**Part 1 – Solution:**

1. Since in L3 language "let" expression can be seen as a "syntactic sugar" of anonymous lambda , we can exchange each let expression with its corresponding lambda expression, evaluate the lambda expression and get a valid value.
   So in this case "let" is not considered "special form" , it's evaluation processes is exactly as lambda evaluation processes.

2. Types of semantic errors:
   - Invalid variable type pass to procedure:
     **Example:**
     (define square (lambda (x) (* x x)))
     (define true #t)
     (square true) => all symbols are valid (parsing pass) but semantic error occurs, square expects number but gets boolean instead.

   - Incorrect calculations:
     **Example:**
     (/ 1 0) = > all symbols are valid (parsing pass) but semantic error occurs, dividing by zero is not legal.

   - None exist variables:
     **Example:**
     (define ID (lambda (x) x))
     (ID var) => all symbols are valid (parsing pass) but semantic error occurs, there is no reference to variable "var" unbound identifier).

   - Apply procedure in wrong way:
     **Example:**
     (1 + 2) => all symbols are valid (parsing pass) but semantic error occurs, the interpreter expects prefix notation so it tries to interpret "1" as applying procedure which is not correct.

**3.**
**1.**

**The Changes are marked:**

**<program>**
**<exp>** ::= <define-exp> | <cexp>
<define-exp>
**<cexp>** ::= <num-exp> | <bool-exp> | <prim-op> | <var-ref>|
(if <exp> <exp> <exp>)
| (lambda (<var-decl>*) <cexp>+)|
(quote <sexp>)| | (<cexp> <cexp | value >*)

**<prim-op>** ::= + | - | * | / | < | > | = | not | and |
or | eq? |cons | car | cdr | pair? | list? |
number? | boolean? | symbol? | display | newline
**<num-exp>** ::= a number token
**<bool-exp>** ::= #t | #f
**<var-ref>** ::= an identifier token
**<var-decl>** ::= an identifier token
**<sexp>** ::= a symbol token | ( <sexp>* ) |
( <sexp> . <sexp> )
**Value** = Number | Boolean | Closure | Void | Sexp
**SExp** = Symbol | Number | Boolean | EmptySExp | Pair(SExp,
SExp)

**Zoom-in to the changes:**

**AppExp:**(<cexp> <cexp | value>*).
**Value** = Number | Boolean | Closure | Void | Sexp
**SExp** = Symbol | Number | Boolean | EmptySExp | Pair(SExp,
SExp)

**Explanation:** we let the AppExp hold both values and CExps , so there is no need anymore to use the procedure "valueToLitExp" to change the value into literal expression. We still want to keep the literal expression because it's value to the language is beyond substituting vriable's references.

2. In order to support these changes we need to change the interpreter accordingly in the following way:
* We don't need anymore to substitute values into literal expression before evaluating the application of an proc-expression (primitive proc and compound pr oc).
* We need to support application of an expression on both values and expression all together and not only on values.

**Note:** The only case that an app expression would hold both values and expressions , or only values is in case of the substituting the interpreter does on var references with their equivalent values from arguments.

3. In my opinion it is better to use the original way we evaluate the parse tree and not support to hold values in the AST , because it allows us to detect different kinds of errors in separate stages and take care on each of them separately.

   Furthermore the new implementation does not make life easier since we still need to let the interpreter the ability to evaluate values and expressions all tog ether

4. The valueToLitExp is a function that used in the substitution model in order to substitute the value of a variable with equivalent literal expression that AST is expect to get(AST holds expressions). In normal evaluation such function is not needed since we exchange variables with expressions (In the normal model the evaluation is calculated after the phase of substitution) , so the AST holds the type it expects.

5. In the following program Applicative evaluation execute faster than normal evaluation:

   (define add3 (lambda (x) (+ x x x)))
   (add3 (+ 3 3))

   Lets analyze both evaluation:
   **Applicative:**
   applicativeEval((add3 (+ 3 3))) =>
   ApplicativeEval(add3) => closure<(x) , (+ x x x)>

   ApplicativeEval((+ 3 3)) =>
   ApplicativeEval(+) => <primOp +>
   ApplicativeEval(3) => 3
   ApplicativeEval(3) => 3
   ⇨ 6

   ApplicativeEval(6) (+ 6 6 6)) =>
   ApplicativeEval(+) => <primOp +>
   ApplicativeEval(3) => 6
   ApplicativeEval(3) => 6
   ApplicativeEval(3) => 6

   ⇨ 18.

**Normal:**
NormalEval ((add3 (+ 3 3))) =>
ApplicativeEval(add3) => closure<(x) , (+ x x x)>

NormalEval ((+ (+ 3 3) (+ 3 3) (+ 3 3))) =>

NormalEval (+) => <primOp +>

NormalEval ((+ 3 3)) =>
NormalEval (+) => <primOp +>
NormalEval (3) => 3
NormalEval (3) => 3
⇨ 6

NormalEval (+) => <primOp +>
NormalEval (3) => 3
NormalEval (3) => 3

⇨ 6

NormalEval (+) => <primOp +>
NormalEval (3) => 3
NormalEval (3) => 3
⇨ 6

⇨ 18

**It's easy to observe that the Applicative evaluation took less stages than Normal evaluation.**

In the following program Normal evaluation is faster than applicative evaluation:

**(define x (+ 3 3)) 1**

**Applicative Evaluation:**

ApplicativeEval((define x (+ 3 3)) 1) = >
ApplicativeEval((define x (+ 3 3))
ApplicativeEval((+ 3 3))
ApplicativeEval(+) => <primOp +>
ApplicativeEval(3) => 3
ApplicativeEval(3) => 3
⇨ 6

ApplicativeEval => make bound <x , 6>

ApplicativeEval(1) => 1
⇨ 1

**Normal Evaluation:**
NormalEval ((define x  (+ 3 3)) 1) = >
NormalEval ((define x  (+ 3 3))
NormalEval => make bound <x , (+3 3)>

NormalEval (1) = >1
⇨ 1.

**It's easy to observe that the Normal evaluation took less stages than Applicative evaluation.**

**Part 3.1 – Solution:**

**Explanation of behavior:**

```
#lang lazy
(define x (-))
x
```

The interpreter handles the define expression adding the bound <x , (-)> as is without evaluating the right side of the define. The value of "x" then considered as "promise" means that in some point in the future it's value would be required.

```
#lang lazy
(define x (-))
1
```

The interpreter handles the define expression adding the bound <x , (-)> as is without evaluating the right side of the define. Then the value of "1" is returned.

**The problem and solution of define implementation in L3:**

The problem is that we evaluate the right side of the define expression leading to that when we want to use the defined variable we get the full evaluated expression. The solution is not to evaluate the right side at all , and hold the bound <var , cexp>. That would do the affect of the lazy language.