University of Virginia **Out: 2 December 2011**
cs1120: Introduction of Computing **Due: 11:01 am, Wednesday, 30 November**
Explorations in Language, Logic, and Machines

Exam 2 - Solutions

First Question

1. [Average: 10.0/10] According to Dean Kamen's speech at the Rice Hall dedication (quoting Walt Havenstein), what is "the only sport I know where everyone can turn pro"?

   (Note: if you had the misfortune of not being able to attend Dean's talk, you should still be able to answer this question by searching for the quote using DuckDuckGo, Google, or Bing.)

   The FIRST robotics competition (see http://www.usfirst.org/). More general answers about science and technology were also accepted.

Running Time Analysis

2. [Average: 9.2/10] Describe the worst-case asymptotic running time of the all-positive? Scheme procedure defined below (from the Exam 1 comments). You may assume that all elements of **p** have values below some bound *k*, so the running time of **>** is constant. Remember to clearly define all variables you use in your answer.

```
(define (all-positive? p)
  (if (null? p)
     true ; reached end without finding non-positive, so result is true
     (if (> (car p) 0)
        (all-positive? (cdr p)) ; keep looking
        false))) ; found one non-positive
```

The running time of **all-positive?** is in $\Theta(N)$ where N is the number of elements in p. There are N recursive calls to **all-positive?**, since each call **cdr**'s down the list. The work for each application of **all-positive?** is (at worst) applications of **null?**, **>**, **car**, and **cdr**. The running time for **null?**, **car**, and **cdr** is always constant. As stated in the question, we can also assume the running time for **>** is constant since the maximum value of the input is bounded and does not scale with N. Hence, the total running time for each application of **all-positive?** is constant, and the total running time for N applications is in $\Theta(N)$.

3. [Average: 8.7/10; this was easier than 2, but many people were not confident enough to give the same answer twice!] Describe the worst-case asymptotic running time of the allPositive Python procedure defined below (that behaves similarly to the Scheme procedure above). You may assume that all elements of **p** have values below some bound *k*, so the running time of **<=** is constant. Remember to clearly define all variables you use in your answer.

```
def allPositive(p):
  for x in p:
    if x <= 0: return False
  return True
```

The running time of **allPositive** is in $\Theta(N)$ where N is the number of elements in p. The for loop goes through the elements of p, so there will be N iterations of the loop body. The loop body runs in constant time since we assume the running time of the **<=** comparison is constant.

4. [Average: 8.4/10] Describe the worst-case asymptotic running time of the **goldStarsSquare(g)** Python procedure defined below. The input, **g**, is a natural number. You should state clearly all assumptions you make. Remember to clearly define all variables you use in your answer. The best answers will be in terms of the *size* of the input, not the value of **g**, but you will receive nearly full credit for a correct answer in terms of the value of **g**.

```
def rowGoldStars(g):
  for i in range(0, g):
    print "*",
  print # (end the row by printing a new line)

def goldStarsSquare(g):
  for i in range(0, g):
    rowGoldStars(g)
```

The running time of **goldStarsSquare** is in $\Theta(V^2)$ where V is the magnitude (value) of **g**, assuming the running time of **print** is linear in the length of what is printed (or that the running time of **print** is constant, since in this case, each print application has the same input length).

The **rowGoldStars** procedure is a **for** loop that iterates from 0 to g-1. The running time of the loop body is constant, so the running time of **rowGoldStars** is in $\Theta(G)$ where G is the value of the input to **rowGoldStars**.

The **goldStarsSquare** procedure is a **for** loop that iterates from 0 to g-1. The loop body calls **rowGoldStars(g)**, which as analyzed above has running time in $\Theta(G)$ where G is the value of the input to **rowGoldStars**. In this case, the value of the input is the input **g** to **goldStarsSquare**, which we defined as V, so the running time is in $\Theta(V)$. The number of iterations of the loop body is the value of **g** (= V), so the total running time is in $V \times \Theta(V) = \Theta(V^2)$.
For an answer in terms of the *size* of the input, rather than its value when interpreted as a number, we need to consider the largest number that can be written down using N input squares. With binary notation, this is 2^N-1. (With more input symbols, it could be larger, like

10^N-1 for decimal, but within the asymptotic operators this doesn't make a difference.) So, in terms of the input size *N*, the running time is in $\Theta((2^N)^2)$. We can simplify this further since $(a^m)^n = a^{mn}$, so it simplifies to $\Theta(2^{2N})$, which within the asymptotic operator can be written equivalently as $\Theta(2^N)$.

Mutation

5. [Average: 8.6/10] Define a procedure that takes as input a mutable list of numbers, and modifies the list so that each element is replaced with its negation. You may use *either* Scheme or Python to define your procedure (your choice). For example, in Scheme you would define the **mlist-negate!** procedure that behaves like this:

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

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

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

In Scheme:

```
(define (mlist-negate! p) (mmap - p))
```

(to get exactly the behavior shown, we should do:

```
(define (mlist-negate! p) (begin (mmap - p) (void))
```

since **mmap** returns the value of the modified list.

Without using mmap:

```
(define (mlist-negate! p)
   (if (null? p)
       (void)
       (begin
          (set-mcar! p (- (mcar p)))
          (mlist-negate! (mcdr p)))))
```

In Python:

```
def listNegate(p):
   map(lambda x: -x, p)
```
or:
```
def listNegate(p):
  for i in range(0, len(p)):
    p[i] = -p[i]
```

6. [Average: 7.3/10] Define a Scheme procedure **make-cumulative!** that takes as input a mutable list, and modifies the list so that each element is the cumulative total of all elements up to and including itself. For example,

   ```
   > (define p (mlist 1 2 3 4 5))
   > (make-cumulative! p)
   > p
   {1 3 6 10 15}
   ```

   Hint: you probably should start by defining a helper procedure. (For full credit for this question, you must use Scheme. But, a correct answer using Python is worth most of the credit.)

This one is fairly tricky to do in a functional-style (of course it can't be purely functional since it is using mutation, but in the Scheme-style of programming) since you have to unravel the list in the right order.

```
(define (make-cumulative-helper! p tot)
 (if (null? p)
    (void)
    (begin
      (make-cumulative-helper! (mcdr p) (+ (mcar p) tot))
      (set-mcar! p (+ (mcar p) tot)))))

(define (make-cumulative! p)
 (make-cumulative-helper! p 0))
```

(Note that my original solution for this was wrong! I had the order of the statements in the begin expression swapped, so the value of **(mcar p)** used to pass in the new total was the wrong value, after updating it with the cumulative total. I foolishly pasted the results into the exam without actually looking that they were correct. Sorry for the confusion on this...but another example of why programming with mutation gets more complicated.)

Another way to do this in Scheme, but with a more imperative style of programming, is to keep the running total in a variable. The risk here is that you need to make sure the variable is initialized correctly for each application.

```
(define running-total 0)

(define (make-cumulative! p)
  (if (null? p)
     (set! running-total 0) ; get ready for next application
     (begin
       (set! running-total (+ running-total (mcar p)))
       (set-mcar! p running-total)
       (make-cumulative! (mcdr p)))))
```

This style is more natural in Python:

```
def makeCumulative(p):
   total = 0
   for i in range(0, len(p)):
      total = total + p[i]
      p[i] = total
```

Interpreters

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

> *Expression* $\rightarrow$ *AndExpression*
> *AndExpression* $\rightarrow$ **(and** *ExpressionList***)**
> *ExpressionList* $\rightarrow$ ε
> *ExpressionList* $\rightarrow$ *Expression ExpressionList*

The evaluation rule is:

> To evaluate the **and** expression, **(and E_0 E_1 … E_n)**, evaluate each sub-expression in order until one evaluates to a **false** value. If any sub-expression evaluates to a false value, the value of the **and** expression is false, and none of the following sub-expression is evaluated. If none of the sub-expressions evaluate to a false value, the value of the and expression is **true**. (Note that this means **(and)** should evaluate to **true**.)

7. [Average: 8.1/10] Explain why **and** needs to be a special form (that is, it requires modifying the Charme evaluator and could not be implemented as a primitive procedure).

The **and** expression must be a special form in Charme since it does not evaluate all its subexpressions the way a normal application does. The first operand subexpression must be evaluated first, and then the next operand is only evaluated if the previous ones evaluated to **true** values. There is no way to define a procedure that behaves this way, since the evaluation rule for application is to first evaluate all the subexpressions.

8. [Average: 7.1/10] Define an **evalAnd(expr, env)** procedure that implements the evaluation rule for the and special form. You may assume the value passed in as **expr** is parsed Charme expression and that corresponds to a syntactically valid and expression.

(Note that the original question didn't include the **env** parameter, which is necessary to have an evaluation environment.)

```
def evalAnd(expr, env):
   for clause in expr:
      if not meval(clause, env):
         return False
   return True
```

Computability

The cs1120 staff is getting tired of having to grade so many programming questions, so wants to build a procedure that automates grading by checking if a solution submitted implements the same function as our reference solution.

9. [Average: 6.7/10] Is the *Correct-Answer* problem defined below computable? For full credit, your answer must include a convincing proof supporting your answer.

   Input: Strings that define two Python procedures, *r* (the reference procedure) and *s* (the submitted procedure).

   Output: True if the procedure defined by *s* correctly implements the same function as the procedure defined by *r*. We say *s* correctly implements the same function as *r* if for every input for which *r* produces a value, s produces the same value.

The *CORRECT-ANSWER* problem is **not computable**.

To prove that it is not computable, we show how an algorithm that solves *CORRECT-ANSWER* could be used to create an algorithm that solves the *HALTING-PROBLEM*, which we know is impossible. Assume **correctAnswer(r, s)** is an algorithm that solves *CORRECT-ANSWER*. Define **halts(p)**, an algorithm that solves the *HALTING-PROBLEM* as:

```
def halts(p):
   correctAnswer('def r(x): return 37', 'def s(x): eval(' + p + '); return 37')
```

The reference procedure, passed in as the first input, is a procedure that always produces 37 as its output no matter what the input is. The submitted procedure is a procedure that first evaluates ***p***, the input to halts, and then returns 37. If **p** does not halt, this will not produce a value, and **correctAnswer** will output False, which the correct result for **halts(p)**. If **p** does halt, **s(x)** will return 37 for all inputs x, so **correctAnswer** will output True, which is the correct result for **halts(p)**. Hence, an algorithm that solves *CORRECT-ANSWER* would allow us to create an algorithm that solves the *HALTING-PROBLEM*, so such an algorithm must not exist and *CORRECT-ANSWER* must not be computable.

10. [Average: 7.0/10] Is the *PROBABLY-CORRECT-ANSWER* problem defined below computable? For full credit, your answer must include a convincing proof supporting your answer.

    Input: Strings that define two Python procedures, *r* (the reference procedure) and *s* (the submitted procedure), a set *V* of test inputs, and a number *t* for the maximum number of steps.

    Output: True if the procedure defined by *s* probably implements the same function as the procedure defined by *r*. We say *s* probably implements the same function as *r* if for each element *v* of *V*, if *r*(*v*) produces a value within *t* steps, *s*(*v*) produces the same value within *t* or fewer steps.

Yes, *PROBABLY-CORRECT-ANSWER* is computable. Indeed, you have been using an algorithm that solves it since September - this is what the Alonzo-bot service does when you submit code for automated testing!

The way to implement *PROBABLY-CORRECT-ANSWER* is to simulate *s*(*v*) for up to *t* steps (or in the Alonzo-bot server's cause, for some maximum running time). If it has not finished by t steps, it is considered wrong (even if it would have produced a correct answer in *t*+1 steps).