Question 1.  [14 marks]

Consider the following definitions, and expansion-time transformation(s).

(define-syntax-rule (A v)
  (lambda (u)
    (set! v (u v))
    v))
(define a +)
(define b -)
(define c sqr)

Part (a)  [2 marks]

Write out the expansion of the following:

(define Aa (A a))
(define Ab (A b))
(define Ac (A c))
(Aa (lambda (b) c))
(a 3)

Part (b)  [6 marks]

Draw the FINAL trace diagram from running ALL the above code AND:

(Ab (lambda (a) (lambda (a) (c a))))
(Ac (lambda (c) (c 2)))
Part (c) [6 marks]

For each of the following, state the value or precise error it would produce if now evaluated (independently of each other):

(a 3)

(b 5)

(c 7)
Question 2.  [8 marks]

Consider the following code:

(define Not (lambda (b) (if b #f #t)))
(define Or (lambda (b0 b1) (if b0 b0 b1)))
(define-syntax-rule
  (Or1 <boolean-expression-0> <boolean-expression-1>)
  (if <boolean-expression-0>
    <boolean-expression-0>
    <boolean-expression-1>))
(define T (lambda ()
  (displayln "T")
  #t))

Part (a)  [4 marks]

For each of the following expressions, state the value produced when the expression is run/evaluated, OR the precise error that occurs.

Not
Or1
(procedure? Not)
(procedure? Or1)
(map Not (list (< 1 2) #t))
(map Or1 (list (< 1 2) #t))

Part (b)  [4 marks]

For each of the following expressions, do the following.
1. Write down the final expression after it is transformed/expanded (according to the above code) at compile/expansion time. Don’t do the standard transformations that Racket does, e.g. don’t tag function calls with #\%app.
2. State the value produced when the expression is run/evaluated, AND any displayed output.

(Or (T) (T))

(Or1 #t (T))
Question 3. [10 marks]
For each of the following, give an equivalent expression (assume all variables accessed are defined).
The only functions you may use are \textit{f} and \textit{list}.
The only other operation you may use is \textit{quote}.

\textbf{Part (a)} [3 marks]
\begin{verbatim}
(apply f (list a b c))
\end{verbatim}

\textbf{Part (b)} [3 marks]
\begin{verbatim}
(map f '(list a b c))
\end{verbatim}

\textbf{Part (c)} [2 marks]
\begin{verbatim}
(apply f (list (list a b) (list c d)))
\end{verbatim}

\textbf{Part (d)} [2 marks]
\begin{verbatim}
(apply map (list list '(1 2 3) '(4 5 6) '(7 8 9)))
\end{verbatim}
**Question 4. [12 marks]**

Consider a function `fns-called` that takes a list representing code for a nested function call, where all the function expressions are represented by symbols (to represent variable access).

It returns a list of the functions (as symbols for their variable names) in the order they would be called.

```
(fns-called '(a (b c))) ; produces: 
'(b a)
```

**Part (a) [4 marks]**

Show the expected results of each of the following:

```
(fns-called '(x y z))

(fns-called '(u (w)))

(fns-called '(t (u (w)) (x y z)))
```

**Part (b) [4 marks]**

Write a Racket expression that produces the result of `(fns-called '(t (u (w)) (x y z)))`, by combining the Racket expressions `(fns-called '(x y z))` and `(fns-called '(u (w)))`.

You may not use the symbols `u w x y z` in any other way, nor rely on knowing anything special about the arguments in the Racket expressions `(fns-called '(x y z))` and `(fns-called '(u (w)))`, nor call `fns-called` on anything else.

But you do know that `fns-called` produces a list of the names of the functions in the order they would be called.

**Part (c) [4 marks]**

Write `fns-called`. Use `map` and/or `apply` as appropriate.

---

Total Marks = 44