COMP 524 Final

Fri, Dec 6, 2019, 12-3pm

Instructions
1. In all questions, assume the implementation of Lisp discussed in class, which is not the same as C-Lisp.
2. You can use defun or setq to implement functions unless the problem constrains you otherwise.
3. Please spread out and try and sit in alternate seats.
4. This is a closed book exam.
5. Write in the space provided. Write on the opposite side of a question page if there is not enough space to solve a problem. Use the overflow pages only if necessary.
6. There are:
   • 12 numbered pages including this one.
   • 4 questions.
7. The exam should be printed double sided – in case it is not, please do not write on the blank unnumbered pages.
8. If you need to make any assumptions to clarify a problem, write your assumptions down. Only reasonable assumptions get full credit.
9. You will get partial credit, so write down part of the solution even if you cannot develop the complete solution.
10. You will not be penalized for errors in syntax as long as your algorithm is understandable.
11. You are not required to comment or annotate any code you write, but may get partial credit if you write appropriate comments/annotations but incorrect code.
12. If you cannot remember the name of a predefined language operation or type, invent it, and identify what it does.
13. Your solutions may be graded not only for correctness but also for style and efficiency.
14. Please inform the proctor of anything in the exam that you think is a mistake.
15. Walk over to the proctor to ask questions so that you do not disturb others.
16. If you do not understand some English word, do not hesitate to ask the proctor. Naturally, you are expected to know the computer science terms defined in class.
17. Write clearly using a pen/pencil with a dark color – we will be scanning your exams.
18. Do not unstaple the exam. Unstapled exams may not be graded.

NAME , ONYEN (ALL LARGE LEGIBLE CAPITALS, YOUR EXAM MAY NOT BE GRADED OTHERWISE)

(CAP. NAME) __________________________________________(CAP. ONYEN) __________________________

Pledge: I have neither given nor received unauthorized aid on this exam.
(signed) __________________________________________

For survey purposes, please indicate the time at which you turned in the exam. __________
1. [30 pts.] **Understanding Recursive Lambda Execution**

   a) Consider the following function:

   ```lisp
   (setq f1
     (lambda (L)
       (cond
         ((eq L Nil) NIL)
         (T  (cons
               (atom (car L))
               (funcall f1 (cdr L)))
             )
           )
         )
     )
   )
   
   Give the output of:
   (funcall f1 (list 1 (cons 2 3) 4))
   
   Output F1 Call: _______________________________________
   
   b) Now consider the following similar function:

   ```lisp
   (setq f2
     (lambda (L)
       (cond
         ((eq L Nil) NIL)
         (T  (cons
               (not (car L))
               (funcall f2 (cdr L)))
             )
           )
         )
     )
   )
   
   Give the output of:
   (funcall f2 (list T T NIL))
   
   Output F2 Call: _______________________________________
EXTRA SPACE but please try and fit your answer in the assigned space
2. [40 PTS.] **HIGHER-ORDER FUNCTIONS**

a) Here you will write a higher-order function, filterList, that takes two arguments, the first one a function and the second a list. filterList should capture the commonality between f1 and f2 defined in Q1. In particular:

\[
\text{(funcall filterList \ldots L)}
\]

should behave the same as

\[
\text{(funcall f1 L)}
\]

where \ldots is an appropriate function you pass to filterList to simulate f1. For example:

\[
\text{(funcall filterList \ldots \text{(list 1 \text{\,cons\,} 2 3) 4})}
\]

should give the same output as:

\[
\text{(funcall f1 \text{(list 1 \text{\,cons\,} 2 3) 4})}
\]

Similarly:

\[
\text{(funcall filterList \ldots L)}
\]

Should behave the same as:

\[
\text{(funcall f2 L)}
\]

where \ldots is an appropriate function you pass to filterList to simulate f2. For example:

\[
\text{(funcall filterList \ldots \text{(list T T NIL)}})
\]

should be the same as:

\[
\text{(funcall f2 \text{(list T T NIL)}})
\]

Naturally, the two arguments you pass to filterList in the above cases should be different and filterList should make use of both of its arguments. It should, of course, embody the commonality but not the differences in f1 and f2.

As our implementation of the Lisp interpreter supported it, you can assume that you can funcall basic operations by quoting them. For example,

\[
\text{(funcall (quote +) 2 3)}
\]

Is the same as:

\[
\text{(+ 2 3)}
\]

The code of the two original functions is reproduced below:

\[
\begin{align*}
\text{(setq f1} & \text{ (lambda (L)} \text{ (cond}} \\
& \text{ \text{((eq L Nil) NIL)} \text{) \text{ (T}} \text{ \text{(cons}} \\
& \text{ \text{(atom (car L))} \text{ (funcall f1 (cdr L))}} \text{)) \text{) \text{) \text{) \text{))))}} \\
\text{(setq f2} & \text{ (lambda (L)} \text{ (cond}} \\
& \text{ \text{((eq L Nil) NIL)} \text{) \text{ (T}} \text{ \text{(cons}} \\
& \text{ \text{(not (car L))} \text{ (funcall f2 (cdr L))}} \text{)) \text{) \text{) \text{) \text{))))}}
\end{align*}
\]
Implementation of filterList.

b) As mentioned above:
   \((\text{funcall \ filterList \ ... \ L})\)
   should behave the same as
   \((\text{funcall \ f2 \ L})\)

   What does \(...) need to be to make this true?
   
   _______________________________________________________________________

c) For this part of the question, you can assume that your implementation of filterList is correct.
   Use the curry operator on filterList to define a new function, \(\text{curriedF1}\), that has the same
   behavior as \(f\), that is:
   \((\text{funcall \ f1 \ L})\)
   Is the same as:
   \((\text{funcall \ curriedF1 \ L})\)

   _______________________________________________________________________

   _______________________________________________________________________

   _______________________________________________________________________
3. [40 pts.] Lambda vs Function Expressions

Consider the following code, which we will refer to as the loaded code:

```
(setDeepCopyFunctionEnvironment ...)  
(setq x 10)  
(setq y 20)  
(setq lambdaPrint  
    (lambda (deltaX deltaY)  
      (print (+ x deltaX))  
      (print (+ y deltaY))  
    )  
)  
(setq functionPrint  
    (function  
      (lambda (deltaX deltaY)  
        (print (+ x deltaX))  
        (print (+ y deltaY))  
      )  
    )  
)  
```

We will consider two variations of this code in which the ... in the first call are T and nil, respectively. In the three parts of this question, you will give the output of the following alternative, independent uses of the (two variations) of the loaded code. You should give the output of the print functions called, and also the output the Lisp interpreter produces after processing the funcall. You need not give the output the Lisp interpreter produces after processing the setq calls.

(a) Calling lambdaPrint with DeepCopyFunctionEnvironment T: Assume the variation of the loaded code in which ... is T, that is, the first expression in it is:

```
(setDeepCopyFunctionEnvironment T)
```

Give the output of:

```
(setq x 20)  
(setq y 40)  
(funcall lambdaPrint 1 2)
```

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________
(b) **Calling functionPrint with DeepCopyFunctionEnvironment T:** Assume the variation of the loaded code in which ... is T, that is, the first expression in it is:

```
(setDeepCopyFunctionEnvironment T)
```

Give the output of:
```
(setq x 20)
(setq y 40)
(funcall functionPrint 1 2)
```

(c) **Calling functionPrint with DeepCopyFunctionEnvironment Nil:** Assume the variation of the loaded code in which ... is Nil, that is, the first expression in it is:

```
(setDeepCopyFunctionEnvironment Nil)
```

Give the output of:
```
(setq x 20)
(setq y 40)
(funcall functionPrint 1 2)
```
4. [50 pts.] **EAGER EVALUATION AND BASIC JOINER**

The purpose of this question is to determine your understanding of the concept of threads, and how the basic joiner and eager evaluation, as discussed in class, work.

Assume the class, BasicJoiner, exists that implements the Joiner interface discussed in class, and it has been replaced by the following class, which simply traces the execution of the constructor and methods of BasicJoiner.

```java
public class TracingJoiner extends BasicJoiner{
    public TracingJoiner(int numThreads) {
        super(numThreads);
        System.out.println("Joiner Created:" + numThreads);  
    }
    @Override
    public synchronized void finished() {
        System.out.println("Finished called.");
        super.finished();
        Tracer.info(this, "Finished returned.");
    }
    @Override
    public synchronized void join() throws InterruptedException {
        System.out.println("Join called.");
        super.join();
        System.out.println("Join returned.");
    }
}
```

Assume that the main thread is named m, and the i\textsuperscript{th} thread created to do eager evaluation is named t\textsubscript{i}. As we saw in class, the basic parameterless function, printThread, executed by a thread named t\textsubscript{i}, prints and returns the name of the thread:

```plaintext
> (printThread)
t\textsuperscript{i}
t\textsuperscript{i}
```

Your task in this question is to give output produced by executing the following code:

```lisp
(setEvalMode "EAGER")
(setEagerPool nil)
(list (printThread) (printThread) (printThread))
```

You need to give the tracing output produced by TracingJoiner, the output of the printThread functions called, and also the output the Lisp interpreter produces after processing the list call. You need not show the result of evaluating the setEvalmode and setEagerPool calls.

**Because of the nature of threading, multiple but not arbitrary outputs can be produced by the same sequence of instructions.** To show your understanding of the underlying concepts of threading, basic joiner and eager evaluation, give two possible outputs of the code above in parts (a) and (b) respectively.

Assume that the main thread evaluates the first expression in a list operation.
### a) First Possible Output (fill the first column first and then the next column):

| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |

### b) Second possible output (fill the first column first and then the next column):

| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |
| __________________________ | __________________________ |


EXTRA SPACE but please try and fit your answer in the assigned space
EXTRA SPACE but please try and fit your answer in the assigned space