CS 571
Midterm Solution

Oct 24
100 points **Open book, open notes**
Time: 85 minutes **No Electronic Devices**

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.
Justify all answers
Please write your answers only within the provided exam booklets.

There are a total of 6 questions.

1. A Unix *path* is defined in stages as follows:

   Path Component A *path component* is a sequence of one-or-more characters which does not contain any occurrences of the `/` or `NUL` characters.

   Relative Path A *relative path* is a sequence of one-or-more path components separated by a single `/` character.

   Absolute Path An *absolute path* consists of the `/` character optionally followed by a relative path.

   Unix Path A Unix path is either an absolute or a relative path.

   Provide a regex for Unix paths. You may use `\/` to represent the regex matching `/` and `\0` to represent the regex matching the `NUL` character. You should factor (using intermediate named regex's) or format your answer to ensure that it is readable and understandable. *10-points*

```
path-component = [^\/\0]+
rel-path = path-component ( \/ path-component )*
abs-path = \/ rel-path?
path = abs-path | rel-path
```

Alternately, without using any intermediate named regex's:

```
\/ | \/? [^\/\0]+ (\/ [^\/\0]+)*
```

2. An *X-expression* is either an atom, or two X-expressions surrounded by parentheses and separated by `.` (period), or a sequence of one-or-more X-expressions surrounded by parentheses.

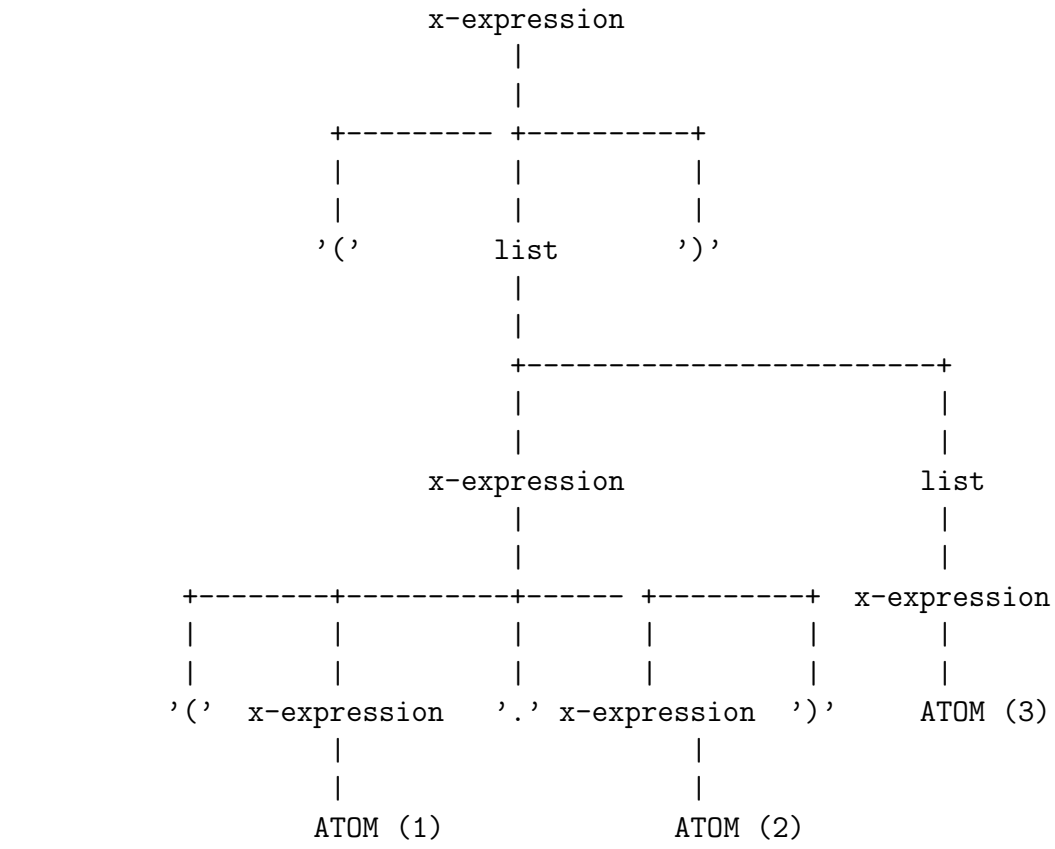
   (a) Give a grammar for *X-expressions*. You should use the set of terminals { `ATOM`, `'('`, `')'`, `'.'`}.

   (b) Use your grammar to provide a *parse tree* for the X-expression `( (1 . 2) 3)`, where the integers will be scanned as `ATOM` terminals.

20-points

(a)
```
x-expression
  : ATOM
  | '(' x-expression '.' x-expression ')'
  | '(' list ')'
  ;
list
  : x-expression
  | x-expression list
  ;
```

(b)



3. Given the following program in a language which supports nested functions as well as both lexically-scoped (indicated using a `lex` declaration) and dynamically-scoped variables (indicated using a `dyn` declaration):

```
lex lex1 = 1;  //lexically scoped var lex1
dyn dyn1 = 2;  //dynamically scoped var dyn1

f(param_f) { //define function f with single parameter param_f
  lex lex1 = 3;
  dyn dyn1 = 4;

  g(param_g) { //define function g with single parameter param_g

    return lambda(x) { return x + param_f*param_g + lex1*dyn1; };
```

```
  }

  return g(dyn1);

}

print f(6)(7);
```

What will be printed by the above program. Please remember to justify your answer. *20-points*

When the anonymous function is defined, the parameters and lexically scoped variables are bound. So at the point of definition of the anonymous function, we have `param_f` with value 6, `param_g` with value 4 (the value of `dyn1` when `g()` is called), and `lex1` with value 3; hence the function returned is:

```
lambda(x) { return x + 6*4 + 3*dyn1; }
```

When the anonymous function is run, the parameter `x` is bound to 7 and `dyn1` is bound to its current dynamic value 2. Hence the function returns `7 + 6*4 + 3*2` which is 37. Hence 37 will be printed.

4. Describe how you would represent a CFG using basic S-expressions.

   (a) Specifically, describe how you would use S-expressions to represent *terminals*, *non-terminals*, *rules* and *grammars*.

   (b) Show your representation for the example CFG:

```
s : a
  | b
  ;
a : A
  ;
b : B
  ;
```

   (c) Describe how you would hide the details of your representation from users of your representation.

15-points

(a) One possible representation:

Terminal Symbol A terminal symbol T would be represented as the pair `(term . T )` for some atom T.

Non-Terminal Symbol A non-terminal symbol N would be represented as the pair `(non-term . N )` for some atom N.

Rule A rule would be represented by a pair where the first element of the pair is a non-terminal symbol representing the LHS of the rule and the second element of the pair is a list containing the symbols on the RHS of the rule.

Grammar A grammar would be represented by a pair where the first element of the pair is a non-terminal representing the start-symbol of the grammar and the second element of the pair would be a list of the rules for the grammar.

(b)
```
((non-term . s)
 ((non-term . s) (non-term . a))
 ((non-term . s) (non-term . b))
 ((non-term . a) (term . A))
 ((non-term . b) (term . B)))
```

Note the simplification above by having the second element of both grammars and rules be lists.

(c) The representation would be hidden between a set of accessor functions. Using Scheme notation, we could have something like the following:

```
;;return the symbol representing the start-symbol for the grammar
(g-start-symbol grammar)

;;return a list of rules for non-terminal symbol non-term in grammar.
(g-rules grammar non-term)

;;return the non-terminal symbol for rule.
(g-lhs rule)

;;return a list of symbols representing the rhs of rule.
```

```
(g-rhs rule)

;;Return #f iff sym does not represent a non-terminal symbol
;;(non-terminal? sym)

;;Return #f iff sym does not represent a terminal symbol
;;(terminal? sym)
```

This would allow hiding the representation. So the representation provided in part (a) could be changed to a more efficient indexed representation, but the use of the representation would remain unchanged.

5. Write a Scheme function `(count-atoms` *s-exp*`)` which counts the number of atoms (non-pairs) in a maximally simplified representation of S-expression *s-exp* (i.e., the `'()` terminating proper lists should be ignored).

   Example log:

```
> (count-atoms 'a)
1
> (count-atoms '(a ()))
2
> (count-atoms '(a . ()))
1
> (count-atoms '(a b . ()))
2
> (count-atoms '(a b ()))
3
>
```

   20-points

   The key to this question is to decompose the *s-exp*.

   - If *s-exp* is not a pair, then it is an atom which should be counted; hence the function should return 1.

- If *s-exp* is a pair, then the number of atoms is the number of pairs in the `car` plus the number of pairs in the `cdr`. However, when the `cdr` is null, then the number of pairs in the `cdr` should be counted as 0.

This results in the function:

```
(define (count-atoms s-exp)
  (if (pair? s-exp)
      (+ (count-atoms (car s-exp))
      (let ([tail (cdr s-exp)])
         (if (null? tail)
             0
             (count-atoms tail))))
      1))
```

6. Discuss the validity of the following statements:

   (a) All evaluable Scheme expressions are S-expressions.

   (b) All S-expressions are evaluable Scheme expressions.

   (c) Assuming that a stack grows towards high memory, then within a stack frame for a function, the parameters to the function will be located at higher addresses than the local variables of the function.

   (d) Languages which allow recursive functions **must** use stack allocation for function parameters and local variables.

   (e) If a language requires a left-associative binary operator $\oplus$ to have the same precedence as a right-associative binary operator $\otimes$, then the language is ambiguous.

15-points

(a) An evaluable Scheme expression is either an atom or the application of a function to zero-or-more scheme expressions, both of which can be represented as S-expressions. Hence the statement is **true**.

(b) Not all S-expressions are evaluable Scheme expressions because an evaluable Scheme expression which is a pair requires something which represents a function as the first element of the pair. For example, `(1 2)` is an S-expression but is not an evaluable Scheme expression (note that `'(1 2)` which is the same as `(quote (1 2))` is an evaluable Scheme expression). Hence the statement is **false**.

(c) The parameters will be pushed onto the stack before the local variables. Since the stack grows towards high memory, the parameters will be pushed at lower addresses than the local variables. Hence the statement is **false**.

(d) Languages which allow recursive functions **must** use stack allocation for function parameters and local variables.

Languages which allow recursive functions definitely cannot use static allocation for function parameters and local variables. However, instead of stack allocation, they can use heap allocation for function parameters and local variables; this may not be particularly efficient but will always work (in fact, this may be necessary when a parameter or local variable value escapes the function activation). Hence the statement is **false**.

(e) When a binary operator is left-associative, then it's left-operand can contain a non-parenthesized expression with a top-level operator of the same or higher precedence, whereas it's right operand can contain a non-parenthesized expression with a top-level operator with strictly higher precedence. A similar statement holds for a right-associative operator.

Consider $x \otimes y \oplus z$. This expression could represent either $x \otimes (y \oplus z)$ or $(x \otimes y) \oplus z$ without violating the above associativity constraint. So the same expression can represent multiple syntactic structures and the language is ambiguous. Hence the statement is **true**.