Programming Languages
Exam 2

Write code using Scam syntax and semantics unless otherwise directed. Do not use assignment or loop constructs such as \texttt{while} or \texttt{for} unless otherwise directed. Make sure your test has seven questions; you will receive zero credit for missing questions/answers.

1. Consider the \texttt{do-while} statement in the language \texttt{Arggh!}, an example of which follows:

\begin{verbatim}
do {
   display(x);
   x = x + 1;
} while (x < 0);
\end{verbatim}

In an \texttt{Arggh!} \texttt{do-while}, the keyword \texttt{do} is followed by a block. The block is followed by the keyword \texttt{while}, which, in turn, is followed by a parenthesized expression. The \texttt{do-while} statement is terminated by a semicolon.

Assume that the following is true:

(i) a block encloses a sequence of definitions followed by a sequence of statements
(ii) a block may have no definitions and no statements
(iii) a block may have definitions but no statements
(iv) a block may have statements but no definitions
(v) a statement is an expression followed by a semicolon or a statement is a do-while statement.
(vi) the grammar rule for a definition already exists
(vii) the grammar rule for an expression already exists
(viii) the terminals of the language include \texttt{DO}, \texttt{WHILE}, \texttt{OBRACE}, \texttt{CBRACE}, \texttt{OPAREN}, \texttt{CPAREN}, and \texttt{SEMI}

You are to define a series of grammar rules to recognize \texttt{Arggh!} \texttt{do-while} statements. Use the convention that terminals are written in (mostly) upper-case and non-terminals are written in (mostly) lower-case. If you use non-terminals beyond \texttt{expression} and \texttt{definition}, you must provide the associated grammar rules.

2. Draw a parse tree that represents a \texttt{do-while} in the \texttt{Arggh!} language. Then define a parsing function that reflects the grammar rule from the previous question and generates this parse tree. For full credit, all subcomponents of the \texttt{do-while} must be present in the tree. Use pseudocode for the function definition; assignment is allowed. You do not need to implement the parsing functions for the subcomponents. Remember, you need to draw a single tree and define a single function. Render nodes in the parse tree as tagged circles and sub-parse trees as tagged blobs. Points will be deducted for irrelevant nodes in the parse tree.

3. Draw the environment diagram that results from execution of the following expressions, in the order given. Label your environments such that the order of their creation is apparent. Denote with an asterisk which environments, if any, will be garbage collected.

\begin{verbatim}
(define x 1)
(define (f y)
   (define (g z)
      (define (iter result count)
         (cond
            ((= count 0) result)
            (else (iter (+ result 1) (- count 1)))
         )
      )
      (iter 0 (+ (- x y) z))
   )
   g)
(define a (f (+ x x)))
(define b (a 3))
\end{verbatim}
4. In the cons-cell style of the textbook, draw a representation, of a structure, pointed to by \( t \), such that:

```lisp
scam> (car t)
((1 2 3))
scam> (cadadr t)
(3 . 4)
scam> (caadr t)
(3 . 4)
scam> (caddr t)
5
scam> (cdddr t)
(6 7)
scam> (cddadr t)
nil
```

All cells in the structure must be shown.
5. Consider the following versions of accumulate:

\[
\begin{align*}
\text{(define (accumulate1 op base items)} & ) \\
& (\text{cond}) \\
& (\text{((null? items) base)})) \\
& (\text{(else (op (car items) (accumulate1 op base (cdr items))))}) \\
\end{align*}
\]

\[
\begin{align*}
\text{(define (accumulate2 op base items)} & ) \\
& (\text{define (iter store source)})) \\
& (\text{cond}) \\
& (\text{((null? source) store})) \\
& (\text{(else (iter (op store (car source)) (cdr source))}) \\
\end{align*}
\]

\[
\begin{align*}
\text{(define (accumulate3 op base items)} & ) \\
& (\text{define (iter store source)})) \\
& (\text{cond}) \\
& (\text{((null? source) store})) \\
& (\text{(else (iter (op (car source) store) (cdr source))}) \\
\end{align*}
\]

For the following expressions, give the resulting value:

\[
\begin{align*}
& (\text{accumulate1 + 0 (list 1 2 3)}) \\
& (\text{accumulate2 + 0 (list 1 2 3)}) \\
& (\text{accumulate3 + 0 (list 1 2 3)}) \\
& (\text{accumulate1 - 0 (list 1 2 3)}) \\
& (\text{accumulate2 - 0 (list 1 2 3)}) \\
& (\text{accumulate3 - 0 (list 1 2 3)}) \\
& (\text{accumulate1 cons nil (list 1 2 3)}) \\
& (\text{accumulate2 cons nil (list 1 2 3)}) \\
& (\text{accumulate3 cons nil (list 1 2 3)}) \\
\end{align*}
\]

Extra credit (3 points): Under what situations do all three versions of accumulate return the same values? For credit, your answer must be as general as possible.

6. Define the function allPairs, which generates all pairings of two given lists; items in the first given list come first in the resulting pairings. For example, the expression \((\text{allPairs '(a b c d) '(1 2 3)})\) would produce the result:

\[
\begin{align*}
& ((a 1) (a 2) (a 3)) \\
& (b 1) (b 2) (b 3) \\
& (c 1) (c 2) (c 3) \\
& (d 1) (d 2) (d 3)
\end{align*}
\]

You must use this strategy: generate the first row, then generated the rest of the table, then combine the two results with \text{append}. You may not define any helper functions and you may only call \text{map} once.
7. Recall the definition of the Church Numeral \textit{two}:

\begin{verbatim}
(define two (lambda (f) (lambda (x) (f (f x)))))
\end{verbatim}

Consider these definitions concerning Church Numerals:

\begin{verbatim}
(define a (lambda (f) (lambda (x) f)))
(define b (lambda (f) (lambda (x) x)))
(define c (lambda (f) b))
(define (g n) ((((n c) a) 'FLURB) 'GLORP))
\end{verbatim}

What does \((g \text{ zero})\) return?

What does \((g \text{ one})\) return?

What does \((g \text{ two})\) return?

Give more appropriate names for \(g\), \textit{FLURB}, and \textit{GLORP} that reflect the semantics of these symbols.