# Assignment 3

## Part 1: Concept Questions

1.What is the role of the function value→lit-exp in the substitution-evaluator? Why is it needed?
Answer: The role is to transform evaluated values to the AST that represented them before evaluation. For example transform the value 3 to (num-exp 3). The applicative eval process evaluates the args, and transforms them from ast to values. But the substitution needs the AST that represents the calculated values, in order to evaluate the substituted args in the body.

2.The normal evaluator doesn't need value->lit-exp. Why?
Answer: The normal evaluator don't need to calculate the args. It just substitutes.
The args are already represented as ASTs (cexp) and aren't transformed to calculated values, therefore no "unparsing" is needed.

3.The environment-evaluator doesn't need value->lit-exp. Why?
Answer: This process does not execute substitution. Instead, when facing a var-ref, the interpreter activates apply-env on the var-ref and looks for it in the environment, which is bounded to a calculated value (If exists).

4.In the substitution model procedure application involves renaming. Find an example that requires repeated renaming.
Answer:
(define z (lambda (val1 val2) (+ val1 val2 )))

(((lambda (x) (lambda (z) (x z)))
(lambda (w) (z w ((lambda (w) w) 5))))
2)
It's app expression. So needs to make renaming in the body. 1st rename.

After the 1st rename, z will transform to z1 in the body and substitution will be performed.
((closure (z1) ((proc-exp (w) (z w ((lambda (w) w) 5)))) z1))

It's an app-exp, so again renaming on the body will be called.
After the 2nd rename, w will transform to w1 in the body, and the inside w (at the inside lambda) will be transformed to w2. Then substitution will be performed. ((closure (w1) (z w1 ((lambda (w2) w2) 5))) 2)

It's an app-exp, so again renaming on the body will be called.
After the 3rd rename, w2 will transform to w21 in the body and substitution will be performed.

eval( (z 2 ((lambda (w21) w21) 5)) ) -> 7
But without second rename: ((closure (w) (z w ((lambda (w) w) 5))) 2)

And without third rename:  eval  (z 2 ((lambda (w) 2) 5))  -> 4

We can see that the result is DIFFERENT when the repeated renaming isn't being executed, therefore this is an example that REQUIRES repeated renaming in order to get a correct result.

5.List the advantages and disadvantages of keeping a small language core and a large library of derived expressions.

Answer:

**Advantages**: Few primitives and data types. Simple syntax. Evaluation rules are simple, not many special cases. Although, it's very expressive language. It can be Turing complete. It can also be functional language (like scheme).

It's comfortable for programmers that work with the language properties.

It's easy to write interpreter for these kind of languages, because of the small core and rules.

**Disadvantages**: Not a multi-paradigm language. Won't support in a comfortable way in lots of paradigms like Java for example: Object Oriented, or support in easy Server-Client communication, etc.

6.What are the reasons for switching from the Substitution Model to the Environment Model?

Answer: The problem with the substitution model was efficiency. A quite-short expression could be renamed and substituted again and again recursively, coping the expression's AST again and again, causing extensive memory allocation / garbage collection, and making the whole process be very slow. On the other hand, on the environment model, there won't be copies of the expression's AST again and again, the data necessary for evaluation will be saved in the environment that is a part of every closure.

7.Can the apply-procedure procedure have as a parameter a Racket closure (i.e. not a primitive)? explain and address interpreters.

Answer: Theoretically yes, but it will require changes in our language.

First of all, another case in the apply-procedure needs to be added: (procedure? Proc),

The args are already calculated, so what is left is to apply the procedure on the args.

Our meta-language (Scheme) works similarly to the environment model, so when applying these kind of procedures on arguments, no substitution will be required.  Currently, we don't have these support, so error will be thrown.

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.

Answer: At evaluation of a letrec expression, we need to evaluate the body of closure x, which might include expression y that is defined in the next binding. But in order to evaluate that expression y, it needs a binding of y to it's value, (which is defined later on), and there is where the problem erupts. On the other hand, at type checking, although it doesn't have y's value yet, it just needs to compare it with the type annotations, that are already given to us.

Therefore no evaluation of the body is needed, and there aren't any recursive environment problems.

9.In the type-inference.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.

Answer:

(typeof-exp (parseL5 '(let ((MY 1)) (let ((TYPE MY)) (let ((ANNOTATION TYPE)) (let ((IS ANNOTATION)) IS))))) (make-empty-tenv))

Will give us this output:

-> '(tvar T11974 #&(tvar T11973 #&(tvar T11972 #&(tvar T11971 #&num-te))))

Which is a chain of Tvar1→Tvar2→Tvar3→Tvar4 that is eventually a num-te.

The typeof-let-exp procedures' return value is:

(typeof-exps body (extend-tenv tenv vars var-tes))

So that's the reason for the chaning creation. (The extending of the env and the recursive call to typeof-exp).