

Programming Paradigms

Seminar 2

weeks (11-15 October 2021 and
18-22 October 2021)

Please present the Seminar 1. Please start to work in class on Seminar 2 and complete it at home (until the deadline).

DEADLINE for Seminar 2 is in the following weeks: 25-29 October 2021 and 1-9 November 2021. If you present it in week: 8-12 November 2021 you can get maximum grade 7 on this assignment. After 12 November 2021 you will get the grade 0 for this assignment.

Exercise 1. (Combinations) The number of combinations of r items taken from n is equal to the number of subsets of size r that can be made from a set of size n . This is written $\binom{n}{k}$ in mathematical notation and pronounced “ n choose k ”. It can be defined as

follows using the factorial: $\binom{n}{k} = \frac{n!}{k! * (n - k)!}$, which leads naturally to the following Oz

function:

```
declare
fun {Comb N K}
  {Fact N} div ({Fact K} * {Fact N-K})
end
```

For example, {Comb 10 3} is 120, which is the number of ways that 3 items can be taken from 10. This function is not very efficient because it might require calculating very large factorials, but it is probably the simplest. Write a more efficient version of Comb.

(a) As a first step, use the following alternative definition to write a more efficient function:

$$\binom{n}{k} = \frac{n * (n - 1) * \dots * (n - k + 1)}{k * (k - 1) * \dots * 1}$$

Calculate the numerator and denominator separately and then divide them. Make sure that the result is 1 when $k = 0$. In other words, the result of {Comb N 0} should be 1.

(b) The above definition may overflow for large numbers. To prevent this, we could compute the numerator and denominator progressively, allowing for division to take place incrementally, as follows: $(n/1) * (n-1)/2 * (n-3)/3 * \dots * (n-k+1)/k$. Write a recursive definition Comb that has the above computational characteristic.

Exercise 2. (Reverse) Given a list $L=[x_1 x_2 \dots x_N]$, its reverse would $[x_N \dots x_2 x_1]$. One standard way of writing such a reverse function is shown below. If you execute `{Reverse ['I' 'want' 2 go 'there']}`, it should return/display `[there go 2 want 'I']`.

```
declare
fun {Append L1 L2}
  case L1 of
    nil then L2
  [] H|T then H|{Append T L2}
  end
end
```

```
fun {Reverse L1}
  case L1
  of nil then nil
  [] H|T then {Append {Reverse T} [H]}
  end end
{Browse {Reverse ['I' 'want' 2 go 'there']}}
```

However, this implementation has quadratic time complexity since each `Append` call takes time that is linear to its first argument. Using a second parameter that contains the partially reversed list, write another definition of `Reverse` that is more efficient, without using `Append` calls.

Exercise 3. (Infinite Structures) One of the advantage of lazy evaluation is the support for infinite data structures. This may be useful for generating a list of infinite prime numbers. Given below is a function to generate a list of prime numbers. If you run it, it will go into a **loop**. Modify it to make it lazy. After that, use it to write a function {GetAfter N} that would return the first prime number after a given value.

```
fun {Sieve L} case L of
    nil then nil
    [] H|T then H|{Sieve {Filter T H}}
end end
fun {Filter L H}
    case L of
        nil then nil
        [] A|As then if (A mod H)==0 then {Filter As H}
                      else A|{Filter As H} end
    end end
fun {Prime} {Sieve {Gen 2}} end
fun {Gen N} N|{Gen N+1} end
{Browse {Prime}}
```

Exercise 4. (Binary Search Tree) Consider a binary search tree (BST) data structure of the following form:

`<BTree> ::= leaf(<Int>) | node(<Int>, <BTree>, <BTree>)`

The elements of this tree are assumed to be sorted if it has the following properties:

- a. It is leaf node with an integer value.
- b. It is a node where the left and right subtrees are sorted, and
all integers in its left subtree $<$ integer in current node
integer in current node \leq all integers in its right subtree.

A simple example of this BST data structure is the following :

```
N1 = node(3 nil N2)
N2 = node(5 nil nil)
```

Implement the following methods that manipulates/queries binary search tree :

- (i) `Insert :: {<BTree>, <Int> } → <BTree>`
// inserts an integer into a binary search tree (BST)
- (ii) `Smallest :: <BTree> → <Int>`
// returns the smallest element of the given BST
- (iii) `Biggest :: <BTree> → <Int>`
// returns the largest element of the given BST
- (iv) `IsSortedBST :: <BTree> → <Bool>`
// checks if the given tree has the sortedness property
// Hint : may use Smallest/Biggest functions for this method