## Question 1

1. Let is a special form in L3 because it has different calculation rules than a regular expression. “let a = 5” for example tells the interpreter to bind the variable ‘a’ with num value 5.
2. L3 Semantic errors:
3. Call a function with wrong number of arguments:

define a; \\ Define should get 2 arguments

1. Using primOp on illegal types:

(+ 5 “bestStringEva”)

1. Trying to calculate the value of a variable which doesn’t have VarRef:

(+ a 3) when a wasn’t previously defined.

1. Division in 0:

((lambda (x y) (x \ y)) 2 0))

1. **3.1.**

Define value and sex-value:

<value> ::= <sexp-value>

<sexp-value> ::= number | Boolean | string | primOp | closure

add then we’ll add <value> as a <cexp> sub-type

**3.2**

We will need to update applyClosure and remove the line:

const litArgs = map(valueToLitExp, args);

and supply the arguments directly to evalSequence:

return evalSequence(substitute(body, vars, args), env);

**3.3**

We will choose valueToLitExp for two reasons:

* The code will be more readable because there will be a clear separation between values and expressions
* The code will be consisted of less types and therefore less functions that deal with those types.

1. In applicative strategy, the arguments are calculated and turned into values first, and then the closure body expressions gets substituted by the values - therefore valueToLitExp is needed to cast the values back to exp to match the closure body.

On the other hand, in normal strategy the evaluation of the expression only happens last – so that the casting is not needed and valueToLitExp is redundant.

1. Normal is faster than applicative:

((lambda (x) 1 ) arg )

The calculation of arg won’t be evaluated in the normal strategy because the closure doesn’t have any use in this evaluation, whilst it will be computed first in the applicative approach.

Applicative is faster than Normal:

((lambda (x) (+ x x) ) arg )

In applicative, arg will evaluated once at first and applied to x, while in normal strategy arg will be evaluated twice, because it appears twice in the body of the closure.

## Question 3.1

(define x (-))

x

Returns #<promise:x>

X holds a primOp which can’t be evaluated right away (it needs arguments), therefore the lazy language saves it as a promised object – an object which its evaluation not done immediately and gets delayed.

(define x (-))

1

Returns 1.

1 is already an evaluated value, therefore it not gets delayed by the lazy language.

The problem with the implementation of define in L3-Normal, is that the expression assigned in define is evaluated immediately, same as conducted in applicative manner.

The solution is to keep the define value as a compound expression, which will be evaluated later on.