Koç University
College of Engineering

COMP 301: Programming Language Concepts

Midterm Examination I
Nov 16, 2009, Monday 18:30-21:30
CAS B24 & CAS B07

Instructor: Metin Sezgin
Time Allowed: 120 minutes

Name:_____

Student Number:_____

NOTE: Explain your answers in full. Provide all the work in your exam paper, but make sure the answer boxes have nothing but your final answer to the questions. Include signatures (contracts) for all scheme functions that you implement.

I pledge on my honor that I have neither given nor received unauthorized assistance on this exam.

Signature:_____

Question	Worth	Grade
1	20	
2	20	
3	20	
4	10	
5	10	
6	20	
Total	100	

1. (20 points)

   Consider the set of pairs $S = \{(0,0),(1,1),(2,1),(3,2),(4,3),(5,5),(6,8),(7,13)...\}$ which contains the index of each Fibonacci number along with its value. For example, $(6,8) \in S$ because 8 is the sixth Fibonacci number.[1]

   (a) (6 points) Define the set S in a bottom-up fashion.

   Answer:

   (b) (6 points) Define the set S in a top-down fashion.

   Answer:

   (c) (7 points) Define the set S using rules of inference.

   Answer:

[1] Remember that the i^{th} number in the Fibonacci series $(0,1,1,2,3,5,8,...)$ is defined by the relation $F_i = F_{i-1} + F_{i-2}$ for $i > 1$, and the first two Fibonacci numbers are defined as $S_0 = 0$, $S_1 = 1$.

2. (20 points) Implement a function fun-list that takes an integer n returns a list that has n occurrences of n, n-1 occurrences of n-1 etc. Write your answer in the answer box, and include a one sentence description of what each line in your code does in the back.

   For example:

```
> (fun-list 0)
()
> (fun-list 1)
(1)
> (fun-list 2)
(2 2 1)
> (fun-list 3)
(3 3 3 2 2 1)
> (fun-list 4)
(4 4 4 4 3 3 3 2 2 1)
```

   Hint: Define an auxiliary recursive procedure that keeps track of the computation context (i.e., variables that keep a record of where you are in your computation).

Answer:
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

3. (20 points)

The function extend-env provided with the LET language accepts one variable and one binding only. A variant of this function provided below handles multiple bindings at a time.

```
1  (define extend-env*
2    (lambda (syms vals old-env)
3      (if (null? syms)
4          old-env
5         (extended-env-record (car syms) (car vals) (extend-env*
6                                                     (cdr syms)
7                                                     (cdr vals) old-env)))))
```

(a) (10 points) Write down the signature (contract) for this procedure.

Answer:

(b) (10 points) Explain how this procedure works in plain English in fewer than 50 words. Feel free to refer to line numbers. You have a limit of 50 words, so use your resources intelligently to demonstrate that you truly understand how the function works.

Answer (50 words max):

Word count:

4. (10 points) We have studied two languages so far: LET and PROC. These two languages differ in many aspects, but one has to do with the set of denoted and expressed values supported by each language.

   (a) (5 points) List the set of denoted and expressed values for LET and PROC.

   Answer:

   (b) (5 points) What do denoted and expressed values specify in a language? In particular, imagine a language where procedures are expressible, but not denotable. What does this mean in practice for the language user user?

   Answer (50 words max):

   Word count:

5. (10 points) Draw the abstract syntax tree for the following programs written in the PROC language. The PROC grammar is provided in Fig. 1 for your convenience.

   (a) (5 points)

```
1 (proc (f) (f (f 77))
2  proc (x) -(x,11))
```

Answer:

(b) (5 points) HINT: You can label an reuse parts of your answer from the previous part to save time here.

```
1 let f = proc (x) -(x,11)
2 in (f (f 77))
```

Answer:

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(` *Expression* `,` *Expression* `)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (` *Expression* `)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Expression ::= `proc (` *Identifier* `)` *Expression*
`proc-exp (var body)`

Expression ::= `(` *Expression* *Expression* `)`
`call-exp (rator rand)`

Figure 1: The PROC grammar.

6. (20 points) Moon Microsystems is adding features to LET to build a new language D-flat (D♭), a competitor to C#. Back when the project was spawned, the idea of D-flat sounded good but now the project team is stuck and they need your help in adding a few new features to LET.

   More specifically, D♭ extends LET by having rational numbers. Moreover, they want all functions that work on regular numbers to work on rational numbers as well.

   (a) (5 points) Carefully look over the implementation of the LET language and list procedures that should be modified to accommodate the extension.

   Answer:

(b) (5 points) Write down the specification for the new zero? function.

Answer:

(c) (10 points) You are now asked to modify D♭ such that whenever it is possible, the value of numeric expressions are kept as integers (i.e., instead of 8/2, we want 4). Which functions require modification now, and how?

Answer: