University of Virginia **Out: 16 October 2011**
cs1120: Introduction of Computing **Due: 11:01 am, Wednesday, 14 October**
Explorations in Language, Logic, and Machines

Exam 1 - Comments

Name:	***Alyssa P. Hacker***	**UVa Email ID: a p h 3 s k**

Language

1. (Average: 8.6 / 10)

(a) Give a BNF grammar that produces the language of DNA sequences. Your grammar should produce all strings of **zero or more nucleotides**, where the nucleotides are represented by elements from the set { **A**, **C**, **G**, **T** }, and no other strings.

Solution:

DNASequence ::= *Nucleotide DNASequence* | ε
Nucleotide ::= **A** | **C** | **G** | **T**

Comments:

Some common mistakes with this problem were:

1. Not allowing for zero nucleotides. For example,
   DNASequence ::= *Nucleotide MoreElements*
   MoreElements ::= *Nucleotide MoreElements* | ε
2. Allowing *Nucleotide* to be empty:
   Nucleotide :: A | C | G | T | ε.
   This will technically produce all of the correct strings and allow for zero nucleotides, but it also allows for an infinite number of empty strings in the middle of the sequence. If we try to parse a given string using this BNF, there will be infinitely many ways to parse it correctly, which can be very confusing.

(b) Give a BNF grammar that produces the language of DNA sequences that contain **one or more full codons**. Each codon is a sequence of three nucleotides. For example, your language should contain "**ACCGACTAA**" but should not contain "**ACGA**" or "" (the empty string).

Solution:

DNASequence ::= *Codon MoreCodons*
MoreCodons ::= *Codon MoreCodons* | ε
Codon ::= *Nucleotide Nucleotide Nucleotide*
Nucleotide ::= **A** | **C** | **G** | **T**

2. (6.6 / 10) For each item below, check the one answer that best characterizes the entire Scheme fragment shown. (Note: this is not asking you about what it evaluates to!)

a. **(+ 0)**

_X_ Application Expression
___ Primitive Expression
___ Procedure Expression
___ Special Form

b. **((lambda (v) v) 3)**

_X_ Application Expression
___ Primitive Expression
___ Procedure Expression
___ Special Form

Comment: Many people missed this question. **(lambda (v) v)** itself would be a Procedure Expression, but the full expression is an Application Expression (whose first sub-expression is a Procedure Expression and whose second sub-expression is a Primitive Expression). The result should be clear from following the Scheme grammar.

c. **(lambda (v) +)**

___ Application Expression
___ Primitive Expression
_X_ Procedure Expression
___ Special Form

d. **(if true false true)**

___ Application Expression
___ Primitive Expression
___ Procedure Expression
_X_ Special Form

e. **((if true (lambda (a) false) (lambda (b) true)) 3)**

_X_ Application Expression
___ Primitive Expression
___ Procedure Expression
___ Special Form

Procedures

3. (9.9 / 10) Define a procedure, **any-matches**, that takes as input three numbers and outputs **true** if any two of the numbers are equal, and outputs **false** otherwise. For example,
   (any-matches 3 7 3) should evaluate to **true**
   (any-matches 3 4 5) should evaluate to **false**
   (any-matches 7 3 7) should evaluate to **true**

Solution:

```
(define (any-matches a b c)
    (or (= a b) (= a c) (= b c)))
```

4. (6.3 / 10) Define a procedure, **is-composite?**, that takes as input a natural number, and outputs **true** if that number is composite, and outputs **false** otherwise. A natural number, *n*, is composite if it is divisible by some number that is greater than 1 and less than *n*. You do not need to worry about efficiency in your solution – a simple, slow, and correct procedure is worth full credit.

   You may use the **is-divisible?** procedure defined below in your **is-composite?** definition. The **is-divisible?** procedure takes two inputs, and evaluates to **true** if the first input is divisible by the second input, and **false** otherwise.

```
(define (is-divisible? v d) (= (modulo v d) 0))
```

Solution: The easiest way to solve this was to define a helper procedure:

```
(define (is-composite? n)
  (if (= n 1) false (composite-helper n (- n 1) ) ; or (floor (sqrt x)) instead of (- x 1)

(define (composite-helper n v)
   (if (= v 1) false
       (if (is-divisible? n v)
           true
           (composite-helper n (- v 1))
```

Another good approach is to use the loop procedure from Class 16/17:

```
(define (is-composite? n)
 (loop 2 false
       (lambda (index) (<= index (floor (sqrt n)))) ; test
       (lambda (index) (+ index 1)) ; update
       (lambda (index res) (or (is-divisible? n index) res)))) ; combine results
```

5. (9.2 / 10) Define a procedure, **all-positive?**, that takes as input a list. The output should be true if the all the elements in the input list are positive; otherwise, the output should be false. For example,

    (all-positive? (list 2 4 6 8)) should evaluate to **true**
    (all-positive? (list 2 0)) should evaluate to **false**
    (all-positive? null)) should evaluate to **true**

Solution:

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

A more clever approach is to use **is-pure?** from question 6:

```
(define (all-positive? p) (is-pure? positive? p))
```

6. (7.6 / 10) Define a procedure, **is-pure?**, that takes as input a list and a test procedure. The output should be true if the result of applying the test procedure to each element in the list is true; otherwise the output should be false. For example,

    (is-pure? even? (list 2 4 6 8)) should evaluate to **true**
    (is-pure? even? (list 2 4 6 8 9)) should evaluate to **false**
    (is-pure? (lambda (v) false) null)) should evaluate to **true**

Solution:

```
(define (is-pure? f p)
  (if (null? p)
      true
      (if (f (car p))
          (is-pure? f (cdr p))
          false)))
```

We can make a shorter definition using the **or** and **and** special forms:

```
(define (is-pure? f p)
  (or (null? p)
      (and (f (car p)) (is-pure? f (cdr p)))))
```

Several people were tempted to use **map** for this, but it is fairly tricky to define this using **map**. The most obvious way does not actually work:

```
(define (is-pure? f p)
```

```
    (apply and (map f p)))
```

The reason it doesn't work is because **and** is a special form, not a procedure, so it cannot be used as the input to **apply**. One way to implement it using **map** would be:

```
(define (is-pure? f p)
   (if (= (apply * (map (lambda (v) (if (f v) 1 0)) p)) 1) true false))
```

7. (8.0 / 10) Define a procedure, **factors**, that takes as input a number and outputs a list containing all the non-trivial factors of that number. A natural number *b* is a factor of *a* if *a* is divisible by *b*. The trivial factors (1 and the input number) should not be included. You may use the **is-divisible?** procedure from question 4 in your definition. For example,

   (factors 6) should evaluate to the list **(2 3)**
   (factors 7) should evaluate to the empty list
   (factors 1120) should evaluate to the list
   (2 4 5 7 8 10 14 16 20 28 32 35 40 56 70 80 112 140 160 224 280 560)

   Hint: define a **factors-helper** procedure and use it do define **factors** as:

```
(define (factors n)
   (list-reverse (factors-helper (- n 1) n)))
```

   (If you use this, it is only necessary to show the definition of **factors-helper**.)

Solution:

One way is to define the helper procedure is:

```
(define (factors-helper t n)
  (if (< t 2)
      null
      (if (is-divisible? n t)
         (cons t (factors-helper (- t 1) n))
         (factors-helper (- t 1) n))))
```

One common mistake here is getting the correct base cases. Since **t** is a number, not a list, this shouldn't be when **t** is **null**. Instead, the base case is when **t** is 1, or equivalently whether **t** is less than 2. We don't want to use **(= 0 t)**, since this would include 1 as a factor, and only non-trival factors should be included according to the problem definition.

If you didn't use the hint, a good way to define factors without using a helper procedure is to use **list-filter** (from Chapter 5):

```
(define (factors n)
```

(list-filter (lambda (x) (is-divisible? n x)) (cdr (intsto (- n 1))))

The **(cdr (intsto (- n 1)))** expression evaluates to a list of the numbers from **2** to **(- n 1)**. Then, we use **list-filter** to keep only the elements in that list that **n** is divisible by.

8. (6.75 / 10) Lefty O'Doul proposes replacing the standard *List* structure, with a new structure he calls a *Tsil* (the t is silent, of course) defined as:

   A *Tsil* is either (1) **llun** or (2) a Pair whose first cell is a *Tsil*.

   where **llun** is a new special built-in value (analogous to **null**), and the built-in procedure **llun?** (analogous to **null?**) that takes one value as input and outputs **true** if that value is **llun**, and **false** otherwise.

   Define a procedure, **tsil-map**, analogous to **list-map**, that takes as input a procedure and a *Tsil*, and produces as output a *Tsil* that contains as elements the result of applying the input procedure to each element of the input *Tsil* in the same order.

Solution:
A Tsil is essentially a list that works in the opposite direction. Recall that we defined a List as:

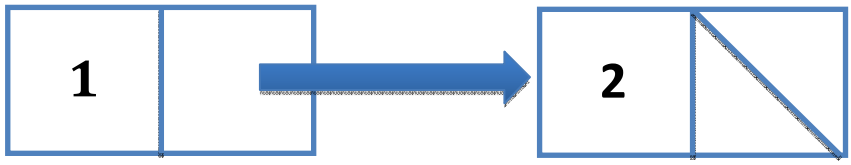
$$List ::= Pair \mid null$$
$$Pair ::= element\ List$$

From the problem statement, it is clear that a Tsil is similar to a normal list but just goes in the opposite direction (indicated by definition or the carefully chosen name). We can build a Tsil BNF:

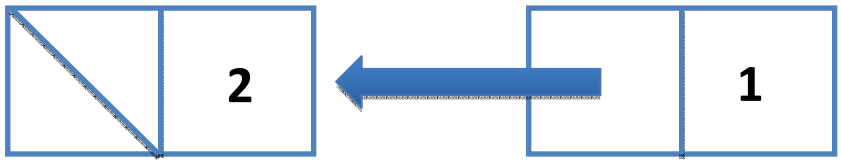
$$Tsil ::= llun \mid Pair$$
$$Pair ::= Tsil\ element$$

It may also help to draw a diagram.

List



Tisl



Before answering the question, let's consider how to build **tsil-sum**, analogous to **list-sum**:

```
(define (list-sum p)
  (if (null? p) 0
      (+ (car p) (list-sum (cdr p)))))
```

To do that with a *Tsil* we work in the other direction:

```
(define (tsil-sum p)
  (if (llun? p) 0
      (+ (cdr p) (tsil-sum (car p)))))
```

Now that we have a simple Tsil traversal works, we can modify **list-map** to **tisl-map** fairly easily:

```
(define (list-map func p)
 (if (null? p) null
    (cons (func (car p)) (list-map func (cdr p)))))
```

becomes:

```
(define (tsil-map func p)
 (if (llun? p) llun
     (cons (tsil-map func (car p)) (func (cdr p)))))
```

Note the order of the inputs to **cons** is reversed from the way it would be for a List.

The point of this problem was not to confuse you with Tsils, but to see how well you really understand Lists.

The most common mistake was:

```
(define (tsil-map func p)
  (if (llun? p) llun
      (cons (func (car p)) (tsil-map func (cdr p)))))
```

It is a near identical copy of the **list-map** function and will not produce a Tsil as the first element of each pair is a value and not a Tsil as defined. It is also worth mentioning that the function should be evaluated on the **cdr** of the Tsil, not the **car** (which is a Tsil, not a single element).

Machines

9. (7.9 / 10) Design a Turing Machine that takes as input a tape in the form *a*=*b*# where *a* and *b* are sequences of zero or more symbols where each symbol is either **0** or **1**. The output should be **#1** (this can be anywhere on the tape, but there should be exactly one **#** on the output tape) if *a* and *b* are exact matches, and **#0** otherwise. Your answer should first describe in English (or Schemish) a high-level description of how your machine works. Then, you should provide a precise description of your machine, either as a complete list of transition rules or as a diagram.

Solution: There were several acceptable answers. As mentioned in the posted comments, it was fine to assume that the input string were the same length. The machine we need is very similar to the machine we saw in class for less than comparison.

The most common errors were not marking a start state, leaving out important inputs, or not having enough states to keep track of everything. If the English description was good but machine was not quite perfect you still received most of the points for this question.

The Turing machine should start at the left of tape marked by **#** then move right and begin to read *a*. The first digit of *a* is read and there are two states for if it was **1** or **0**, replace it with a new symbol (e.g., **x**). If there was no digit but an **=** we can move to the end of the tape and output **#1** (the strings were equal, taking advantage of the assumption that the input lengths were the same).

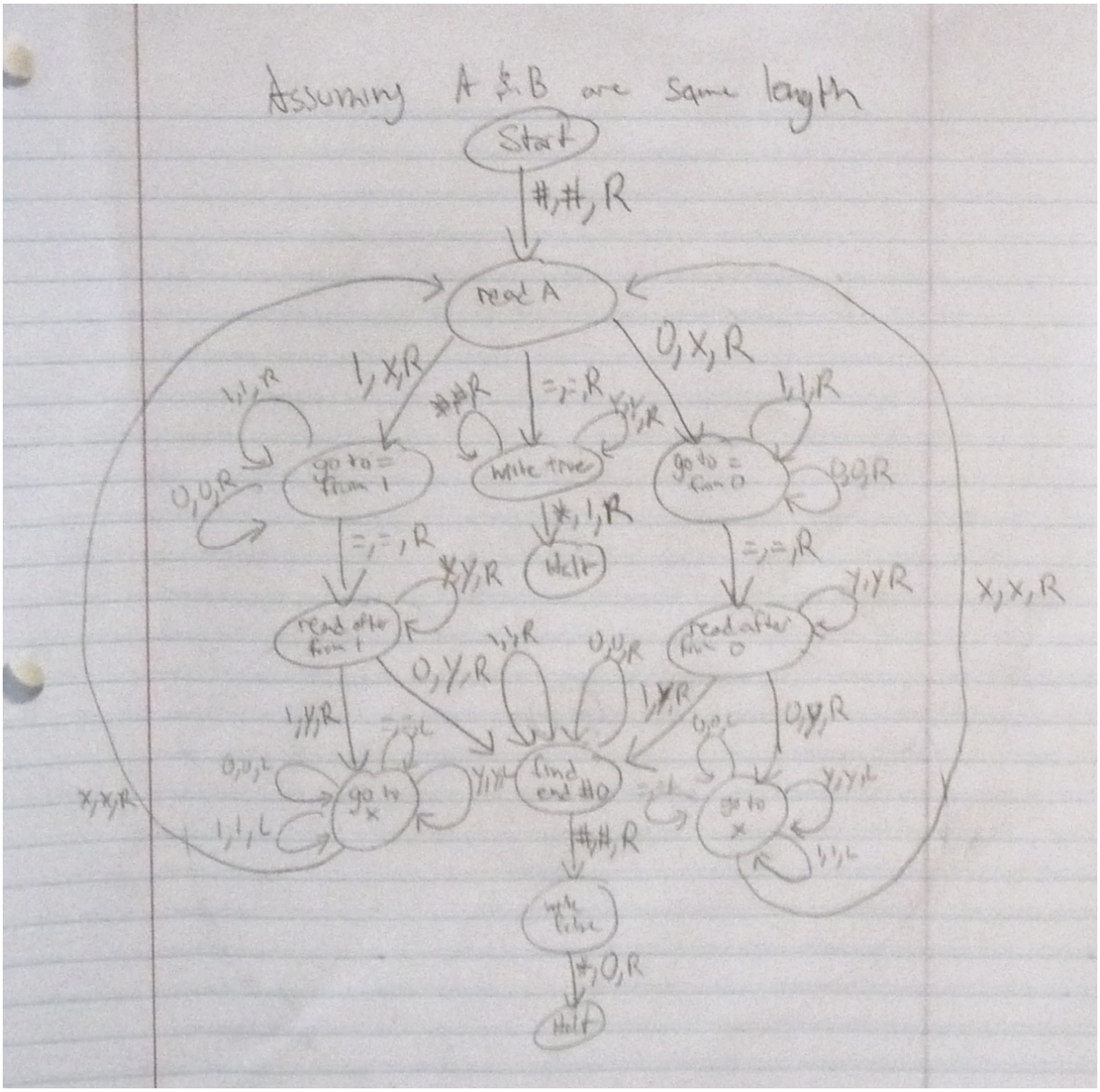
Otherwise, move right until the **=** sign and check the digit after the **=**. If it is different from what was read earlier (which we know from the state we are in) we know we can stop checking, move to the end of the tape and output **#0**. If they are the same replace the value with a new symbol (e.g., **y**), then move back to the last **x** that was written in *a*. Using a different symbol makes it easier to search back for the rightmost **x**, instead of having to look for the **=** on the way (like the example we did in class).

Move right and do the same steps again until all values in *a* have been replaced with an **x**. On your last run you will stop on **=** and know that all value have been the same. Then move to the end of the tape and write **#1**.

We can describe the machine using state transition rules:

Current State	Read	Write	Next State	Move
Start	#	#	read A	R
ReadA	0	x	Goto=from0	R
ReadA	1	x	Goto=from1	R
ReadA	=	=	Writetrue	R
Goto=from0	1	1	Goto=from0	R
Goto=from0	0	0	Goto=from0	R
Goto=from0	=	=	Readafterfrom0	R
Goto=from1	1	1	Goto=from1	R
Goto=from1	0	0	Goto=from1	R
Goto=from1	=	=	Readafterfrom1	R
Readafterfrom0	y	y	Readafterfrom0	R
Readafterfrom0	1	y	Findend	R
Readafterfrom0	0	y	Gotox	R
Readafterfrom1	y	y	Readafterfrom1	R
Readafterfrom1	1	y	Gotox	R
Readafterfrom1	0	y	Findend	R
Findend	1	1	Findend	R
Findend	0	0	Findend	R
Findend	#	#	Writefalse	R
Gotox	1	1	Gotox	L
Gotox	0	0	Gotox	L
Gotox	=	=	Gotox	L
Gotox	y	y	Gotox	L

Gotox	**x**	**x**	ReadA	R
writefalse	*****	**0**	halt	halt
writetrue	**y**	**y**	writetrue	R
writetrue	**#**	**#**	writetrue	R
writetrue	*	**1**	halt	halt



Tandem Repeats

A tandem repeat in a genome is a sequence of nucleotides that repeats two or more times consecutively. Tandem repeats are common throughout the genome, and have many important uses including genetic fingerprinting (this is what is used in DNA tests to identify criminals, since the number of repetitions for certain sequences is highly variable across humans and easy to detect) and diagnosing diseases (for example, in a normal FMR-1 gene the sequence CGG repeats 6-54 times consecutively, but repeats over 200 times in patients with a form of autism). In questions 10 and 11, you will define procedures for counting tandem repeats.

10. (9.0 / 10) Define a procedure, **contains-matching-prefix**, that takes as input two lists, *p* and *s*. The output should be **true** if the sequence of elements at the beginning of *p* exactly matches the complete sequence of elements in *s*. Otherwise, the output should be **false**.

    For example,
    (contains-matching-prefix (list 1 2 3 4) (list 1)) should evaluate to **true**
    (contains-matching-prefix (list 1 2 3 4) (list 1 2 4)) should evaluate to **false**
    (contains-matching-prefix (list 1 2) (list 1 2)) should evaluate to **true**
    (contains-matching-prefix (list 1 2) (list 1 2 3)) should evaluate to **false**
    (contains-matching-prefix (list 1 2) null) should evaluate to **true**

Solution:

```
(define (contains-matching-prefix p q)
  (or (null? q)
      (and (eq? (car p) (car q))
           (contains-matching-prefix (cdr p) (cdr q)))))
```

Without using **or** or **and**:

```
(define (contains-matching-prefix p q)
  (if (null? q)
      true
      (if (eq? (car p) (car q))
          (contains-matching-prefix (cdr p) (cdr q))
          false)))
```

11. (5.0 / 10) Define a procedure, **count-tandem-repeats**, that takes as input a list, *p*, and a number, *n*. The output should be the total number of times the first *n* elements of *p* are consecutively repeated at the beginning of *p*. Repetitions may not overlap.

    For example,
    (count-tandem-repeats (list 1 2 1 2 1 3) 2) should evaluate to **2**
    (count-tandem-repeats (list 2 2 2 2 2 2) 2) should evaluate to **3**
    (count-tandem-repeats (list 1 2 3 1 2 1 2 3) 3) should evaluate to **1** [corrected]

You should feel free to define any helper procedures you think are useful, as well as to use any procedures defined in the course book or problem sets.

Solution:

Here is the solution we discussed in Class 22:

```
(define (list-prefix p n)
  (if (= n 0)
      null
      (cons (car p) (list-prefix (cdr p) (- n 1)))))

(define (count-prefix-repeats p q)
  (if (contains-matching-prefix p q)
      (+ 1 (count-prefix-repeats ((n-times cdr (length q)) p) q))
      0))

(define (count-tandem-repeats p n)
  (count-prefix-repeats p (list-prefix p n)))
```

Note that this solution is not defensive – it will produce errors if **n** is greater than the length of **p**, and will run forever if **n** is **0**.

12. Do you feel your performance on this exam will fairly reflect your understanding of the course material so far? If not, explain why.

Comments: Most people thought the exam was fair, but that it was difficult to write procedures without being able to use an interpreter or work with other people. It is definitely hard to get things exactly right without being able to try running your code in an interpreter, but it is also the case that if you understand things well you should be able to get close to a correct answer (and definitely enough for full credit) without needing to use an interpreter.

Needing to work with other people is a different issue. For most people, it is a very good learning experience to be able to do the problem sets with other people, but it may also sometimes be hard to know if you could answer the questions on your own and sometimes it is much easier to understand an answer you've had help to work out than to construct one on your own. We will still have a open collaboration policy on future problem sets, but if you felt like you were not able to do as well as you would like on the exam because you are not learning enough how to solve problems on your own, I would encourage you to try doing more of the problem set questions without help first (but still to ask for help when you are really stuck, just to try harder to get yourself unstuck first).

13. How long did it take you to complete this exam?

Comments: The average was just under 5 hours, which is definitely longer than I intended, but there was a lot of variance here. Three people reported 2 hours or less (and all of them got at least a 90+ on the exam), four people reported over 10 hours, and most people reported 3-5 hours. Since it was an open book exam, it is a bit difficult to interpret these numbers, since some of the time was probably spent on things that would normally be pre-exam preparation.

14. What topics to you hope to see in the remainder of the course?

Comments: The two most common answers here were other programming languages, the Internet, and more on encryption. We will soon start using Python (starting with Problem Set 6), and will also use Java towards the end of the class, so you will definitely get some experience with other programming languages soon. We will have probably two classes about networking and the Internet, as well as some on building web applications later. I'm not sure how much more we'll do on encryption (although my research group works on this, and I'm definitely happy to talk about it more), but we will certainly need to cover "Trick-or-Treat" protocols in time for Halloween.

15. Do you trust your classmates to follow the honor expectations in this class? (Feel free to write comments instead or in addition to checking one or more of the options.)

    13 Yes, I trust them completely.

    2 **[Write in something between trust completely and vast majority.]**

    29 I worry that there may be a few transgressions, but I believe the vast majority of the class is honorable and it is fair and beneficial to rely on this.

    1 I think this class places too high a burden on students' honor, and there are enough dishonorable students that it is unfair on the honorable students.

    0 I have reason to suspect that other students violated the honor policy on problem sets.

    0 I have direct knowledge of other students violating the honor policy on problem sets.

    0 I have reason to suspect that other students violated the honor policy on this exam.

    0 I have direct knowledge of other students violating the honor policy on this exam.