300014529 אלדר דמארי

Question 1: Concepts and properties

1.

a. The Value data structure is the adt for creating the Evaluator's Type "Value", in order to distinguish between computed values and preventing them to be affected by repeated evaluation.
it contains 3 procedures

```
Constructor: make-value(Scheme-type-value)
             Type: [Scheme-type-value -> LIST]
Predicate: value?(exp)
           Type: [T->Boolean]
Selector: value-content(value-exp)
          Type: [LIST->Scheme-type-value]

Example of use:
>(make-value 5)                 ; >('value (5))
>(value-content(x))             ; >5
```

b. **Renaming:**Performs consistent renaming of bound variables, in order to let the substitution procedure take place and substitute all free variable occurrences, without mixing them up.

```
Example of use:
>(define x 5)
>(lambda (x) (+ x 1) x)
```

<u>Renaming process:</u>

```
a. x in lambda is a bound variable. Therefore x has been
   renamed with a new name (generated by gensym)-(x->x1).
b. For all occurrences of x in lambda,x is being replaced
   with x1. ⇒ (lambda (x1) (+ x1 1) x)
c. Now free variable occurrences of x can be substituted ⇒
   (lambda (x) (+ x 1) 5)
```

c. The advantage of keeping a small language kernel is that in this way we can provide further abstraction and flexibility to the language, the evaluator provides semantics and implementation only to its kernel expressions and all other (libraries of derived expressions) are defined in terms of the core expressions, and therefore, are independent of the semantics and the implementation.
in other words it makes it really easy to define other expressions in terms of existing ones, for example: if can be a derived expression in term of cond.

d. In lexical scoping the declaration that binds a variable occurrence can be determined based on the program text (scoping structure), without any need for running the program. This property enables better compilation and management tools. It is done by determining the correlation between a variable declaration (binding instance) and its occurrences, based on the static code and not based on the computation order.

e. The two evaluation algorithms differ in the application step:
  - applicative-eval:

  Rename the procedure body and the evaluated arguments, substitute the parameters by the evaluated arguments, and evaluate the substituted body
  - env-eval:

  Bind the parameters to the evaluated arguments, extend the environment of the procedure, and evaluate the procedure body with respect to this new environment.

  That applicative-eval is replaced by variable bindings in an environment structure that tracks the scope structure in the code.
  The environment structure in a computation of env-eval is always isomorphic to the scope structure in the code.
  Therefore, the evaluated arguments step, variable binding and the computations of body,are equivalent.

f. There are several expressions that cannot be repeatedly evaluated, they accessories to create a fundamental procedures and therefore the evaluator must manage it own primitive-procedure adt (distinguish between user procedures).
  The constructor make-primitive-procedure is being used during the constructor of the Global Environment,and in the add-binding! procedure once a new expression which contains primitive procedures is inserted to the GE.

g.
  i. The environment evaluation model improves the substitution model by replacing renaming and substitution in procedure application by environment generation (and environment lookup for finding the not substituted value of a variable).
  ii. The environment model does not handle the problem of repeated analyses of procedure bodies. This is the contribution of the analyzer: A single analysis in static time, for every procedure.

  The analyzing env-eval improves env-eval by preparing a procedure that is ready for execution, once an environment is given.

2.

a. **False**, a new environment is created only when the computation reduces to the evaluation of a closure body. Therefore, there is a 1:many correspondence between environments and lexical scopes.
  for example when using recursion such as:
  (define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))).
  when applying (fact 3), we will receive 4 frames for each $0 \le i \le 3$, all of them corresponds to the same lexical scope (the procedure definition body of fact).

b. **False**, a tree structure of environments data structure is maintained by "*pointers*", due to the fact that there are frames that need to be accessible from different places in the tree.Therefore working with stack structure will limit our accessibility from frame to frame.

c. **True**, define allows us to create and use the same expression over and over again but it does not change the fact that the evaluators code is functional, for example we can let,letrec, let* instead.

d. **True**, DrRacket closure will be the output of the evaluator working with Scheme language in DrRacket. Therefore there must be in the interpreter a way to handle DrRacket closures.

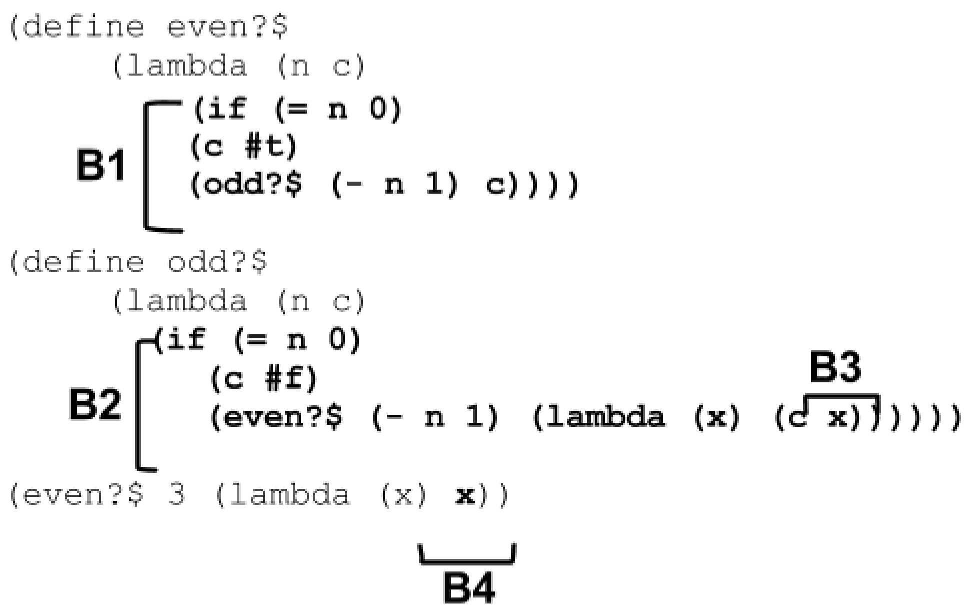
e. **False**, for example let the following code:

```
>(define make-pair (lambda (x y)
                         (lambda (sel)
                           (sel x y))))
>(define p1 (make-pair 5 10)
>(let ((x 1)
       (y 1))
     (p1 (lambda (first second)
             first)))
```

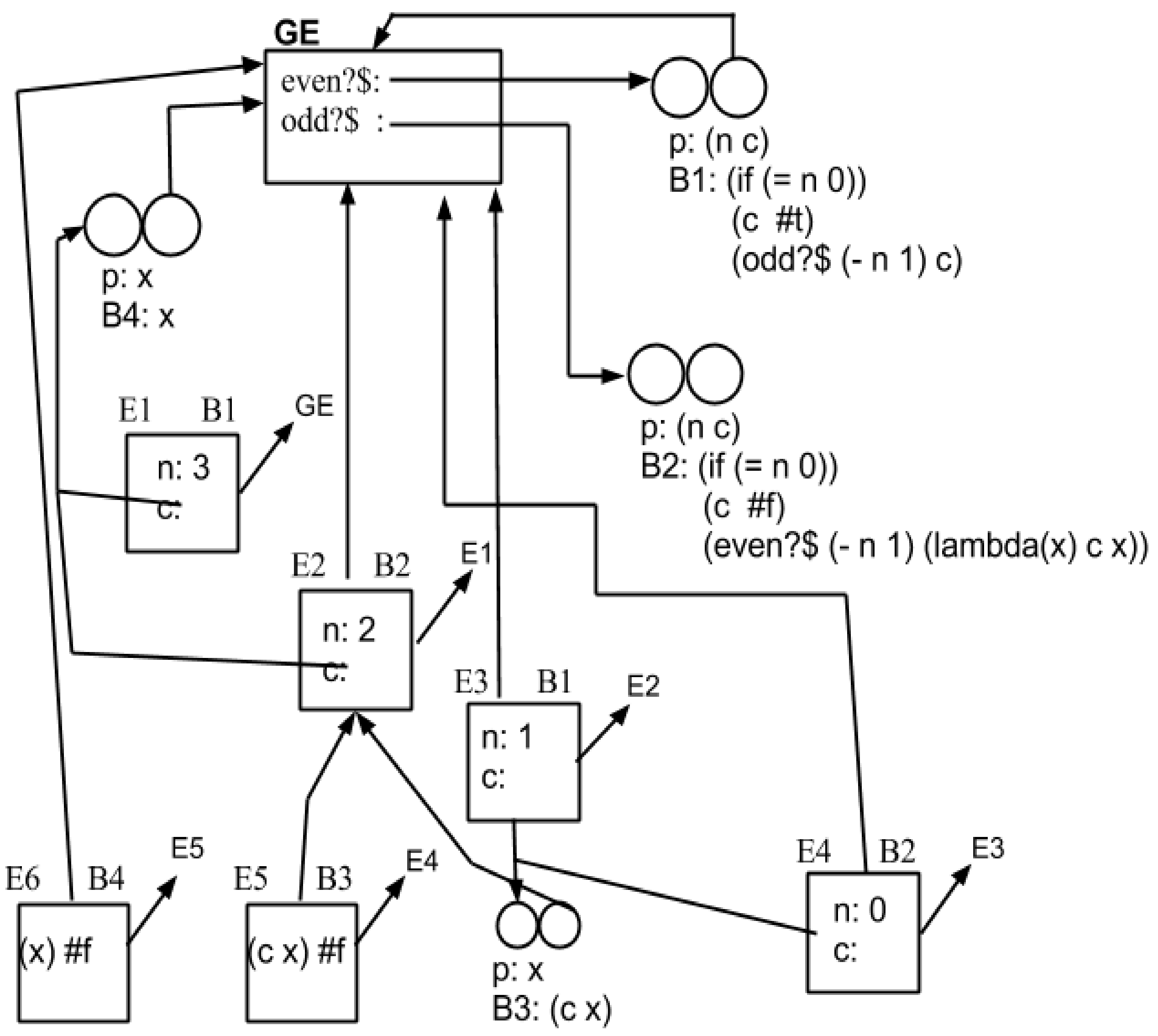
when we evaluate the code using the environment model and lexical scoping the variables first and second receive the values of p1, first=5 second=10
but using the environment model and dynamic scoping the variables first and second receive the refering to the calling frame therefore first=1 second=2.

Question 3: Environment diagram

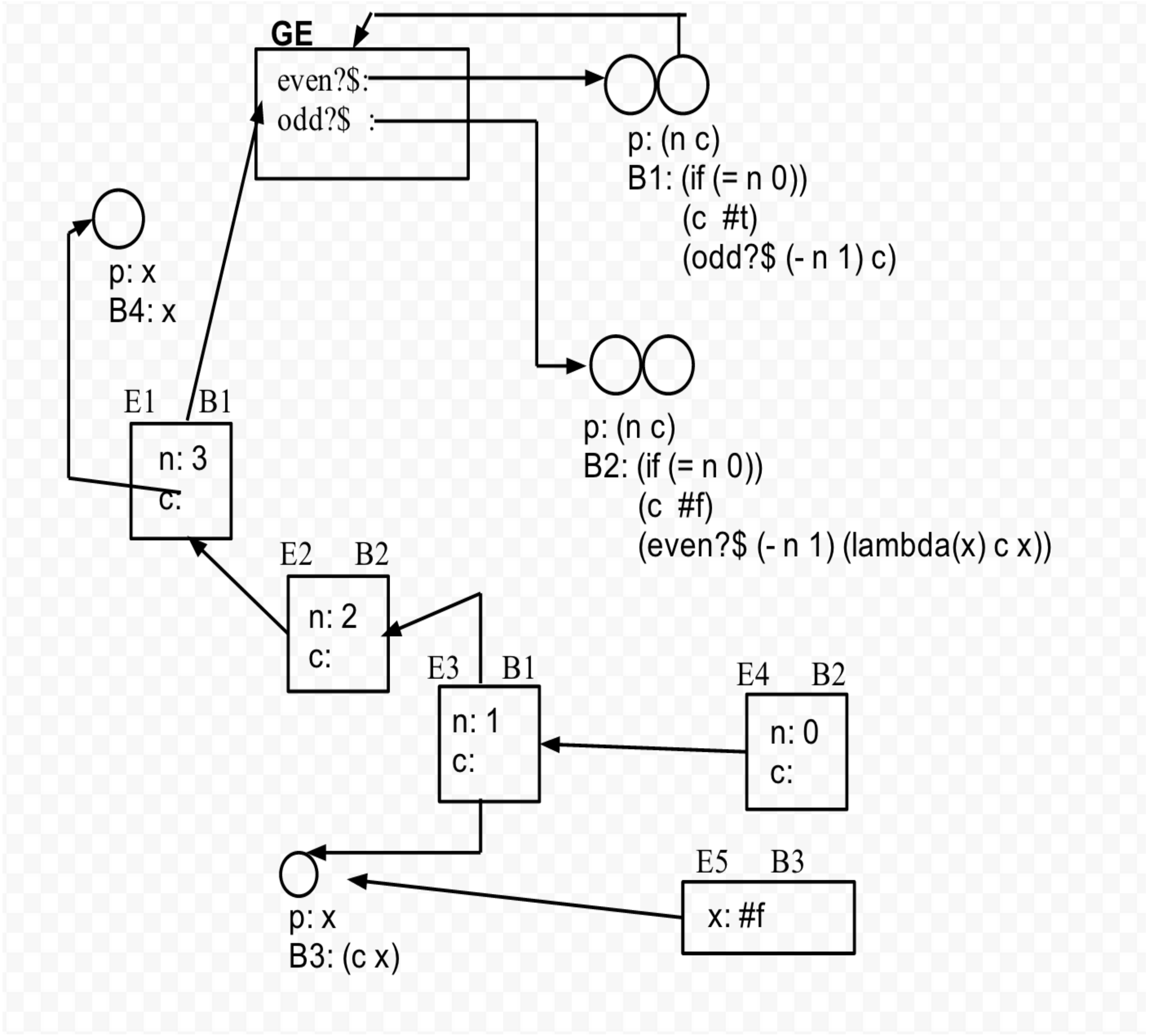
1. Static block marker



```
(define even?$
     (lambda (n c)
       (if (= n 0)
       (c #t)
       (odd?$ (- n 1) c))))
(define odd?$
     (lambda (n c)
      (if (= n 0)
        (c #f)
        (even?$ (- n 1) (lambda (x) (c x))))))
(even?$ 3 (lambda (x) x))
```


GE
even?$:
odd?$:
p: (n c)
B1: (if (= n 0))
(c #t)
(odd?$ (- n 1) c)
p: (n c)
B2: (if (= n 0))
(c #f)
(even?$ (- n 1) (lambda(x) c x))
p: x
B4: x
E1 B1
n: 3
c:
GE
E2 B2
n: 2
c:
E1
E3 B1
n: 1
c:
E2
E4 B2
n: 0
c:
E3
E6 B4
(x) #f
E5
E5 B3
(c x) #f
E4
p: x
B3: (c x)

2.Result of evaluation: infinite loop



3.

(define foo

(lambda goo

(let (a 1)

(b 1)

(lambda (arg) (goo arg)))))

> *(define goo (foo (lambda (y) (+ y* 5))))

Question 4: Data structures of the evaluators

1.

   b. In Lazy implementation it is easier to add new operations.Because once implementing new operation there is no need to change older operation implementations that work with the new operation.

2.

   b. The list-based is preferred. In the List-based method we save memory usage and runtime (less closures are used, while searching for a variable).
   using build-in commands of scheme when working on lists (i.e sequence operations) is an advantage instead of using user commands.
   While using the lookup-based method we look in all the frames above. An “expensive” process.

Question 5: Evaluator errors

1. `(define the-global-environment (make-the-global-environment))`

   The error occurring: Installation error.

   We are trying to work in an environment that was not yet defined.
   the evaluator we are using never initialized the global environment that contains primitive procedures that derive user’s procedure, or defining variables without declaring where they belong.

2. `(derive-eval 'a)`
   The error occurring: programming error.

   The evaluator is trying to evaluate a variable that was not yet defined and it is the user’s responsibility to define variables beforehand.