**Assignment 2**

**Part 1: Concept Questions**

**Submission instructions**: Write all answers in the file hw3\_part1.pdf

1. What is the role of the function value->lit-exp in the substitution-evaluator? Why is it needed?
2. The normal evaluator doesn't need value->lit-exp. Why?
3. The environment-evaluator doesn't need value->lit-exp. Why?
4. In the substitution model procedure application involves renaming. Find an example that requires repeated renaming.
5. List the advantages and disadvantages of keeping a small language core and a large library of derived expressions.
6. What are the reasons for switching from the Substitution Model to the Environment Model?
7. Can the apply-procedure procedure have as a parameter a Racket closure (i.e. not a primitive)? explain and address interpreters.
8. Explain why we can typecheck letrec expressions without specific problems related to recursion and without the need for recursive environment like we had in the interpreter.
9. In the [type-inference.rkt](https://www.cs.bgu.ac.il/~ppl172/wiki.files/code/types/L5-typeinference.rkt) implementation - we represent Type Variables (TVar) with a content field (which is a box which contains a Type Expression value or #f when empty). In this representation, we can have a TVar refer in its content to another TVar - repeatedly, leading to a chain of TVars. Design a program which, when we pass it to the type inference algorithm, creates a chain of length 4 of Tvar1->Tvar2->Tvar3->Tvar4. Write a test to demonstrate this configuration.

**Part 2: Environment Model**

**Submission instructions**

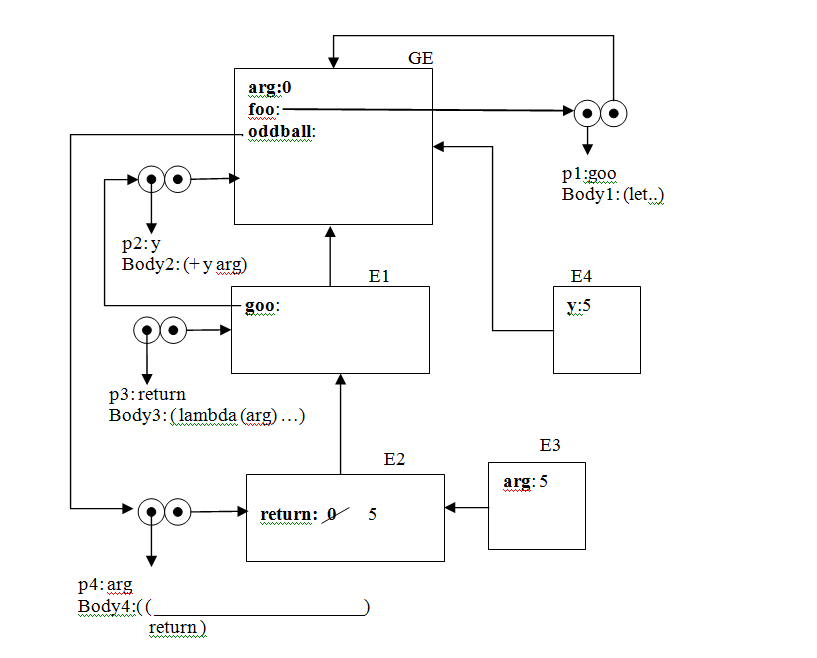
Write all answers in the file hw3\_part2.pdf

**Question 2.1**

Draw an environment diagram for the following computation. Make sure to include the lexical block markers, the control links and the returned values.

1. (define even?$
2. (lambda (n c)
3. (if (= n 0)
4. (c #t)
5. (odd?$ (- n 1) c))))
7. (define odd?$
8. (lambda (n c)
9. (if (= n 0)
10. (c #f)
11. (even?$ (- n 1) (lambda (x) (c x))))))
13. (even?$ 3 (lambda (x) x))

**Question 2.2**

Given the following diagram, complete the code below. 

1. (define arg 0)
3. (define foo
4. \_\_\_\_\_\_\_\_\_\_\_\_\_
5. \_\_\_\_\_\_\_\_\_\_\_\_\_
6. \_\_\_\_\_\_\_\_\_\_\_\_\_
7. \_\_\_\_\_\_\_\_\_\_\_\_\_
8. \_\_\_\_\_\_\_\_\_\_\_\_\_
9. \_\_\_\_\_\_\_\_\_\_\_\_\_
11. (define oddball (foo (lambda (y) (+ y arg))))
12. (oddball 5) ; 5

**Part 3: Support Both Applicative and Lazy Evaluation**

**Introduction**

In this part you are asked to extend the syntax of procedure declarations to let the user control whether or not arguments are to be delayed. For example, the definition

1. (define f
2. (lambda ( a (b lazy) c)
3. ...))

would define f to be a procedure of three arguments, where the first and third arguments are evaluated when the procedure is called (applicative order), and the second argument is delayed (normal order). Thus, ordinary procedure definitions will produce the same behavior as ordinary Scheme, while adding the lazy declaration to each parameter of every compound procedure will produce the behavior of the lazy evaluator.

You are asked to to extend both the substitution interpreter and and environment interpreter. You may use the changes in the AST for both.

**Examples**

The following code should return a value of 1:

1. (define f
2. (lambda (a (b lazy))
3. a))
5. (f 1 (/ 1 0))

The following code raise an exception:

1. (define f
2. (lambda (a (b lazy))
3. a))
5. (f (/ 1 0) 1)

**Substitution Interpreter**

**Submission Instructions**

The directory hw3\_part3\_sub contains all the files needed for the L3 interpreter, and a test file. Modify the files in the directory, and make sure the tests are running.

**What to do**

Design and implement the changes required to produce such an extension to the substitution interpreter. You will have to implement new syntax procedures to handle the new syntax for lambda (the variable declarations). You must also arrange for eval or apply to determine when arguments are to be delayed, and to force or delay arguments accordingly. Hint: you have an implementation of L3-normal. Use it.

**Environment Interpreter**

**Submission Instructions**

The directory hw3\_part3\_env contains all the files needed to run the environment interpreter. Modify the files to answer this question.

**Normal Order in the Environment Model**

Almost all the required changes center around procedure application.

The basic idea is that, when applying a procedure, the interpreter must determine which arguments are to be evaluated and which are to be delayed. The delayed arguments are not evaluated; instead, they are transformed into objects called *thunks*. The thunk must contain the information required to produce the value of the argument when it is needed, as if it had been evaluated at the time of the application. **Thus, the thunk must contain the argument expression and the environment in which the procedure application is being evaluated**.

The process of evaluating the expression in a thunk is called *forcing*. In general, a thunk will be forced only when its value is needed: when it is passed to a primitive procedure that will use the value of the thunk; when it is the value of a predicate of a conditional; when it is the value required; and when it is the value of an operator that is about to be applied as a procedure.

**What to do**

Design and implement the changes required to produce this extension to the environment interpreter. You can reuse the syntax procedures from the previous question. You must also arrange for eval or apply to determine when arguments are to be delayed, and to force or delay arguments accordingly.

**Part 4: Type Checker**

**Submission Instructions**

Put all files in the directory hw3\_part4.

**Support define Expressions**

Add support for define expressions in the Type Checker. For example, the following code should be typed void:

1. (define [foo : number] 5)

and the following code should raise a type error:

1. (define [foo : number] (lambda (x y) (+ x y)))

**Support Pair(T1, T2) Compound Type**

Add support for the Pair type in the type checker. Follow the following steps:

* Add Pair as part of the TExp type language
* Add primitives cons, car, cdr to the type checker
* Add quote special form and its typing rule
* Extend the type language implementation to support comparison of type expressions including Pair.

**Examples**

1. (lambda ([a : number] [b : number]): (Pair number number)
2. (cons a b))