Fall 2018 Programming Languages Qualifying Exam

CODE_____

This is a closed book test.

Correct, clear and precise answers receive full marks

Please start a new page for each question.

There are five (5) questions, each 20 points

1) Language Dynamic Memory

Consider the following code segment. Explain what happens at runtime and what sort of runtime support may be used to address the issue. You should list at least two different distinct methods for addressing the identified issue. Note: An infinite loop is not the core issue.

```
String S;
while(true) S = new String("hello");
```

The runtime system could experience out of memory (on the heap). The runtime system can address this by

1. ***ignoring the problem and allowing the operating system to make the program have an exception for requesting memory not available (usually a runtime error).***
2. ***Maintaining an additional set of memory in the heap that keeps track of references. When reference count is 0, then put back in the free list. Requires that runtime system also perform some sort of garbage collection***
3. ***When memory is exhausted, have the memory allocation algorithm use some form of mark/sweep which starts from variables in the stack and walking references from there into the heap, marking all elements reachable. Then take the remaining elements in the heap and garbage collect them***

*2) **L**anguage Runtime Support*

How do languages support threaded programs? Be precise about the support needed for thread creation and thread termination. Be clear about the typical start point (including the rational) of a thread, how data is shared at startup of the thread and how resultant data is shared at thread

termination.

Most runtime systems that support multiple threads must partition the runtime stack into individual stacks. When a thread is started, it starts at the start of a function/subroutine call. Parameters passed in come via reference pointer to a record of data (anonymous tuple) that can reside in the heap, stack or data segments.

Since threads can run asynchronously, collecting return values directly from the function/subroutine call is problematic. To alleviate this problem the programmer either places the return value in a heap tuple or can write to a common global data element (usually found in the data segment).

3) *Regular Expressions*

Consider the set of strings containing the letters {**a,b**} where each element of the set has exactly one substring which contains **aaa.**

a) Provide a regular expression which represents this set of strings

The key to the answer is to generate strings with none, one or two as and then add in the aaa at the end

b*((a|aa)b+)*aaa(b+(a|aa))*b*

b) Consider the regular language (a|b)*bab. Create mechanically an Non Deterministic Finite Automata (NFA) for the grammar.

TBD –READER SHOULD BE ABLE TO SEE PROPER RESULT

c) From your NFA in part b, create a Deterministic Finite Automata.

TBD – READER SHOULD BE ABLE TO DETERMINE PROPER RESULT

4) *Language Design and Implementation*

Consider Method Overloading. Describe how a language implements method overloading. Be specific on what data structures are needed and how they are used during compilation time (for compiled languages) and during run-time for interpreted languages.

To manage method overloading, the symbol table must be enhanced to allow the same named method to be in the symbol table, and is differentiated with the number and types of the parameters. During compilation, whenever a method ID is encountered, the compiler most also determine the number and types of the method. The compiler then uses the enhanced symbol table to determine if the appropriate method is defined. Same applies for interpreted languages.

5) Functional Programming

a) write the function **append** which has the format
(append L1 L2) and returns a list where the elements of L1 are appended to the front of L2 in the same order

```
(define (append L1 L2) ;; assumes L1 is a null ending list
  (cond ((null? L1) L2)
        (else (cons (car L1) (append (cdr L1) L2)))))
```

b) Using append, write **flatten** which takes a single list and creates a new list with no embedded lists. For example :

(flatten '((1 2) ((3) (((4)))))) == '(1 2 3 4)

```
(define (flatten L)
  (cond (( null? L) '())
        ((atom? (car L)) (cons (car L) (flatten (cdr L))))
        (else (append (flatten (car L)) (flatten (cdr L))))))
```