Q1. Yes, because if we don’t use expressions, then the program will not return any value, and nothing will happen in our program. A good example for a language like so is scheme, where we don’t change the given object, we just change values instead.

Q1.2

A .Special forms are required in programing languages because sometimes we need to define special semantics that are not like the semantic in the default procedure application. For example, if we observe if special form, it contains sub expression, such as then and else and if it were primitive operator than the default procedure application would not evaluate the sub expression correctly, which would lead to an error.

b. it could be defined as a primitive operator. However, if it were a primitive operator, it could not achieve its special form – in case there are multiple operands at the same expression, instead of stopping once it found one false expression, it would continue to check the rest of the operands.

Q1.3

Syntactic sugar is a term for a more concise syntax that provides the same functionality for something that already exists. It aims to help make code shorter, therefore, easier to write. No new functionality is introduced.

For example, the COND structure is a syntactic abbreviation, be because it allows abbreviating multiple conditions in a way that is much more understandable.

Another good example would be the LET structure. it is a syntactic abbreviation, because we could have created the same purpose using the primitives, however the code would be a lot longer and much more difficult to understand.

Q1.4
a.

(define x 1) (let ((x 5) (y (* x 3))) y)

The value of y is 3.

(let ((x 5) (y (* x 3))) is the val expressions and after that we have the body of the let exp, hence we first define and then calculate the body exp.

(define x 1) (let* ((x 5) (y (* x 3))) y)

b.

The value of y is 15.

Because the val exp is calculated sequentially in let* we take the value of x=5.

c.

(define x 2)

(define y 5)

(let ((x 1) (f (lambda (z) (+[x:2 0][y:1 0] [z: 0 0])))) ([f:0 1][x: 0 0]))

(let* ((x 1) (f (lambda (z) (+ [x: 1 0][y:2 0][z: 0 0])))) ([f:0 1][x: 0 0]))

d.

```
(define x 2)
(define y 5)
(let
((x 1)
(f (lambda (z) (+ x y z))))
(f x))
(let
((x 1)
(f (lambda (z) (+ (- x 1) y z))))
(f x))
```

d.

(let x 1 (let ((f (lambda (z) (+ x y z)))) (f x)))

e.

(define x 2)
(define y 5)

```
(let
((x 1)
(f (lambda (z) (+ x y z))))
(f x))
(define f (lambda (z) (+ x y)))
(f x)
```

Question 2 contracts :

1. **Make-ok :**
   Signature : (val<T>)
   Type : OkResult<T>
   Purpose : creates an "ok" type result with the value it received.
   Pre-conditions : the functions ok? , error? and result? need to work correctly.

   Tests :
   (define check (make-ok(5)))
   (error? check) ----> false
   (ok? check) ----> true
   (result? check) ----> true

2. **Make-error :**
   Signature : (msg : string)
   Type : FailureResult<T>
   Purpose : creates a "failure" type result with the string it received.
   Pre-conditions : the functions ok? , error? and result? need to work correctly.

   Tests :
   (define z (make-error("Error")))
   (error? check) ----> true
   (ok? check) ----> false
   (result? check) ----> true

3. **Ok? :**
   Signature : (res : Result<T>)
   Type : boolean
   Purpose : checks if the type is of "ok" or not, and responds accordingly.
   Pre-conditions : the functions make-ok and make-error need to work correctly.

   Tests :
   (ok? (make-ok(5))) ----> true
   (ok? (make-error("Error"))) ----> false

4. **error? :**
   Signature : (res : Result<T>)
   Type : boolean
   Purpose : checks if the type is of "failure" or not, and responds accordingly.
   Pre-conditions : the functions make-ok and make-error need to work correctly.

   Tests :
   (error? (make-ok(5))) ----> false
   (error? (make-error("Error"))) ----> true

5. **result? :**
   Signature : (res : Result<T>)
   Type : boolean
   Purpose : checks if the type is of "Result" or not, and responds accordingly.
   Pre-conditions : the functions make-ok and make-error need to work correctly.

   Tests :
   (result? (make-ok(5))) ----> true
   (result? (make-error("Error"))) ----> true
   (result? 5) ----> false

6. **Result->val :**
   Signature : (res : Result<T>)
   Type : string
   Purpose : returns the value if the result is "ok", or the message if the result is "failure".
   Pre-conditions : the functions make-ok and make-error need to work correctly.

   Tests :
   (result->val (make-ok(5))) ----> 5
   (result->val (make-error("Error"))) ----> "Error"

7. **get :**
   Signature : (dict : <T1,T2> ,k : <T1>)
   Type : Result<T2>
   Purpose : gets a dictionary and a key, and returns the value in dict assigned to the given key as an ok result. In case the given key is not defined in dict, an error result should be returned.

   Tests :
   (define d (make-dict))
   (put d 1 2)
   (get d 1) ----> 2

8. **make-dict :**
   Signature : non
   Type : dict<K,V>
   Purpose : creates a new dictionary with a "dict" tag and an empty list.
   Pre-conditions : non.

   Tests :
   (define d (make-dict))
   (car d) ----> "dict"

9. **dict? :**
   Signature : non
   Type : (e : T)
   Purpose : type predicate for dictionaries.
   Pre-conditions : the function make-dict needs to work correctly.

   Tests :
   (define d (make-dict))
   (dict? d) ----> true

10. **put :**
    Signature : (dict : <T1,T2> ,k : <T1> , v : <T2>)
    Type : Result<dict<T1,T2>>
    Purpose : gets a dictionary, a key and a value, and returns a result of a dictionary with the addition of the given key-value. In case the given key already exists in the given dict, the returned dict should contain the new value for this key
    Pre-conditions : the function get needs to work correctly.

    Tests :
    (define d (make-dict))
    (put d 1 2)
    (get d 1) ----> 2

11. **map-dict :**
    Signature : (dict : <T1,T2> , f(x: T2) => <T2>)
    Type : Result<dict<T1,T2>>
    Purpose : gets a dictionary and an unary function, applies the function of the values in the dictionary, and returns a result of a new dictionary with the resulting values.
    Pre-conditions : f needs to be an unary function of the same type dict is.

    Tests :
    (define d (make-dict))
    (put d 1 2)
    (define d2 (map-dict(d f)))
    (get d2 1) ----> f(2)

12. **filter-dict :**
    Signature : (dict : <T1,T2> , pred(k : <T1> , v : <T2>) => boolean)
    Type : Result<dict<T1,T2>>
    Purpose : gets a dictionary and a predicate that takes (key value) as arguments, and returns a result of a new dictionary that contains only the key-values that satisfy the predicate.
    Pre-conditions : the types of pred's arguments are of the same types as the types of the key and value in the dictionary.

    Tests :
    (define d (make-dict))
    (put d 1 2)
    (define d2 (filter-dict(d pred)))
    (get d2 1) ----> pred(2)