HW #1

- Write and test outermost beta-reduction
  - on your own
    - not using a builtin version (eval, apply)
  - in your favorite language
    - decide on representation
  - email to me end of month (incl. documentation)

Haskell

```
data Term = Var Char | Lambda Char Term | App Term Term

-- subst e x v: replace free x in e with v
subst:: Term -> Char -> Term -> Term
subst (Lambda y e) x v = if x == y then (Lambda y e) else let e' =
(subst e x v) in (Lambda y e')
subst (Var y) x v = if x == y then v else (Var y)
subst (App e1 e2) x v = App (subst e1 x v) (subst e2 x v)

-- one leftmost beta step
eval:: Term -> Term
eval (App (Lambda x e) v) = subst e x v -- beta rule
eval (App (App e1 e2) v) = let e = eval (App e1 e2) in (App e v) -- app1
rule

-- beta until a value is obtained
beta:: Term -> Term
beta (Lambda x e) = Lambda x e
beta (App e1 e2) = let e = eval (App e1 e2) in beta e
```

Scheme

```
(define (subst a x b)
  ;; replace a for all free x in b
  (cond ((equal? b x) a) ; replace
        ((symbol? b) b) ; done
        ((equal? (first b) 'lambda) ; abstraction
         (if (equal? (second b) x) ; bound
             b
             (list 'lambda (subst a x (second b)) (subst a x (third b)))))
        (#T (list (subst a x (first b)) (subst a x (second b))))))

(define (beta1 e)
  ;; single leftmost-outermost step
  (cond
   ((symbol? e) e)
   ((equal? (first e) 'lambda) e)
   ((equal? (first (first e)) 'lambda)
    (subst (second e) (second (first e)) (third (first e))))
   (#T (list (beta1 (first e)) (second e)))))

(define (beta e)
  ;; repeated outermost
  (if (or (symbol? e) (equal? (first e) 'lambda))
      e
      (beta (beta1 e))))
```

Python (Tomer)

```
def createLambda(l1, l2 = None, v = None):
    if l2 == None and v != None:
        return "(lambda "+v+"."+l1+")"
    if l2 != None and v == None:
        return "("+l1+" "+l2+")"
    if l2 == None and v == None:
        return l1

def betaNormalOrder(LAMBDA):
    if len(LAMBDA) == 1:
        return LAMBDA
    if LAMBDA[0:7] == "(lambda": # of the form (lambda v. M)
        v = LAMBDA[8]
        return "(lambda "+v+"."+betaNormalOrder(inner(LAMBDA))+")"
    else: # of the form (M N)
        (M,N) = pair(LAMBDA)
        if LAMBDA[0:8] == "((lambda":
            v = M[8]
            inn = betaNormalOrder(inner(M)) # beta over the inner of the leftmost.
            M = betaNormalOrder(rewrite(inn,N,v))
            return betaNormalOrder(M)
        else:
            M = betaNormalOrder(M)
            if M[0:7] == "(lambda":
                return betaNormalOrder("("+M+" "+N+")")
            else:
                return "("+M+" "+betaNormalOrder(N)+")"

def pair(LAMBDA):
# partitions an expression of the form (M N) into (M,N).
    p = center(LAMBDA)
    M = LAMBDA[1:p]
    N = LAMBDA[p+1:-1]
    return (M,N)

def rewrite(M, N, v):
# rewrites the scope M: v := N.
    if len(M) == 1 and M == v: # if a simple variable and == the re-written.
        return N # re-write.
    elif len(M) == 1 and M != v: # if a simple variable and != the re-written.
        return M # stay the same.
    if len(M) > 7 and M[0:7] == "(lambda":
        u = M[8]
        if v == u:
            return M
        else:
            return "(lambda "+u+"."+rewrite(inner(M),N,v)+")"
    else:
        (A,B) = pair(M)
        return "("+rewrite(A,N,v)+" "+rewrite(B,N,v)+")"

def inner(LAMBDA):
    return LAMBDA[10:-1]

def beta(LAMBDA, f = betaNormalOrder):
# a general beta call function.
    return f(LAMBDA)

def center(s):
    if s[1] != "(": # of the form: (v N)
        return 2
    count = 1
    pos = 1
    while count>0 and pos<len(s):
        pos = pos+1
        if s[pos] == ")": count = count-1
        if s[pos] == "(": count = count+1
    if count == 0: return pos+1
    else: return None
```

Python (Amir)

```python
def normal_form(lam):
    # Converts a lambda expression to its normal form (if such a form exists)
    # using repeated execution of the beta rule on the first possible place,
    # until one can't apply it anywhere.
    new_lam = one_step_towards_normal_form(lam)
    while new_lam != lam:
        lam = new_lam
        new_lam = one_step_towards_normal_form(lam)
    return lam

def one_step_towards_normal_form(lam):
    # Executes the beta rule once, in the earliest place where it's possible to
    # do that, and returns the result. If it can't be applied anywhere, the
    # expression is returned without changes.
    lam = re.sub(' ', '', lam)
    match1 = re.search('\[lambda', lam) # Try and find places to apply the beta rule.
    match2 = re.search('\[([A-Z_]+)', lam) # Even where the lambda is a dictionary item.
    if not match1 and not match2:
        return lam # Nowhere to apply the beta rule.
    if not match1 or (match1 and match2 and match1.start(0) > match2.start(0)):
        return (lam[:match2.start(1)] +
                ready_expressions[lam[match2.start(1):
                                      match2.end(1)]] +
                lam[match2.end(1):]) # Replace the dictionary item with it's definition and return.
                                     # The next time we apply the beta rule it will apply it.
    variable_start = match1.start(0) + 8 # The ( brackets where the variable starts.
    variable_end = find_matching_bracket(lam, variable_start - 1) # The ) where it ends.
    variable = lam[variable_start:variable_end] # Isolate the variable.
    body_start = variable_end + 2
    body_end = find_matching_bracket(lam, body_start - 1)
    body = lam[body_start:body_end] # similarly for the body.
    parameter_start = find_matching_bracket(lam, match1.start(0)) + 2
    parameter_end = find_matching_bracket(lam, parameter_start - 1)
    parameter = lam[parameter_start:parameter_end] # and for the parameter of the lambda expression.
    return lam[:match1.start(0)] + replace_variable(body, variable, parameter) + lam[parameter_end+1:]

def replace_variable(body, variable, parameter):
    # Replaces any free occurance of the variable with the parameter in the body
    # and returns the result.
    if body[0] == '[': #the body is of form [function]{argument}.
        function_start = 1
        function_end = find_matching_bracket(body, function_start - 1)
        function = body[function_start:function_end] # Isolate the function.
        argument_start = function_end + 2
        argument_end = find_matching_bracket(body, argument_start - 1) # Isolate the argument.
        argument = body[argument_start:argument_end]
        return ('[' + replace_variable(function, variable, parameter) + ']' +
                '{' + replace_variable(argument, variable, parameter) + '}') # Replace recursively in both.
    if body[:7] == 'lambda(': # The body is itself a lambda.
        other_variable_start = 7
        other_variable_end = find_matching_bracket(body, other_variable_start - 1)
        other_variable = body[other_variable_start:other_variable_end]
        if other_variable == variable: # Ignore bounded instances of the variable.
            return body
        other_body_start = other_variable_end + 2
        other_body_end = find_matching_bracket(body, other_body_start - 1)
        other_body = body[other_body_start:other_body_end]
        return ('lambda(' + other_variable + ')(' +
                replace_variable(other_body, variable, parameter) + ')') # Replace recursively.
    # If non of the other ifs was entered, the body is just some variable.
    if body == variable:
        return parameter # This is the variable to be replaced.
    return body # This isn't, so we leave it unchanged.


def find_matching_bracket(string, bracket_index):
    # Finds the bracket matching the one in string[bracket_index] - the closing
    # bracket which matches the opening one or vice versa.
    brackets = 1
    i = bracket_index
    sign = -1 # By default, we assume it to be a closing bracket
    if (string[bracket_index] == '(' or
        string[bracket_index] == '[' or
        string[bracket_index] == '{'):
        sign = 1 # It's an opening bracket
    while brackets > 0:
        i += sign
        if ((string[bracket_index] == '(' and string[i] == ')') or
            (string[bracket_index] == '[' and string[i] == ']') or
            (string[bracket_index] == '{' and string[i] == '}') or
            (string[bracket_index] == ')' and string[i] == '(') or
            (string[bracket_index] == ']' and string[i] == '[') or
            (string[bracket_index] == '}' and string[i] == '{')):
            brackets -= 1
        if string[bracket_index] == string[i]:
            brackets += 1
    return i

def numeric_value(lam):
    # If a normal form lambda expression represents a number,
    # returns its numeric value calculated recursively.
    if normal_form('[IS_ZERO]{' + lam + '}') == 'TRUE':
        return 0
    return 1 + numeric_value(normal_form('[PREDECESSOR]{' + lam + '}'))

def calculate(lam):
    # Returns the numeric value of a lambda expression after simplification to a normal form.
    return numeric_value(normal_form(lam))
```

Java (Noam)

```
public class LambdaExp {
	// the lambda variable
	private char variable;
	// left part of the exp
	private LambdaExp func;
	// right part of the exp
	private LambdaExp input;
	// if exp is a character, his value.
	private char value;

	// if exp is inside closer, remove them.
	private String removeCloser(String exp){
		if(exp.charAt(0) != '(')
			return exp;
		int length;
		do {
			length = exp.length();
			int cNum = 1, i;
			for(i=1; i< exp.length() && cNum != 0; i++)
				if(exp.charAt(i) == '(')
					cNum++;
				else if(exp.charAt(i) == ')')
					cNum--;
			if(i==exp.length())
				exp = exp.substring(1, exp.length()-1);

		}while(length != exp.length() && exp.charAt(0) == '(');
		return exp;
	}

	// if exp build from two closers, return the start index of the second.
	private int sep(String exp){
		if(exp.length() == 1||exp.length() == 2)
			return 1;

		int cNum = 1, i;
		for(i=1; i< exp.length() && cNum != 0; i++)
			if(exp.charAt(i) == '(')
				cNum++;
			else if(exp.charAt(i) == ')')
				cNum--;

		return i;
	}

	// return LambdaExp's copy
	private LambdaExp copyTree(LambdaExp toCopy){
		if(toCopy == null)
			return null;
		return new LambdaExp(toCopy.value, toCopy.variable, copyTree(toCopy.func),
copyTree(toCopy.input));
	}

	// copy LambdaExp value to this
	private void copy(LambdaExp toCopy){
		this.variable = toCopy.variable;
		this.value= toCopy.value;
		this.func = copyTree(toCopy.func);
		this.input = copyTree(toCopy.input);
	}

	//replace all the instances of a variable with LambdaExp
	public void place(char variable, LambdaExp toPlace){
		if(this.variable == variable)
			return;
		if(this.value == variable){
			copy(toPlace);
			return;
		}
		if(func != null)
			func.place(variable, toPlace);
		if(input != null)
			input.place(variable, toPlace);
	}
```

```
public LambdaExp(String exp){
	exp = removeCloser(exp);
	if(exp.length() == 1){
		value = exp.charAt(0);
		return;
	}
	if(exp.charAt(0) == 'l'){
		variable = exp.charAt(1);
		exp = exp.substring(3);
		exp = removeCloser(exp);
	}
	int sep = sep(exp);
	func = new LambdaExp(exp.substring(0, sep));
	if(sep < exp.length())
		input = new LambdaExp(exp.substring(sep));
}

public LambdaExp(char value, char variable, LambdaExp func, LambdaExp input){
	this.value = value;
	this.variable = variable;
	this.func = func;
	this.input = input;
}


public String toString(){
	String exp = "";
	String expEnd = "";
	if(this.value != 0)
		return exp + this.value + expEnd;
	if(this.variable != 0){
		exp = exp + "l" + this.variable+".(";
		expEnd = ")" + expEnd;
	}
	if(this.input == null)
		return exp+ this.func.toString() + expEnd;
	return exp + "(" + this.func.toString() + ")(" + this.input.toString() + ")" + expEnd;
}

// make one step and return the new LambdaExp
public LambdaExp calcStep(){
	if(func == null)
		return this;
	if(func.variable == 0 || input == null){
		func = func.calcStep();
		return this;
	}
	char var = func.variable;
	func.variable = 0;
	func.place(var,input);
	if(variable == 0){
		if(func.input == null && func.value == 0)
			return func.func;
		return func;
	}
	input = null;
	return this;
}
```

```
	public LambdaExp(String exp){
		}

	// check if LambdaExp is in terminal state
	public boolean isTerminal(){
		if(this.func == null)
			return true;
		if(this.input != null && func.variable != 0)
			return false;
		return this.func.isTerminal();
	}
}

public class Main {
	public static void main(String[] args) {
		LambdaExp a = new LambdaExp(args[0]);
		System.out.println(a.toString());
		// while expression is not in terminal state
		while(!a.isTerminal()){
			//make one step
			a = a.calcStep();
			//print exp
			System.out.println(a.toString());
		}
	}
}
```

C# (Yaron)

```
//File: Term.cs
/* Ex #1 in Advanced Computational Models:
 * ---------------------------------------
 * Outer-most beta reduction of lambda calculus terms to a final normal form.
 * Submitted by: Yaron Harel 032340671
 * *******************************************************************/
```

[illegible]

```
//File: LineParser.cs
/* Ex #1 in Advanced Computational Models:
 * ---------------------------------------
 * Outer-most beta reduction of lambda calculus terms to a final normal form.
 * Submitted by: Yaron Harel 032340671
 * *******************************************************************/
using System;
using System.Linq;
using System.Text;


namespace MyOutermostBetaReduction
{
    //this class creates a tree of lambda-calculus terms from a line(string)
    public class LineParser
    {
        private string lineToParse = String.Empty;

        //property
        public string Line { get { return lineToParse; } }

        public const char LAMBDA_SIGN = '^';
        private int position = 0;

        private static bool isLegalChar(char c)
        {
            if (Char.IsLower(c) || c == '(' || c == ')' || c == '.' || c == LAMBDA_SIGN) return true;
            else return false;
        }

        //CTOR
        public LineParser(string s)
        {
            if (s != null)
            {
                //remove spaces and tabs
                lineToParse = s.Replace(" ", "");
                lineToParse = lineToParse.Replace("\t", "");

                //check string is legal as a term
                foreach (char c in lineToParse)
                    if (!isLegalChar(c)) throw new IllegalStringException("String contains illegal chars");
                if (!CheckParenthesisAndPointAndLambda(lineToParse)) throw new IllegalStringException("Illegal structure of term");
            }
        }

        //checks that a string is a legal term
        public bool CheckParenthesisAndPointAndLambda(string s)
        {
            bool inLambdaParams = false;
            bool sawVar = false;
            int counter = 0;
            foreach (char c in s)
            {
                switch (c)
                {
                    case LAMBDA_SIGN:
                        if (inLambdaParams == true) return false;
                        else
                        {
                            inLambdaParams = true;
                            sawVar = false;
                        }
                        break;
                    case '(':
                        if (inLambdaParams == true) return false;
                        else counter++;
                        break;
                    case ')':
                        if (inLambdaParams == true) return false;
                        else counter--;
                        break;
                    case '.':
                        if (inLambdaParams == false || sawVar == false) return false;
                        else inLambdaParams = false;
                        break;
                    default: //variable
                        sawVar = true;
                        break;
                }
                if (counter < 0) return false;
            }
            if (counter == 0 && inLambdaParams == false) return true;
            else return false;

        }

        //do the hard work of parsing, uses 'parse2' as a helper
        public Term Parse()
        {
            Term result2 = Parse2();
            Term result = result2;
            while (result2 != null)
            {
                result2 = Parse2();
                if (result2 != null) result = new Application(result, result2);

            }
            return result;
        }


        //do simple parsing of abstraction, variables and aplication
        public Term Parse2()
        {
            Term result = null;
            if (position < Line.Length)
            {
                if (Line[position] == LAMBDA_SIGN)
                {
                    position++;
                    result = ParseAbstraction();
                }
                else if (Line[position] == '(')
                {
                    position++;
                    result = Parse();
                    position++;//for ')'
                }
                else if (Char.IsLower(Line[position]))//   case 'a'-'z'= variable
                {
                    result = new Variable(Line[position]);
                    position++;
                }
            }

            return result;
        }
        //a method that is called by parse2, specifically for parsing of abstractions
        public Term ParseAbstraction()
        {
            Term result = null;
            Variable p = null;
            Term b = null;
            //move to parameters
            if (Char.IsLower(Line[position]))
            {

                p = new Variable(Line[position]);
                position++;
                if (Char.IsLower(Line[position])) b = ParseAbstraction();
                else if (Line[position] == '.')
                {
                    position++;
                    b = Parse();
                }
                else throw new IllegalStringException("illegal structure of term");
                result = new Abstraction(p, b);
            }
            return result;
        }
    }

    //illegal chars/structure of term
    public class IllegalStringException : Exception
    {
        public IllegalStringException(string msg) : base(msg) { }
    }

}




//File: Program.cs
/* Ex #1 in Advanced Computational Models:
 * ---------------------------------------
 * Outer-most beta reduction of lambda calculus terms to a final normal form.
 * Submitted by: Yaron Harel 032340671
 * *******************************************************************/

//Lambda-sign is: ^
//Variables are:'a'-'z'
//Abstraction format: for example: ^xyz.x(yz)
//Blanks are removed and are therefore meaningless
//Application, for example, to apply x on y, use: 'xy'

//examples:
//  ^fx.x  //this is for number 0
// ^fx.fx  //this is for number 1
// ^nfx.f(nfx) //this is for succesor-function



using System;
using System.Collections.Generic;



namespace MyOutermostBetaReduction
{


    //
    class Program
    {
        static void Main(string[] args)
        {
            string newLine = string.Empty;
            System.Console.WriteLine
                ("Welcome, please enter a beta-term\n(use {0} for lambda,'cls' to clear screen, or 'quit' to exit)\n\n",
                LineParser.LAMBDA_SIGN);
            LineParser lp = null;

            //loop for user input until 'quit' is typed
            while (!newLine.ToLower().Equals("quit"))
            {
                Console.Write("Term>>");
                newLine = System.Console.ReadLine();
                //clear screen
                if (newLine.ToLower().Equals("cls")) System.Console.Clear();
                else if (!newLine.ToLower().Equals("quit"))
                {
                    try{
                        //create a parser
                        lp=new LineParser(newLine);

                        //parse the newline into a tree of terms(variables, abstractions and applications)
                        Term result = lp.Parse();
                        if (result == null) continue;

                        System.Console.WriteLine("{0}", result);

                        //try to reduce(outermost) the result
                        Term afterReduction = result.TryReduction();

                        //if afterReduction,result are the same, we're in normal form
                        while (afterReduction!=result)
                        {
                            result = afterReduction;
                            afterReduction = afterReduction.TryReduction();
                            System.Console.WriteLine("{0}", result);//print next reduction result

                        }

                    }
                    catch(Exception e){ //exception of wrong syntax of term is catched
                        System.Console.WriteLine("error:{0}",e.Message);
                    }
                }
                System.Console.WriteLine();

            }

        }
    }
}
```