Programming Paradigms Fall 2022
Homework Assignment №3

Innopolis University

November 24, 2022

About this assignment

You are encouraged to use homework as a playground: don't just make some code that solves the problem, but instead try to produce a readable, concise and well-documented solution.

Try to divide a problem until you arrive at simple tasks. Create small functions for every small logical task and combine those functions to build a complex solution. Try not to repeat yourself: for every logical task try to have just one small function and reuse it in multiple places.

The assignment is split into mutliple exercises that have complexity specified in terms of stars ($\star$). The more stars an exercise has the more difficult it is. Exercises with three or more stars ($\star\star\star$) might be really challenging, so please make sure you are done with simple exercises before trying out the more difficult ones.

The assignment provides clear instructions and some examples. However, you are welcome to make the result extra pretty or add more functionality as long as it does not change significantly the original problem and does not make the code *too complex*.

Submit a homework with all solutions as a single file. The file should work by loading it in DrRacket development environment and simply running it.

This homework will be graded out of 100 point:

1. 60 points for the correctness
2. 40 points for the coding style, accounting for
   - idiomatic use of the functional programming paradigm,
   - code structure,
   - correctly used abstractions,
   - error handling,
   - idiomatic naming of variables and functions,
   - helpful and concise comments, and
   - overall documentation, explanation of taken approaches.

This homework also contains an extra credit exercise.

1 Binary trees

Consider the following representation of binary trees:

- `empty` — an empty binary tree;
- `node(Value, Left, Right)` — an node with a value `Value` and two subtrees (`Left` and `Right`).

Exercise 1.1 (⋆, 3 points)**.** Implement predicate `tree/1` that checks whether a given term is a valid tree.

```
?- tree(empty)
true

?- tree(node(1, empty, node(2, node(3, empty, empty), empty)))
true

?- tree(node(A, empty, node(B, node(C, empty, empty), empty)))
true
```

Exercise 1.2 (⋆⋆, 6 points)**.** Implement predicate `containedTree/2` such that `containedTree(Tree1, Tree2)` is `true` when `Tree1` is contained in `Tree2`:

1. an empty tree is contained in any tree;
2. a non-empty tree is contained in another non-empty tree when they have the same root and both subtrees are contained in the other subtrees respectively;

```
?- subtree(Tree, node(3, node(1, empty, node(2, empty, empty)), node(4, empty, empty)))
Tree = empty
Tree = node(3,empty,empty)
Tree = node(3,empty,node(4,empty,empty))
Tree = node(3,node(1,empty,empty),empty)
Tree = node(3,node(1,empty,empty),node(4,empty,empty))
Tree = node(3,node(1,empty,node(2,empty,empty)),empty)
Tree = node(3,node(1,empty,node(2,empty,empty)),node(4,empty,empty))
```

Exercise 1.3 (⋆, 3 points)**.** Implement predicate `from/2` such that `from(Start, List)` is `true` when `List` is a finite list consisting of consecutive numbers `Start, Start+1, Start+2, ...`.

```
?- from(1, List)
List = []
List = [1]
List = [1, 2]
List = [1, 2, 3]
...
```

Exercise 1.4 (⋆⋆, 6 points)**.** Implement predicate `preorder/2` such that `preorder(Tree, List)` is `true` when `List` contains exactly of values from `Tree` in preorder traversal.

```
?- preorder(node(1, node(2, node(3, empty, empty), empty), node(4, empty, empty)), [3,2,1,4])
false

?- preorder(node(1, node(2, node(3, empty, empty), empty), node(4, empty, empty)), Values)
Values = [1, 2, 3, 4]

?- preorder(Tree, [1,2,3])
Tree = node(1,empty,node(2,empty,node(3,empty,empty)))
Tree = node(1,empty,node(2,node(3,empty,empty),empty))
Tree = node(1,node(2,empty,empty),node(3,empty,empty))
Tree = node(1,node(2,empty,node(3,empty,empty)),empty)
Tree = node(1,node(2,node(3,empty,empty),empty),empty)

?- from(1, _List), preorder(Tree, _List)
Tree = empty
Tree = node(1,empty,empty)
Tree = node(1,empty,node(2,empty,empty))
...
```

Exercise 1.5 (⋆, 3 points)**.** Implement predicates `leq/2` and `less` extending built-in comparison predicates to work with positive and negative infinities:

```
?- leq(-infinity, 4)
true

?- less(3, +infinity)
true

?- leq(4, 3)
false
```

Exercise 1.6 (⋆⋆, 6 points)**.** Implement predicate `bst/1` that checks that a tree is a binary search tree (BST).

```
?- bst(node(3, node(1, empty, node(2, empty, empty)), node(4, empty, empty)))
true

?- bst(node(5, node(1, empty, node(2, empty, empty)), node(4, empty, empty)))
false

?- preorder(Tree, [3,1,2,4]), bst(Tree)
Tree = node(3, node(1,empty,node(2,empty,empty)), node(4,empty,empty))

?- from(1, _List), preorder(Tree, _List), bst(Tree)
Tree = empty
Tree = node(1,empty,empty)
Tree = node(1,empty,node(2,empty,empty))
Tree = node(1,empty,node(2,empty,node(3,empty,empty)))
...
```

Exercise 1.7 (⋆⋆, 6 points). Implement predicate `bstInsert/3` such that `bstInsert{Value, Before, After}` is `true` when `After` is a binary search tree produced from `Before` by inserting `Value` into it.

```
?- preorder(Before, [3]), bst(Before), bstInsert(1, Before, After)
After = node(3,node(1,empty,empty),empty),
Before = node(3,empty,empty)

?- preorder(Before, [4,1]), bst(Before), bstInsert(3, Before, After)
After = node(4,node(1,empty,node(3,empty,empty)),empty),
Before = node(4,node(1,empty,empty),empty)
```

Exercise 1.8 (⋆⋆, 6 points). Implement predicate `bstMin/2` such that `bstMin{Tree, Min}` is `true` when `Min` is the minimum value stored in `Tree`. Implement predicate `bstMax/2` able to find the maximum value similarly.

```
?- preorder(Tree, [4,2,1,3,6,5,7]), bst(Tree), bstMin(Tree, Min), bstMax(Tree, Max)
Max = 7,
Min = 1,
Tree = node(4,
          node(2,
            node(1,empty,empty),
            node(3,empty,empty)),
          node(6,
            node(5,empty,empty),
            node(7,empty,empty)))
```

Exercise 1.9 (⋆ ⋆ ⋆, +0.5% extra credit). Implement predicate `bstDelete/3` such that `bstDelete{Value, Before, After}` is `true` when `After` is a binary search tree produced from `Before` by deleting `Value` from it. Note that this cannot be produced directly from `bstInsert/3`, since that predicate can only insert/delete leaves, not internal nodes.

```
?- preorder(Before, [3,1,2,4]), bst(Before), bstDelete(3, Before, After)
After = node(2,node(1,empty,empty),node(4,empty,empty)),
Before = node(3, node(1,empty,node(2,empty,empty)), node(4,empty,empty))

?- preorder(Before, [3,1,2,4]), bst(Before), bstDelete(1, Before, After)
After = node(3,node(2,empty,empty),node(4,empty,empty)),
Before = node(3, node(1,empty,node(2,empty,empty)), node(4,empty,empty))
```

2 Simple expressions

Exercise 2.1 (⋆, 3 points). Implement predicate `expr/1` that checks if a given term is a valid arithmetic expression:

1. a number;
2. a term $X + Y$ where both X and Y are valid expressions;
3. a term $X * Y$ where both X and Y are valid expressions;

```
?- expr(2+3*4)
true

?- expr((2+X)*4)
false
```

Exercise 2.2 (⋆⋆, 6 points)**.** Implement predicate `expr/2` such that `expr(Expr, Values)` is `true` when `Expr` is an expression term that uses each element (number) from `Values` once (in that order). You may assume that `Values` has known shape (i.e. length):

```
?- expr(Expr, [1,2])
Expr = 1+2
Expr = 1*2

?- expr(Expr, [1,1,1,1])
Expr = 1+(1+(1+1))
Expr = 1+(1+1*1)
Expr = 1+(1+1+1)
Expr = 1+(1*1+1)
...
```

Exercise 2.3 (⋆, 3 points)**.** Implement predicate `equation/2` such that `equation(Values, Result=Expr)` is `true` when `Expr` is an expression term that uses each element (number) from `Values` once (in that order) and `Result` is the number equal to the computed value of `Expr`.

```
?- equation([1,2], Equation)
Equation = (3=1+2)
Equation = (2=1*2)

?- equation([1,1,1], Equation)
Equation = (3=1+(1+1))
Equation = (2=1+1*1)
...
```

Exercise 2.4 (⋆⋆⋆, 9 points)**.** Implement predicate `equations/2` such that `equations(Values, Equations)` is `true` when `Equations` is a list of **all distinct equations** that can be produced with `Values`.

```
?- equations([1,2,3], Equations)
Equations = [
  6=1+(2+3),
  7=1+2*3,
  6=1+2+3,
  5=1*2+3,
  5=1*(2+3),
  6=1*(2*3),
  9=(1+2)*3,
  6=1*2*3
  ]
```