cs1120: Final Exam - Comments

Defining Procedures

For questions 1-3, provide a procedure with the described procedure. For each question, you may use either Scheme or Python for your procedure. If you are not confident your code is correct, it is also a good idea to include an English prose description of your procedure.

1. (Average: 8.8/10) Define a procedure *count-positive* that takes a input a list of numbers and produces as output a number representing the number of positive numbers in the input list.

For example,

Scheme: (*count-positive* (*list* 3 -2 4 0 12))
Python: *count_positive*([3, -2, 4, 0, 12])

should evaluate to 3.

Scheme:

```
(define (count-positive p)
   (if (null? p) 0
      (+ (if (> (car p) 0) 1 0) (count-positive (cdr p)))))
```

or,

```
(define (count-positive p)
   (list-length (list-filter (lambda (x) (> x 0)) p)))
```

Python:

```
def count_positive(p):
  count = 0
  for x in p:
    if x > 0: count = count + 1
  return count
```

or,

```
def count_positive(p):
  len(filter(lambda x: x > 0, p))
```

2. (7.5) **(Exercise 4.8)** Define a procedure *find-maximum-epsilon* that takes as input a function *f*, a low range value *low*, a high range value *high*, and an increment *epsilon*, and produces as output the maximum value of *f* in the range between *low* and *high* at interval *epsilon*.

For example,

Scheme: (*find-maximum-epsilon* (**lambda** (*x*) (* *x* (- 5.5 *x*))) 1 10 1)
Python: *find_maximum_epsilon*(**lambda** *x*: *x* * (5.5 - *x*), 1, 10, 1)

should evaluate to 7.5.

Scheme:

```
(define (find-maximum-epsilon f low high epsilon)
  (if (>= low high)
     (f low)
     (bigger (f low)
              (find-maximum-epsilon f (+ low epsilon) high epsilon))))
```

Python:

```
def find_maximum_epsilon(f, low, high, epsilon):
  res = f(low)
  while low <= high:
     res = max(res, f(low))
     low = low + epsilon
  return res
```

3. (8.25) Define a procedure *item-count* that takes as input a list of items, and outputs a list of <item, count> pairs indicating for each item that appears in the input list the number of times that item occurs.

For example,

Scheme: (*item-count* (*list* 'everything 'plain 'plain 'salt 'sesame 'everything))
Python: *item_count*(['everything', 'plain', 'plain', 'salt', 'sesame', 'everything'])

should evaluate to:

Scheme: (('everything . 2) ('plain . 2) ('salt . 1) ('sesame . 1))
Python: {'everything': 2, 'plain' : 2, 'salt': 1, 'sesame': 1}

(note that the order in which elements appear in the output doesn't matter).

This is very similar to the histogram procedure from Class 27 (and Exam 2).

```
def item_count(p):
    d = {}
    for x in p:
        if x in d:
            d[x] = d[x] + 1
        else:
            d[x] = 1
    return d
```

Analyzing Procedures

4. (9.3) What is the asymptotic running time of the *list-cruncher* procedure defined below:

```
(define (list-cruncher p)
  (if (null? p)
      null
      (if (odd? (car p) )
          (list-cruncher (cdr p))
          (cons (car p) (list-cruncher (cdr p))))))
```

For full credit, your answer must carefully define the meaning of any variables you use.

The running time of *list-cruncher* is in $\Theta(N)$ where N is the number of elements in p.

Other than the recursive call, the procedure involves applications of only constant time procedures: *null?*, *car*, *cdr*, *cons*, and *odd?* (note that *odd?* should be constant time, since we can tell if a number is odd by only looking it its rightmost bit, so the time to test for odd-ness does not scale with the size of the input number). The number of recursive calls is the number of elements in p, since each recursive call passed in (*cdr p*), and the base case is when the list is null.

5. (8.1) What is the worst-case asymptotic running time of the *disjoint* procedure defined below?

```
def disjoint(p1, p2):
    """Returns true if p1 and p2 are disjoint (that is, there is no common element in
       both lists.)"""
    for e1 in p1:
        for e2 in p2:
            if e1 == e2:
                return False
    return True
```

For full credit, your answer must carefully define the meaning of any variables you use in terms of the size of the input, and explain what the worst-case inputs are.

The worst-case running time of *disjoint* is in $\Theta(N^2)$ where N is the total input size.

The worst-case input is when the lists are disjoint (or equivalent, when the very last elements checked match). In this case, we need to go through all elements of *p1*, and for each of these, go through all elements of *p2*. The outer for loop will iterate *N1* times, where *N1* is the number of elements in p1; the inner loop will iterate *N2* times for each element in p1, where *N2* is the number of elements in p2. So, the total number of loop iterations (that is, the number of times, the if predicate e1 == e2 must be evaluated) is *N1* * *N2*. For a given input size *N* = *N1* + *N2*, the value of *N1* * *N2* is maximized when *N1* = *N2*.

6. (5.8) Suppose you have two correct sorting procedures, *sortA* and *sortB*. The *sortA* procedure has asymptotic running time in $\Theta(N^2)$ where N is the number of elements in the input list. The *sortB* has asymptotic running time in $O(N \log N)$ where N is the number of elements in the input list. For a given application, you need to sort a list of 1120 numbers. Which procedure is best for this application? (Either provide a clear argument of which procedure is best, or a convincing explanation of why you do not have enough information to answer the question.)

People had a tough time with this one. In some ways, it is a "trick" question, but the intent is to see how well you understand the meaning of the asymptotic operators and can relate theoretical results to practical questions (ala Ali G). The correct answer is that there is not enough information to answer the question. Recall that the asymptotic operators hide constants. So, functions that are in $\Theta(N^2)$ include $0.0001N^2$ and functions that are in $O(N \log N)$ include $99999999999\ N + 123859382493$. So, for these examples, the $O(N \log N)$ function's value for N=1120 is higher than the $\Theta(N^2)$ function's value. For a high enough input N, the $\Theta(N^2)$ value will always eventually be larger than that of any function in $O(N \log N)$ since N^2 grows faster than $N \log N$, but for any particular value of N, the value of a function in $O(N \log N)$ may be higher.

Another reason the *sortA* procedure might be better (even in cases where its running time is higher) is if there are other concerns more important than running time, for example memory use.

Computability

7. (7.1) **(similar to Exercise 12.2)** Is the *Same-Result* problem described below computable or noncomputable? Provide a convincing argument supporting your answer.

Input: Descriptions of two Turing Machines, *M1* and *M2*

Output: If the result of running *M1* starting with an empty input tape is the same as the result of running *M2* starting with an empty input tape, output a 1 at the left edge of the tape. Otherwise, output a 0 at the left edge of the tape. Two Turing Machines are considered to produce the same result if either they both do not halt, or they both halt and leave the final tape with the same contents.

The *Same-Result* problem is noncomputable.

Proof: if we had an algorithm to solve *Same-Result*, we could use it to define an algorithm that solves *Halts*:

```
def halts(p):
  return not same_result(p, INFINITE_LOOP)
```

where INFINITE_LOOP is a Turing Machine that loops forever (never halts).

If the input p halts, it is not the same as INFINITE_LOOP, and halts as defined above correctly returns True. If p does not halt, it has the same result as INFINITE_LOOP, and halts correctly returns False. Thus, a same_result algorithm that solves the *Same-Result* problem would allow us to define a halts algorithm. But, since we know halts is noncomputable, this shows *Same-Result* is also noncomputable.

8. (5.0) **(Challenging)** Is the *Moves-Past-Square-1120* problem described below computable or noncomputable? Provide a convincing argument supporting your answer.

Input: Description of a Turing Machine, *M*

Output: If running *M* starting with an empty input tape would ever move past the 1120^{th} square on the input tape, output true; otherwise, output false. The input tape is infinite in one direction (to the right), and starts on the leftmost square. The squares are numbered starting from 0, so output should be true if M would ever go beyond the square 1120 squares to the right of the left edge of the tape.

The *Moves-Past-Square-1120* problem is computable.

The main intuition why is that the problem is finite: if we limit our TM to 1121 squares (since we don't care what happens after it moves past the 1120^{th} square), the total number of possible TM configurations is finite. It's a huge number: nA^{1120} where n is the number of states in M's FSM and A is the number of alphabet symbols. But, since both n and A must be finite, for any TM M there is some maximum possible number of configurations.

Then, to solve *Moves-Past-Square-1120*, we start with an array containing nA^{1120} elements, all initialized to False. For each simulation step, we check if the element corresponding to the current TM state is True. If it is, that means we have repeated a configuration and the TM is in an infinite loop. The result is False (the TM is in an infinite loop, but it never goes past square 1120). If not, we simulate M's transition to the next state. If the simulate TM ever moves past square 1120, the result is True. Since each step either leads to a result, or changes one of the elements in the configuration array to True, after at most nA^{1120} steps the simulation must be finished.

Thus, this algorithm always halts and always produces the correct result. Hence, *Moves-Past-Square-1120* is computable.

Interpreters

9. (7.7) The Charme interpreter from PS7/Chapter 11 does not define a **begin** special form. Extend the Charme interpreter to support a **begin** special for with the evaluation rule:

Evaluation Rule: Begin. To evaluate a begin expression,

(begin $Expression_1$ $Expression_2$ … $Expression_k$**)**

evaluate each subexpression in order from left to right. The value of the begin expression is the value of the last subexpression, $Expression_k$.

Define an *evalBegin* procedure that evaluates a begin expression. You may assume the expression passed into your procedure is a valid begin expression (that is, you do not need to include defensive error checking code in your procedure).

```
def evalBegin(expr, env):
  for subexpr in expr[1:-1]:
    meval(subexpr, env)
  return meval(expr[-1], env)
```

Computing Concepts

For the last question, you will need another person. This must be someone who has *no significant computing knowledge*. In particular, they should have never written a computer program or taken a computing course. Henceforth, we will call this person the *student*. Your student will need to help you for about 10-15 minutes. It is best if you can find someone who is 5-13 years, but if you can't find someone young it is fine to use a UVa student as your student as long as she/he does not have any significant computing knowledge.

Your goal for this question is to pick some concept from this course and convey something important and interesting about it to your student. The concept you choose to explain may be anything you what that was covered in the class and has some intellectual value. Examples of possible concepts include recursive definitions, defining languages, computability, the Church-Turing thesis, or sorting algorithms, but you do not need to limit yourself to these examples.

10. (8.4) a. Identify one major concept from this course, and describe in your own words what it is and what is interesting and important about it. Write a short plan for how you will introduce and describe this concept to your student, and what you hope your student will understand after your lesson. Feel free to include pictures or drawings, and anything else that you think will be helpful.

b. Find your student and explain the concept to her/him, following your plan from part a. Write a short summary of how things went. Your summary should include interesting questions your student asked and how you answered them, and a description of what you think your student understood at the end of the lesson.

The most popular topics were recursive definitions (by far the most popular), inheritance, running time, languages, and computability. Other interesting choices included endless golden ages, lists, and sorting. Most people had some success conveying their topic to their student.

A few interesting quotes:

"I told this to her and she just kind of looked at me funny and laughed."

"At the end of the lesson, I think she had an appreciation for the rules and methods of combination which underpin all languages, and certainly empathy for how confusing computer science can be at first!"

"He was intrigued by the statement "I always lie" and said thinking about it gave him a headache."

"He said to me, "So it sounds like computer science doesn't really have that much to do with computers at all." 14 years old. Hell of a lot smarter than I was at that age."

"I actually found this portion of the exam very rewarding, becuase I got to show my roommate what I have been doing countless hours this semester. He told me that he'll stick to his reading and essay writing in his history major as this is not how his brain works."