# IIT CS440: Programming Languages and Translators

Homework 1: OCaml Continued

Prof. Stefan Muller
TA: Xincheng Yang

Out: Thursday, Feb. 4
Due: Thursday, Feb. 11, 11:59pm CST

**This assignment contains 2 written task(s) and 7 programming task(s), for a total of 40 points, in addition to a maximum of 5 bonus points.**

The same rules on submission, collaboration, compilation and testing apply as on HW0 (with one exception below), so make sure you understand and follow them. Remember that you may not modify function signatures, including to make non-recursive functions recursive (you may, however, add recursive helper functions unless the instructions for the problem or section say otherwise).

For this assignment, there are three .ml files; submit each, renamed in the same way as before (so, `tailrec.ml` would become `Doe_John_440_tailrec.ml`). There's no real written component on this assignment other than stating your collaborators/outside sources and how long you took, so you can just answer these in a comment at the bottom of `expression.ml` instead of submitting a separate pdf.

## 1 Tail Recursion

**NOTE: All functions in this section must be tail-recursive. Make sure you understand what that means.**

For this section, put your answers in `tailrec.ml`.

**Task 1.1 (Programming, 5 points)**.

Write a **tail-recursive** function `fact : int -> int` that calculates the factorial of an argument $n$ (that is, $n!$). Remember that

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1$$

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 \\ &\dots \end{aligned}$$

**Task 1.2 (Programming, 7 points)**.

Write a **tail-recursive** function `split : 'a list -> 'a list * 'a list` that takes a list and returns a pair of lists such that for all `l`, `(fst (split l)) @ (snd (split l))` is a permutation of `l`. That is, it splits a list into two lists and returns a pair of the two lists. **The lengths of the two lists should differ by at most one**.

For example, `split [1;2;3;4;5]` can return `([1;2;3], [4;5])` or `([1;3;5], [2; 4])` or `([5;3], [4;2;1]`

but not `([], [1;2;3;4;5])` or `([1;2;3], [3;4;5])` or `([1], [5])`.

**Bonus Task 1.3 (Programming, 2 points)**.

Write a **tail-recursive** version of `List.fold_right` *without* using `List.fold_left` or `List.rev` (or re-implementing either of these; your function should make just one pass over the list). Other than tail recursion, your function should behave identically to `List.fold_right`.

*Hint: Try changing the function you pass to the recursive application.*

*Note: This, like other bonus tasks we'll occasionally include in assignments, is quite difficult and is worth a small number of bonus points. Try it for an extra challenge if you want (probably after completing the rest of the assignment).*

# 2 Trees

For this section, put your answers in `trees.ml`.

Higher-order functions aren't just for lists! Recall the algebraic data type of binary trees from lecture:

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

In this section, you'll implement and use higher-order functions over trees.

As an example, we implemented the function `tree_fold : 'a tree -> 'b -> ('a -> 'b -> 'b -> 'b)` that folds over trees like `List.fold_right` folds over lists. For example,

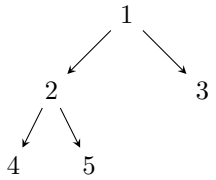
`tree_fold (Node (v1, Node (v2, Leaf, Leaf), Leaf)) u f`

is `f v1 (f v2 u u) u`. Note that our implementation isn't tail-recursive. The Cornell book (linked from the course website) gives a tail-recursive version.

**Task 2.1 (Programming, 5 points)**.

Implement the function `tree_map : 'a tree -> ('a -> 'b) -> 'b tree` that returns a tree with the same structure as the original tree, but with the given function applied to every node. Use `tree_fold`. *Do not* use recursion.

There are various ways of *traversing* a tree to flatten it. Consider the tree below.




An *in-order traversal* goes down the left subtree of a node first, then visits the node itself, then the right subtree. A in-order traversal of the above tree would visit nodes in the order 4, 2, 5, 1, 3. You can think of this as basically visiting the nodes left-to-right as they're drawn on the page.

A *pre-order traversal* visits the node itself, then the left subtree, then the right subtree. A pre-order traversal of the above tree visits the nodes in the order 1, 2, 4, 5, 3.

**Task 2.2 (Programming, 5 points)**.

Implement the function `inorder : 'a tree -> 'a list` that lists the nodes of a tree in the order given by an in-order traversal. (So, `inorder` applied to the above tree would give `[4;2;5;1;3]`.) Use `tree_fold`. *Do not* use recursion.

**Task 2.3 (Programming, 5 points)**.

Implement the function `preorder : 'a tree -> 'a list` that lists the nodes of a tree in the order given by an pre-order traversal. (So, `preorder` applied to the above tree would give `[1;2;4;5;3]`.) Use `tree_fold`. *Do not* use recursion.

# 3 Arithmetic Expressions

For this section, put your answers in `expression.ml`.

In this section, we'll work with a datatype that represents arithmetic expressions of a single variable. This is actually defined as two data types:

```
(* Binary operators. *)
type binop = Add | Sub | Mul | Pow ;;

type expression =
  | Num of float
  | Var
  | Binop of binop * expression * expression
  | Neg of expression
;;
```

`Num a` represents the floating-point number `a`. `Var` represents the only variable, which we'll call $x$. `Binop o e1 e2` represents a binary operation on subexpressions `e1` and `e2`. The binary operation is given by `o` of type `binop` which can be `Add`, `Sub`, `Mul` or `Pow`. `Neg e` is the negation of `e`.

For example, we could represent $-3.0x^2 + x + 2.0$ as

```
Binop (Add,
       Binop (Add,
              Binop (Mul, Neg (Num 3.0), Binop (Pow, Var, Num 2.0)),
              Var),
       Num 2.0)
```

(There are other ways we could represent it too; as we'll learn soon when we start talking about parsing, the grammar for expressions is *ambiguous*.)

## 3.1 Parsing expressions

We've provided functions (below the definition of the data types, in a large block of code you can ignore) for parsing strings into expression datatypes. You may find this helpful when testing your code, unless you particularly like manually typing things like the expression above. The parser is reasonably robust, but not great, so you may need to fiddle around a bit to get your strings in the right format. We can get the above expression by running

```
parse "~3.0*x^2 + x + 2.0";;
```

Note that we use `~` instead of `-` to represent negation. This lets the parser distinguish between negation and subtraction. We also need to explicitly include the `*` between `~3.0` and `x^2` rather than just concatenating the two like we do when writing math normally.

**Continued on next page**

## 3.2 Your tasks

**Bonus Task 3.1 (Programming, 1 points).**

Implement a fold function over expressions. Fold for lists and trees only needed to take 2 arguments (in addition to the list and tree being folded over) because there were two things a list or tree could be: nil and cons for lists, or leaf or node for trees. Fold for lists needs to know two things: what to do on nil (return the base case) and what to do on a cons (apply a function to the head and the recursive result of the tail). Each of the "things" needs to return the same type.

Fold for expressions needs to take four things, all of which take a different kind of expression and return a `'a`.

1. What to do on a `Num`. This is a `float -> 'a` because it needs to know the number.
2. What to do on a `Var`. Like the base case for lists and trees, this is just a `'a` because there's no other information at a `Var`.
3. What to do on a `Binop`. This is a `binop -> 'a -> 'a -> 'a` because it needs to know the operation and the result of folding over both subexpressions.
4. What to do on a `Neg`. This is a `'a -> 'a` because it just needs to know the result of folding over the subexpression.

**Task 3.2 (Programming, 5 points).**

Write a function `contains_var : expression -> bool` that returns true if and only if an expression contains a variable. Your implementation may be recursive (it need not be tail-recursive) or may be non-recursive and use `fold_expr` (even if you didn't do that task; we'll test your implementation with a working version of `fold_expr`.)

**Task 3.3 (Programming, 8 points).**

Write a function `evaluate : expression -> float -> float`. The application `evaluate e v` should substitute `v` for $x$ in the given expression and evaluate it to a float. For example,

- `evaluate (parse "~3.0*x^2 + x + 2.0") 0.0 = 2.0`
- `evaluate (parse "~3.0*x^2 + x + 2.0") 1.0 = 0.0`
- `evaluate (parse "~3.0*x^2 + x + 2.0") 2.0 = -8.0`

Your implementation may be recursive (it need not be tail-recursive) or may be non-recursive and use `fold_expr` (even if you didn't do that task; we'll test your implementation with a working version of `fold_expr`.)

**Task 3.4 (Written, 0 points).**

How long (approximately) did you spend on this homework, total, not including bonus questions? (You can tell us how long you spent on bonus questions, if applicable, separately if you want.) Your honest feedback will help us with future homeworks.

**Task 3.5 (Written, 0 points).**

Who, if anyone, did you collaborate with (and in what way), and what outside sources, if any, did you consult in working on this homework?

**Continued with one more bonus task on next page**

### 3.3 Bonus Task: Differentiation

If you'd like, you can practice some Calc I and take the derivative of expressions.

Remember, if $e$, $e_1$ and $e_2$ are expressions potentially containing the variable $x$:

$$\begin{array}{lcll}
\frac{d}{dx}a & = & 0 & \text{(Constants have 0 derivative)} \\
\frac{d}{dx}x & = & 1 & \text{(Variable)} \\
\frac{d}{dx}(e)^a & = & a \cdot (e)^{(a-1)} \cdot \frac{d}{dx}e & \text{(Power rule, chain rule)} \\
\frac{d}{dx}(e_1 + e_2) & = & \frac{d}{dx}e_1 + \frac{d}{dx}e_2 & \text{(Sum Rule)} \\
\frac{d}{dx}(e_1 - e_2) & = & \frac{d}{dx}e_1 - \frac{d}{dx}e_2 & \text{(Sum Rule)} \\
\frac{d}{dx}(e_1 \cdot e_2) & = & e_2\frac{d}{dx}e_1 + e_1\frac{d}{dx}e_2 & \text{(Product Rule)}
\end{array}$$

**Bonus Task 3.6 (Programming, 2 points)**.

Write a function `derivative : expression -> expression` that returns another expression that is the derivative of the given expression using the rules above. Note that the rule for power above only applies if $a$ is a constant (i.e., does not depend on $x$). There's another, more complicated rule for the derivative of $e_1^{e_2}$ where $e_1$ and $e_2$ both contain $x$, but you're not required to use it. Instead, if the exponent depends on $x$, raise the exception `NotPolynomial`.

*For 1 bonus point*, implement `derivative` recursively (it need not be tail-recursive).
*For 2 bonus points*, implement `derivative` using `fold_expr` (this is slightly trickier than you might think).