Development 8 - Exercises
Unit 3

For these exercises you might need to use `Tuple<a, b>` seen in class:

```
interface Tuple<a, b> {
  fst: a
  snd: b
}
```

Exercise 1:

Implement a function

```
let splitAt = <a>(i: number) => (l: List<a>): Tuple<List<a>, List<a>>
```

that splits the list into two lists, the first one containing all the elements of `l` from position 0 to position `i` included, and the second one containing all the remaining elements. The two resulting lists are returned in a tuple. For example `split 3 [3;5;4;-1;2;2] = fst: [3;5;4;-1], snd: [2;2]` .

Exercise 2:

Implement a function

```
let merge = <a>(l1: List<a>) => (l2: List<a>): List<a>
```

that merges together two **sorted** lists into a single sorted list.

Exercise 3:

Implement a function

```
let mergeSort = <a>(l: List<a>): List<a>
```

implement the Merge Sort algorithm. The Merge Sort returns the list itself when the list has only one element. Otherwise it splits `l` in two lists, one containing the elements between position 0 and `l.length / 2` (Typescript does the floating point division so use `Math.floor` on the result to convert it to the result of the integer division), the other containing the elements from `l.length`

`/ 2 + 1` until the end. Merge Sort is called recursively on the two lists. After this step use the function `merge` above to merge the two sorted lists.

Exercise 4:

Implement a function

```
let mergeSort = <a>(l:  List<a>):  List<a>
```

The Merge Sort returns the list itself when the list has only one element. Otherwise it splits `l` in two lists, one containing the elements between position 0 and `l.length / 2` (Typescript does the floating point division so use `Math.floor` on the result to convert it to the result of the integer division), the other containing the elements from `l.length / 2 + 1` until the end. Merge Sort is called recursively on the two lists. After this step use the function `merge` above to merge the two sorted lists.

Exercise 5:

Implement a function

```
let eval = (expr1:  Expr):  Expr
```

The function allows to evaluate arithmetic expressions. An arithmetic expression is defined as a discriminated union with the following cases:

- An atomic value. This is just a number
- The sum of two expressions.
- The difference of two expressions.
- The product of two expressions.
- The division of two expressions.

The function `eval` for an atomic value simply returns the atomic value. For the other cases it simply evaluates recursively the arguments of the operator and then combine the two results together. For example, evaluating the sum of two expressions `expr1 + expr2` recursively evaluates `expr1` and `expr2` and then sum ups the two atomic values together. The other operators behave analogously.

Exercise 6:

Implement a function

```
let eval = (expr1:  Expr) => (stack:  List<string, number>):  Expr
```

Extend the previous version of the function `eval` by supporting also variables. Variables are another case of the union. Evaluating a variable means looking it up in the `stack`. A `stack` is a list of tuple mapping a variable name to its value. If the lookup is successful `eval` returns the value of

the variable, otherwise it throws an error.

Exercise 7:

Implement a function

```
let run = (program:  List<Satement>) => (stack:  List<Tuple<string, number>>)
```

The function interprets a program from a small imperative language that can consists of the following statements, expressed as a discriminated union.

- Variable assignment. A variable assignment contains two arguments: the name of the variable, and an expression that is used to assign a value to it. First the expression must be evaluated and then, if the variable already exists in `stack`, its value is updated, otherwise we add the variable name and the value in `stack`.
- Print to output. This data structure contains an expresion, which is evaluated and then prints the value on the console.

The evaluation stops when the list of statements is empty.