Functional Programming

Computation

Exercise 1. By "traverse a tree" below we mean: write a function that takes a tree and returns a list of values in the nodes of the tree.

1. Write a function (of type `btree -> int list`) that traverses a binary tree: in prefix order – first the value stored in a node, then values in all nodes to the left, then values in all nodes to the right;
2. in infix order – first values in all nodes to the left, then value stored in a node, then values in all nodes to the right (so it is "left-to-right" order);
3. in breadth-first order – first values in more shallow nodes.

Exercise 2. Turn the function from ex. 1 point 1 or 2 into continuation passing style.

Exercise 3. Do the homework from the end of last week slides: write `btree_deriv_at`.

Exercise 4. Write a function `simplify: expression -> expression` that simplifies the expression a bit, so that for example the result of `simplify (deriv exp dv)` looks more like what a human would get computing the derivative of `exp` with respect to `dv`:

Write a `simplify_once` function that performs a single step of the simplification, and wrap it using a general `fixpoint` function that performs an operation until a *fixed point* is reached: given f and x, it computes $f^n(x)$ such that $f^n(x) = f^{n+1}(x)$.

Exercise 5. Write two sorting algorithms, working on lists: merge sort and quicksort.

1. Merge sort splits the list roughly in half, sorts the parts, and merges the sorted parts into the sorted result.
2. Quicksort splits the list into elements smaller/greater than the first element, sorts the parts, and puts them together.