# CS:3820 Programming Language Concepts
# Fall 2016

## Homework 1

**Due:** Tuesday, Sep 13 by 11:59pm

This is a programming assignment in F#. You can do it alone or in a team two people. In the latter case, you are responsible for contacting other students and form a team.

The grade for the assignment will be given on an individual basis. For team submissions, both students must also submit an evaluation on how well they and their teammate performed as team members. Each evaluation is confidential and will be incorporated in the calculation of the grade.

*Do the assigned readings before starting attempting this assignment.* If you do not, you might find the assignment exceedingly hard.

*Be sure to review the syllabus for details about this course's cheating policy.*

**Submission Instructions** Provide your solutions in an F# script, that is, a *plain text file* called `hw1.fsx`. Dowwload the accompanying file `hw1.fsx` and complete it with your answers. Once you are done, submit that file as an attachment to a ***private note*** on Piazza:

1. Log it to Piazza and go to the Q&A section.
2. Click the blue New Post button.
3. Select Note as the Post Type.
4. Choose the Individual Student(s) / Instructor(s) option for Post to and type "Instructors".
5. Select the hw1 folder.
6. Write "Hw1 solution" in the Summary field.
7. If you did the homework with a team partner, in the Details field, list both your full name and that of your partner and allocate a number of credit points to each of you based on your individual level of contribution to the project. The sum of the points of the two team members should total 100. Examples:

   Same level of contribution by both members:

   ```
   John Doe   50
   Jane Smith 50
   ```

   First team member deserves most of the credit:

```
John Doe   70
Jane Smith 30
```

Second member was a no show:

```
John Doe   100
Jane Smith   0
```

*Your credit allocation should reflect each member's level of participation and effort, not his/her academic ability.* Also, do not consider just the amount of work done but also such factors as attendance to team meetings; responsiveness to emails; meeting of internal and external deadlines; flexibility; quality of work; preparation; cooperativeness; and so on.

8. Attach your solutions by selecting Insert > Insert file from the top bar of the Details field.

9. Click on the orange button to submit.

For team solutions, *only one member of the team should attach the solution.* The other should just provide the team member evaluation.

**Note:** Your `hw1.fsx` file *must compile* in F# Interactive with no syntax/type errors. You may receive a zero for any question whose code does not compile correctly.

# 1 Basic F# Programming

F# has a predefined list datatype. Strictly speaking, however, it is not necessary, For instance, integer lists could be simulated by the following user defined datatype

```
type ilist =
    E
  | L of int * ilist
```

where the constructor `E` stands for the empty list and the constructor `L` builds a list by adding a number in front of another list. Then one can represent, say, a list with elements 1,4,6,7, in that order, with the `ilist` value:

```
L(1, L(4, L(6, L(7, E))))
```

Implement the following functions manipulating `ilist` values. Use pattern matching and recursion as needed. You do not need, and should not use, any F# library functions unless instructed otherwise.

1. Write an F# function `sum: ilist -> int` which takes as input a list $l$ of strings and returns the sum of all the elements in $l$. For example, `sum (L(1, L(3, L(3, E))))` is `7`, while `sum E` is `0`.

2. Write an F# function `elem: int -> ilist -> int` which, given a *positive* integer $n$ and a list $l$, returns the $n^{th}$ element of $l$, if the list has at least $n$ elements, and fails (using `failwith`) with message `"Index out of bound"` otherwise.

   For example, `elem 2 (L(3, L(21, L(11, E))))` is `21`.

3. Write an F# function `isIn: int-> int list -> bool` which, given an integer $x$ and a list $l$, returns `true` if and only if $x$ occurs in $l$.

4. Write an F# function `remove: int -> ilist -> ilist` which, given an integer $x$ and a list $l$, "removes" all the occurrences of $x$ from $l$, that is, returns a list containing (in the same order) all and only the elements of $l$ that differ from $x$.

   For example, `remove 2 (L(1, L(2, L(3, L(3, L(2, E))))))` is `L(1, L(3, L(3, E)))`, `remove 5 (L(1, L(2, L(3, E))))` is `(L(1, L(2, L(3, E))))`.

5. Write an F# function `move: ilist -> ilist -> ilist` which takes two lists and returns the result of inserting the values of the first list into the second, in reverse order.

   For example, `move (L(1, L(2, L(3, E)))) (L(7,E))` is `L(3, L(2, L(1, L(7,E))))`.

6. Use function `move` to implement the function `reverse: ilist -> ilist` that returns the result of reversing its input list.

   For example, `reverse (L(1, L(2, L(3, E))))` is `L(3, L(2, L(1, E)))`.

## 2 Extending the expression language and its interpreter

File `hw1.fsx` contains a definition of the `expr` expression language and the evaluation function `eval` seen in class.

1. Extend `eval` to a new function `eval1` that also handles the additional binary operators `max`, `min`, `==` and `<`, respectively with the following behavior:

   - `max` returns the maximum of its arguments;
   - `min` returns the minimum of its arguments;
   - `==` returns 1 when its arguments have the same value, and returns 0 otherwise;
   - `<` returns 1 when the value of its first argument is smaller that the value of its second argument, and returns 0 otherwise.

2. We would like to extend the expression language with conditional expressions `If(`$e$`,` $e_1$`,` $e_2$`)` corresponding to Java's expression $e$ `?` $e_1$ `:` $e_2$ or F#'s conditional expression `if` $e$ `then` $e_1$ `else` $e_2$.

   Extend the `expr` datatype to a new datatype `expr2` with a new constructor `If` that takes three `expr2` arguments. Then extend `eval1` into a new function `eval2` that handles `If` expressions as follows: `If(`$e$`,` $e_1$`,` $e_2$`)` evaluates to the value of $e_1$ if $e$ has a non-zero value, and to the value of $e_2$ otherwise. Your extension should be such that $e_1$ and $e_2$ are evaluated only as needed, after $e$ has been evaluated.

## 3 An alternative expression language

In this problem we will consider an alternative datatype `aexpr` to represent arithmetic expressions with variables. The datatype should have constructors `CstI`, `Var`, `Add`, `Mul`, `Sub`, respectively for constants, variables, addition, multiplication, and subtraction. With that datatype, an expression like $x * (y + 3)$ is represented as

```
Mul(Var "x", Add(Var "y", CstI 3))
```

instead of

```
Prim("*", Var "x", Prim("+", Var "y", CstI 3)).
```

1. Define the datatype `aexpr`.

2. Define three F# variables `e1`, `e2` and `e3` of type `aepr` whose value corresponds to the expressions $v - (w + z)$, $2 * (v - (w + z))$, and $x + y + z + v$, respectively.

3. Write an F# function `toString: aexpr -> string` to format expressions as strings, with the binary operators written in infix format. For instance, it may format `Sub(Var "x", CstI 34)` as the string `"x - 34"`. For simplicity, put parentheses around any subexpressions, even when they are superfluous according to the standard precedence rules for arithmetic operators. Use the predefined function `string` to convert an integer value to its string representation.

   *Note* `toString` has very much the same structure as an eval function, although it needs no environment argument because it uses names of variables, not their value.

4. Write an F# function `simplify: aexpr -> aexpr` to perform expression simplification. Such simplifications are applied for instance in optimizing compilers. The `simplify` function relies on standard properties of arithmetic to reduce an expression to a simpler, but equivalent one. In particular, it should use the following simplification rules for an arbitrary expressions $e$:

$$\begin{aligned} 0 + e &\longrightarrow e \\ e + 0 &\longrightarrow e \\ e - 0 &\longrightarrow e \\ 1 * e &\longrightarrow e \\ e * 1 &\longrightarrow e \\ 0 * e &\longrightarrow 0 \\ e * 0 &\longrightarrow 0 \\ e - e &\longrightarrow 0 \end{aligned}$$

   This means, for instance, that it should reduce $x + 0$, $1 * x$, and $(1 + 0) * (x + 0)$ all to $x$.

   Your implementation shoud be such that the result of `simplify` is an expression that cannot be simplified further by it. That is, it should be such that, for all expressions $e$,

   `(simplify (simplify` $e$`)) = (simplify` $e$`)` .

   *Notes:* Pattern matching is your friend. Don't forget cases where you cannot simplify anything. Repeated (recursive) passes of `simplify` might be needed to produce an expression that cannot be simplified further. Extra credit will be given to solutions that minimize the number of passes.