-FOREVER problem. This is easy since the result for halts is just the opposite of the result for runsForever:

```
def halts(s):
  return not runsForever(s)
```

10. Is the *RUNS-LONGER* problem defined below computable? For full credit, your answer must include a convincing proof supporting your answer.

    Input: Strings that define two Python procedures, *p* and *q*.

    Output: True if the procedure defined by *p* takes longer to finish running than the procedure defined by *q*. A procedure *takes longer to finish* than another procedure if the first procedure finishes in *K* steps (for some number *K*), and the second procedure does not finish with *K* steps.

This is trickier, but is also **not computable**.

Here's how to use an algorithm that solves **runsLonger(p, q)** to implement **halts(p)**:

```
def halts(p):
  return runsLonger("pass; " + p, p)
```

If **p** halts, that means it finishes in *K* steps for some *K*. Assuming **pass** takes one step, **"pass; " + p** will finish in *K*+1 steps. Hence, when **p** halts, **runsLonger("pass; " + p, p)** returns **True**, which is the correct result.

If **p** does not halt, both inputs to **runsLonger** run forever, and **runsLonger** outputs **False**, which is the correct result.