cs150: Final Exam Comments

Procedures

Questions 1 and 2 ask you to define procedures that solve specified problems. For each question, you may use any programming language that has been mentioned in CS150 (Scheme, Python, Charme, LazyCharme, StaticCharme, Fortran, FP, FL, Java, C, C++, PHP, LISP, or JavaScript). If you use a language other than Scheme or Python, you should clearly state in your answer which language you are using.

1. (average 8.86 / 10) Define a procedure, **`xorlist`**, that takes as input a list, and outputs the result of XOR-ing all elements in the list. The result should be true if the list contains an odd number of true values, and false if the list contains an even number of true values. You may assume a procedure, **`xor`**, is defined that takes two inputs and outputs the exclusive or of those inputs.

Here is a simple Scheme definition:

```
(define (xorlist lst)
   (if (null? lst) #f
       (xor (car lst) (xorlist (cdr lst)))))
```

Here's an alternate definition in Python:

```
def xorlist(lst):
   return lst.count(True) % 2 == 1
```

2. (8.3/10) Define a procedure, **`findlast`**, that takes two inputs, a list and a predicate procedure, and outputs the last element in the list for which the predicate procedure applied to that element evaluates to true. If no element in the list satisfies the predicate procedure, **`findlast`** should produce **`null`** (in Scheme) or **`None`** (in Python). For full credit, your procedure's running time should be in $O(n)$ where n is the number of elements in the input list.

The most straightforward solution in Scheme is to find the first satisfying element on the reversed list:

```
(define (findlast lst proc)
   (let ((rev (reverse (filter lst proc)))
     (if (null? rev) null (car rev))))
```

This approach doesn't satisfy the running time requirement, however, since our reverse procedure has running time in $\Theta(n^2)$ where n is the length of the input list. Hence, we use this procedure:

```
(define (findlast lst pred)
  (if (null? lst)
      null
      (if (pred (car lst))
          (let ((last (findlast (cdr lst) pred)))
            (if (null? last)
                (car lst)
                last))
          (findlast (cdr lst) pred))))
```

Python's imperative style offers an easier way to do this:

```
def findlast(lst, pred):
     for i in range(len(lst) - 1, 0, -1):
          if pred(lst[i]):
               return lst[i]
     return None
```

In order for this procedure to have running time in $O(n)$, we need to know the running time of the list indexing operation. Since the procedure involves up to n (the number of elements in the input list) list index operations (with average parameter $n/2$), we need to know the time for list indexing in Python does not depend on the value of the index or length of the list. In Scheme, accessing the k^{th} element of a list requires k cdr-operations, so it is not constant time (and a procedure like this in Scheme would have running time in $\Theta(n^2)$ which is not in $O(n)$). Python, however, does provide list indexing operations that are essentially constant time, so the given procedure does satisfy the needed running time requirement.

Running Time Analysis

3. (9.06 / 10) What is the asymptotic running time of the **`find-middle`** procedure defined below? Your answer should define all variables it contains and clearly state all assumptions you use.

```
(define (find-middle lst)
  (define (find-middle-helper lst n)
    (if (= n 0)
        (car lst)
        (find-middle-helper (cdr lst) (- n 1))))
  (find-middle-helper lst (floor (/ (length lst) 2))))
```

The running time of find-middle is in $\Theta(n)$ where n is the number of elements in the input lst. Evaluating an application of find-middle involves *one* application of the helper procedure, find-middle-helper. The parameters to find-middle-helper are the original input list, and (floor (/ (length lst) 2)). Computing this requires $\Theta(n)$ work since the length procedure is $\Theta(n)$ where n is the number of elements in its input list (equivalent to the number of elements in lst). So, the total running is the work to produce the parameters to find-middle-helper, in $\Theta(n)$, plus the running time for the find-middle-help application. This procedure cdr's down the list, doing constant work for each element, so it has running time in $\Theta(n)$. Thus, the total running time is in $\Theta(n)$.

4. (8.88/10) What is the asymptotic running time of the **`intersect`** procedure defined below? Your answer should define all variables it contains and clearly state all assumptions you use.

```
(define (contains? lst val)
  (if (null? lst)
      #f
      (if (eq? (car lst) val)
          #t
          (contains? (cdr lst) val))))

(define (intersect lst1 lst2)
  (if (null? lst1) null
      (if (contains? lst2 (car lst1))
          (cons (car lst1) (intersect (cdr lst1) lst2))
          (intersect (cdr lst1) lst2))))
```

$\Theta(n_1 n_2)$ where n_1 is the number of elements in lst1 and n_2 is the number of elements in lst2. The intersect procedure cdr's down lst1, so there are n_1 recursive calls (although there are two instances of intersect applications in the definition of intersect, only one of these is executed on each call, since they are in different sub-expressions of the if expression). Each call involves an application of `(contains? lst2 (car lst1))`. The `contains?` procedure cdr's down its input list, doing constant work for each element. So, this involves $\Theta(n_2)$ work. Since there are n_1 calls to contains?, the total running time is in $\Theta(n_1 n_2)$.

Object-Oriented Programming

5. (8.54 / 10) (Exercise 10.3 from the Course Book) Define a new subclass of **`poscounter`** where the increment for each **`next!`** method application is a parameter to the constructor procedure. For example, **`(make-var-counter 0.1)`** would produce a counter object whose counter has value `0.1` after one invocation of the **`next!`** method. (You should assume all definitions from Chapter 10.)

```
(define (make-var-counter increment)
   (make-subobject
       (make-poscounter)
       (lambda (message)
          (if (eq? message 'next!)
              (lambda (self)
                 (ask self 'adjust! increment)))
              #f))))
```

Note that it is better to use the self objects **`adjust!`** method than to attempt to manipulate the **`count`** variable directly (which is not actually possible here).

Computability/Turing Machines

6. (8.9 / 10) Is the *Writes-Hash Problem* described below computable or uncomputable? Your answer should include a convincing argument why it is correct.

Input: A specification of a Turing Machine, *T*, including its tape, using the Turing Machine description grammar from Lecture 37.

Output: Outputs true if running *T* would ever write a "#" symbol on the tape; otherwise, outputs false.

The Writes-Hash Problem is uncomputable. We know the Halting Problem is uncomputable, so we can prove that Writes-Hash in uncomputable by showing how an algorithmic solution to Writes-Hash could be used to solve the Halting Problem.

To do this, we need to transform the input Turing Machine *T*, into a machine with the same halting behavior as *T*, that never writes a "#" symbol on the tape. We can do this by replacing all the transition rules that would write a "#" symbol with rules that are otherwise identical by instead write a "✶" symbol, where ✶ is some symbol that is not used anywhere in *T*. Then, for every transition rule that has "#" as the input symbol, we need to create an additional transition rule with the same states, tape direction, and output symbol but with "✶" as the input symbol. This ensures that the original behavior of T is preserved, except now it writes ✶ instead of #.

Then, we modify all transitions to the Halt state in T to instead transition to a state that writes a "#" on the tape (and then transitions to Halt). Hence, in the modified machine, no "#" is written except if the machine would halt, and then a # is always written. Thus, the result of the Writes-Hash Problem on the modified TM is a solution to the Halting Problem on the original TM. But, since we know the Halting Problem is uncomputable, this means the Writes-Hash Problem is also uncomputable.

Lambda Calculus

7. (8.65 / 10) Define a Lambda Calculus term, **sub**, that corresponds to subtraction. You may assume all the definitions from Lecture 40. Your **sub** definition should satisfy the following properties:

sub 1 zero $\Rightarrow$ **1**
sub 1 1 $\Rightarrow$ **zero**
sub 2 1 $\Rightarrow$ **1**
sub 2 zero $\Rightarrow$ **2**
sub 5 2 $\Rightarrow$ **3**

def sub $\equiv \lambda$ x. λ y. **if** (**zero?** *y*) *x* (**sub** (**pred** *x*) (**pred** *y*))

This is very similar to the definition of **add** we saw in class, except we are using **pred** for both terms, since as *y* gets closer to zero the value of the output should decrease.

Interpreters

The next two questions as you to modify the StaticCharme interpreter to support the **`begin`** special form. You do not need to modify the interpreter, but if you want to try out your solution you can download the StaticCharme interpreter from http://www.cs.virginia.edu/cs150/final/staticcharme.zip. This has been updated from the version provided in the Exam 2 comments to include the changes to **`meval`** and **`typecheck`** necessary to support begin expressions, with stub procedures for your answers to question 8 and 9.

8. (7.94 / 10) Define the **`evalBegin`** procedure to implement the evaluation rule for **`begin`**. The StaticCharme begin expression should have the same evaluation rule as the Scheme begin expression.

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

9. (**7.48 / 10**) Define the **`typeBegin`** procedure to implement the type checking rule for **`begin`**. The type of a **`begin`** expression is an error type if the expression has no subexpressions, or if any of the subexpressions are mistyped. Otherwise, the type is the type of the last subexpression.

For example,

```
StaticCharme> (begin (+ 3 #t) 4)
Error: Parameter type mismatch: expected (Number Number), given
(Number Boolean)
StaticCharme> (begin )
Error: No expressions in begin
StaticCharme> (begin (+ 3 3) #t #f (* (+ 7 8) 10))
150
```

```
def typeBegin(expr, env):
    assert isBegin(expr)
    res = CErrorType ("No expressions in begin")
    for subexpr in expr[1:]:
        res = typecheck(subexpr, env)
        if res.isError():
            return res
    return res
```

10. (6.38 / 10) a) Explain why it does not make any sense to actually add **`begin`** to StaticCharme, unless additional special forms or primitives are also added.

The begin expression is needed when expressions need to be evaluated for their side-effects. The values of the begin sub-expressions are not used, except for the last one. In StaticCharme, there are no expressions that have side-effects. So, there is no need to use a begin expression; it could always be replaced by just the last sub-expression.

b) Explain why LazyCharme does not need a **`begin`** special form (even if additional primitives and special forms were added to LazyCharme).

In LazyCharme, we could define begin as a normal procedure, so there is no need for it as a special form. We only need it as a special form in Scheme (and LazyCharme), because the order in which application subexpressions are evaluated is not determined. In LazyCharme, operand subexpressions are evaluated when their values are needed, so we could define a begin procedure that uses each value in order.