# Assignment 4
# COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

**Due Date: 14 Nov, 2013**

Your homework is due at the beginning of class on 14 Nov, 2013. All code files must be submitted electronically, and your program must compile.

Q1.20 points In this question, we are rewriting some basic functions from the list library tail-recursively. See the file `cont.ml` in the directory `continuations`.

**a (10 points)** Write `filter` tail-recursively by providing an implementation for

```
aux_filter : ('a -> bool) -> 'a list -> ('a list -> 'b) -> 'b
```

`filter` takes in three input arguments: the predicate of type `'a -> bool`, a list of type `'a list`, and a continuation of type `'a list -> 'b`. The continuation expects as input the result list it has built so far and returns elements of type `'b`.

**b (10 points)** Write `foldr` tail-recursively by providing an implementation

```
aux_foldr : ('a -> 'b -> 'b) -> 'a list -> 'b -> ('b -> 'c) -> 'c
```

As `foldr`, it takes in a function of type `'a -> 'b -> 'b`, a list of type `'a list`, and a base element of type `'b`. In addition, it also takes in a continuation of type `'b -> 'c` which builds incrementally the final result. It takes in as input the result built so far.

Q2 15 points We revisit the implementation of `change` which when given a list of coins (ordered in increasing value) and an amount returns a list of coins which add up to the expected amount. It might fail, if no change can be given.

Implement the revised function `change coin_list amt sc fc` which has type

```
int list -> int -> (int list -> 'a) -> (unit -> 'a) -> 'a
```

The function `change` takes in four arguments: a coin list, an amount, a success continuation, and a failure continuation.

The success continuation builds the list of coins which add up to the expected amount. It is a function `int list -> 'a`. The success continuation `sc` is called upon success, i.e. if our amount it `0`.

The failure continuation is used to handle backtracking. It is used instead of exceptions which we used in class.

HINT: First modify the function `change` which uses exceptions for backtracking such that it is building the result tail-recursively using a success continuation. This function `change_attempt` should have type `int list -> int -> (int list -> 'a)-> 'a`

Q3. 25 points **Lazy functional programming**

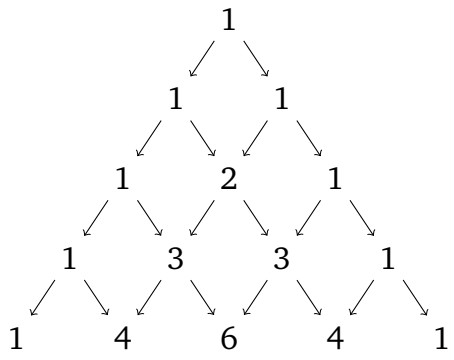
In this question you are going to produce the infinite stream of Hamming numbers.These are numbers that have as prime divisors 2 or 3 or 5 but no other prime divisors. The "or" in the previous sentence is the "logical or" so of course we include numbers that have both 2 and 3 or 2 and 5 and so on as Hamming numbers. Hamming numbers are given by $2^i3^j5^k$ where $i, j$ and $k$ are each greater than or equal to zero. The first few Hamming numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, ....

**a (15 points)** Write a function `merge` that takes two streams of integers in increasing order and with no duplicates and merges them into a single stream containing all the elements of both the streams again in increasing order and with no duplicates. Add this function to the file `stream.ml`.

**b (10 points)** Use the function from (a) to implement the lazy stream `hamming_series`. It is imperative that your stream be built lazily enough that when we listify prefixes of it it does not diverge. The solution is very easy if you think lazily. There is no need for "extra variables" to keep "track of" anything. Add your function `hamming_series` to the file `series.ml`.

Q4. 40 points Pascal's triangle is a number triangle with numbers arranged in staggered rows such that it contains binomial coefficient.

B. Pascal after whom the triangle is named, was the first person to discover the importance of a wide variety of number patterns which exist in this number triangle. In particular, we will explore its diagonal property. We show Pascal's triangle below.




The diagonals of Pascal's triangle contain a number of patterns:

– the first diagonals going along the left and right edges contain only 1's.

- the second diagonal (going from left to right) next to the edge diagonals contain the natural numbers in order.
- Moving inwards, the next pair of diagonals contain the triangular numbers in order; etc.

In addition we can observe that the $i + 1$ diagonal in Pascals triangle correspond to the partial sums of the stream describing the $i$ diagonal.

We will build a Pascal triangle in multiple steps lazily. We can think of the Pascal triangle as a stream of diagonals where each diagonal is itself a stream. Hence a Pascal triangle can be viewed as a stream of streams!

**a (10 points)** First implement a function `psums:int Stream.str -> int Stream.str` which takes a stream of integers and produces a stream of the partial sums.

| Input | Output |
|---|---|
| `1 1 1 1 ...` | `1 2 3 4 ...` |
| `1 2 3 4 ...` | `1 3 6 10 ...` |

**b (10 points)** We now want to generate all diagonals in the Pascal triangle, starting with the stream of ones as a first diagonal. The $i + 1$ diagonal in the triangle corresponds to the partial sums of the stream described by the $i$-th diagonal.

**c (15 points)** Next, we want to turn the stream of diagonals into a pascal triangle which is represented as a stream of rows where each row is represented as a list. To accomplish this, it is helpful to write two functions

```
getNth : int -> 'a Stream.str -> 'a
row : int -> int Stream.str Stream.str -> int list
```

The function, `getNth` takes in an integer `i` and a stream and returns the `i`-th element of the stream.

The function `row` takes in an integer `k` and a stream of diagonals and generates the `k`-th row in the triangle.

**d (5 points)** Write a function `triangle : int Stream.str Stream.str -> int list Stream.str` which takes in a stream of diagonals and returns a stream of rows.