Question 1 - General Terminology (20 points)

1.1 - T (1 point) What is a special form?

A special form is an expression that requires special evaluation rules.

1.2 - T (1 point) What is an atomic expression?

Atomic expression is an expression that does not contain other sub expression

1.3 - T (1 point) What is a compound expression?

Compound expression is an expression that contains other sub expressions

1.4 - T (1 point) What is a primitive expression?

Primitive expression is an expression whose evaluation is built-in in the interpreter of the language.

1.5 - T (4 points) For each one of the following L3 expressions, decide whether it is atomic and/or primitive and/or compound or none

- 1.5.1 (1 point) +: atomic and primitive
- 1.5.2 (1 point) 5: atomic and primitive
- 1.5.3 (1 point) x: x is a variable expression therefore atomic
- 1.5.4 (1 point) ((lambda (x) x) 5): compound

1.6 - T (1 point) Complete the sentence: multiple expressions in the body of a procedure expression (lambda form) is useful mainly when those expressions have a **side effects**

1.7 - T (1 point) Complete the sentence: we call an expression a ”syntactic abbreviation” of another expression when we define the operational semantic of the language, we do not need to define a new computation rule for this expression type, instead we indicate that this expression is equivalent to a combination of other constructs that mean the same thing.
for example – the let-expression is syntactic abbreviation .

1.8 - T (3 points) What is the expression that the following syntactic abbreviation translates to? (you do not have to evaluate the expression - simply write the form it translates to)

(let ((x (lambda (x) (+ x 1)))

(y ((lambda (y) (- y 22)) 23))

(z 6))

(* ((lambda (x) (+ x 1)) 6) ((lambda (y) (- y 22)) 23)))

1.9 - T (3 points) Recall our definition of shortcut semantics (Practical Session 1 reading material). Do and expressions in Racket support shortcut semantics? Prove your answer (answers without proof will not be accepted).

From practical session 1:
The native some and every methods employ a concept known as 'shortcut semantics'. What this means, is that some stops and immediately returns true at the moment it finds an element that satisfies the predicate. every stops and immediately returns false at the moment it finds an element that does not satisfy the predicate.

Example:

```
#lang racket

(define foo
  (lambda (x)
    (display "side effect occured" )(newline)
    x))


(and #f (foo #f)); does not evaluate foo and does not print

(and (foo #f) #f); does evaluate foo and therefore prints
```

Conclusion: Dr.racket supports shortcut semantics.

(4 points) Recall our definition of functional equivalence (Practical

Session 1 reading material). Given the following two procedures foo and goo,

and assuming that the procedure display never fails and works as defined

Two procedures are functionally equivalent if and only if they either: enter into an infinite loop on the same inputs, or throw exceptions on the same inputs, or halt on the same inputs and return the same value on these parameters

```
(define foo (lambda (x) (+ x 1))(
        (define goo (lambda (x)
                (display 'hi-there)
                (+ x 1))))
```

1.10.1 •(2 points) Are foo and goo functionally equivalent according to

our definition in class? Explain (answers without an explanation will not

be accepted

foo and goo are functionally equivalent.
infinite loop: in goo, according to the assumption the display never fails. Therefore goo will enter an infinite loop iff the exp(+ x 1) will enter an infinite loop. If that will happen foo will enter an infinite loop as well.

or throw exceptions same as above

or halt & return the same value: compound procedure application will return the value of the last expression. In our case the last expression in goo and in foo is (+ x 1).

Therefore, functionally equivalent.

1.10.2 •(2 points) Are foo and goo functionally equivalent when considering

side-effects as well as an addition to the definition we saw in class?

Explain (answers without an explanation will not be accepted).

In goo there is a display that never fails and causes side effect.
In foo on the other hand there's no side effects at all.

Therefore, foo and goo are not functionally equivalent considering side effects.

Question 2 - EVALUATION

2.1

((lambda (x) (+ x (+ (/ x 2) x))) x)
evaluate (define x 12) [compound special from]
    evaluate (12) [atomic]
    return value: 12
    add the binding <<x>, 12>to the GE
return value: void

evaluate ((lambda (x) (+ x (+ (/ x 3) x))) x) [compound non-special form]
    evaluate(lambda (x) (+ x (+ (/ x 3) x))) [compound special form]
        return: <closure (x) (+ x (+ (/ x 3) x))>
    evaluate(x) [atomic]
        return value: 12 (GE)
    apply x to the closure:
        replace all occurrences of x in the body of the closure
        evaluate(+ 12 (+ (/ 12 3) 12)) [compound non-special form]
            evaluate(+) [atomic]
                return value: #<procedure:>>
            evaluate (+ (/ 12 3) 12) [compound non-special form]

```
                                    evaluate(+) [atomic]
                                        return value: #<procedure:>>
                                    evaluate (/ 12 3) [compound non-special form]
                                        evaluate(/) [atomic]
                                            return value: #<procedure:>>
                                        evaluate(3) [atomic]
                                            return value: 3
                                        return value: 4
                                    return value: 16
        return value: 28
```

2.2

```
(define last
(lambda (l)
(if (empty? (cdr l))
(car l)
(last (cdr l)))))

evaluate((define last
(lambda (l)
(if (empty? (cdr l))
(car l)
(last (cdr l))))) [compound special form]
        evaluate(lambda (l)
                (if (empty? (cdr l))
                (car l)
                (last (cdr l)))) [compound special form]
                return value: <closure (l) ((if (empty? (cdr l))
                                (car l)
                                (last (cdr l)))
        add the binding <<last> <closure>> to the GE
        return value: void
```

2.3

```
(define last
(lambda (l)
(if (empty? (cdr l))
(car l)
(last (cdr l)))))

(last '(1 2))

evaluate (define last ...) - same as before

evaluate (last '(1 2)) [non-special compound form]
        evaluate(last) [atomic]
                return value: <closure> from GE
        evaluate('(1 2)) [compound data type literal]
                return value: '(1 2)
        apply '(1 2) to the closure:
                replace all occurrences of l in the body of the closure
                evaluate (if (empty? (cdr '(1 2)))(car '(1 2))(last (cdr '(1 2)))) [compound special form]
                        evaluate (empty? (cdr '(1 2)) [compound non-special form]
                                evaluate(empty?) [atomic]
                                        return value: #<procedure:>>
                                evaluate(cdr '(1 2)) [non-special compound form]
                                        evaluate(cdr) [atomic]
                                                return value: #<procedure:>>
                                        return value: '(2)
                                return value: #f
                        evaluate (last (cdr '(1 2)) [compound non-special form]
                                evaluate(last) [atomic]
                                        return value: <closure> from GE
                                evaluate(cdr '(1 2)) [compound non-special form]
                                        evaluate (cdr) [atomic]
                                                return value: #<procedure:>>
                                        return value: '(2)
                                apply '(2) to the closure:
                                        replace all occurrences of l in the body of the closure
                                        evaluate (if (empty? (cdr '(2)))(car '(2))(last (cdr '(2)))) [compound
special form]
                                                evaluate (empty? (cdr '(2)) [compound non-special form]
                                                        evaluate(empty?) [atomic]
                                                                return value: #<procedure:>>
                                                        evaluate(cdr '(2)) [non-special compound form]
                                                                evaluate(cdr) [atomic]
                                                                        return value:
#<procedure:>>
                                                                return value: '()
                                                        return value: #t
                                                evaluate (car '(2)) [compound non-special form]
                                                        evaluate (car) [atomic]
                                                                return value : #<procedure:>>
                                                        return value: 2
                                                ...
                        return value: 2
```

Question 3 - Scopes (10 points)

```
(define fib (lambda (n) ;1
          (cond ((= n 0) 0) ;2
           ((= n 1) 1) ;3
           (else (+ (fib (- n 1)) (fib (- n 2))))))) ;4

(define y 5) ;5
(fib (+ y y));6
```

Binding instance	Appears fist at line	scope	# line of Bound occurrences
fib	1	Universal	4(twice),6
n	1	Lambda body (1)	2,3,4
y	6	universal	6

```
(define triple (lambda (x) ;1
             (lambda (y) ;2
              (lambda (z) (+ x y z))))) ;3
(((triple 5) 6) 7) ;4
```

Binding instance	Appears fist at line	scope	# line of Bound occurrences
triple	1	Universal	4
x	1	Lambda body (1)	3
y	2	Lambda body (2)	3
z	3	Lambda body 3)	3

Question 5 - Syntactic Abbreviation (20 points)

<scheme-exp> -> <exp> | ’(’ <define> ’)’
<exp> -> <atomic> | ’(’ <composite> ’)’
<atomic> -> <number> | <boolean> | <variable> | <quoted-variable>
<composite> -> <special> | <macro> | <form>
<number> -> Numbers
<boolean> -> ’#t’ | ’#f’
<variable> -> Restricted sequences of letters, digits,punctuation marks
<quoted-variable> -> A variable starting with ’
<special> -> <lambda> | <quote> | <cond> | <if>
<macro> -> <let>
<form> -> <exp>+
<define> -> ’define’ <variable> <exp>
<lambda> -> ’lambda’ ’(’ <variable>* ’)’ <exp>+
<quote> -> ’quote’ <variable>
<cond> -> ’cond’ <condition-clause>* <else-clause>
<condition-clause> -> ’(’ <exp> <exp>+ ’)’
<else-clause> -> ’(’ ’else’ <exp>+ ’)’
<if> -> ’if’ <exp> <exp> <exp>
<let> -> ’let’ ’(’ <var-definition>* ’)’ <exp>+
<let*> -> 'let*' '(' <var-definition>* ')' <exp>+
<var-definition> -> ’(’ <variable> <exp> ’)