Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Module: CH-232
Date: 2022-11-25
Due: 2022-12-02

ICS 2022 Problem Sheet #12

Problem 12.1: *bnf grammar for integer numbers* (3 points)

In modern programming languages (e.g., Python, Haskell, Rust), integer numbers can be written in different bases by prefixing the digits with a prefix indicating the base of the number. By way of examples:

```
123                 // decimal number 123, empty prefix
0x7b                // hexadecimal number representing 123, prefix 0x
0o173               // octal number representing 123, prefix 0o
0b1111011           // binary number representing 123, prefix 0b
```

Define a grammar in Backus-Naur form (BNF) for the non-terminal start symbol `<number>`. Your grammar has to cover hexadecimal, decimal, octal, and binary numbers and it should not accept invalid inputs such as `0b42` or `0x` or `hello world`. You may use online tools such as the BNF Playground to test your BNF grammar. Submit your solution as a text file.

Problem 12.2: *pre- and postconditions* (4 points)

Determine the weakest precondition (respectively strongest postcondition) for the following algorithms. Explain how you arrived at your result.

a) What is the strongest postcondition of algorithm 1?

Algorithm 1

Precondition: $X > 8$

1: $Z := X \cdot 2$
2: $Y := Z + 2$
3: $X := Y \cdot 3$

Postcondition: ...

b) What is the weakest precondition of algorithm 2?

Algorithm 2

Precondition: ...

1: $X := X + 2$
2: $Y := X \cdot 4$
3: $X := Y - 4$

Postcondition: $X > 10 \wedge Y < 20$

c) Determine the weakest precondition for algorithm 3:

Algorithm 3

Precondition: ...

1: $X := 3 \cdot Y - 2$
2: **if** $X < 12$ **then**
3: $\quad Y := 3 \cdot X - 9$
4: **else**
5: $\quad Y := X + 6$
6: **fi**
7: $Y := Y - 2$

Postcondition: $7 \leq Y < 25$

Problem 12.3: *levenshtein edit distance (haskell)* (2+1 = 3 points)

The Levenshtein distance or edit distance between two strings quantifies how dissimilar two strings are by counting the minimum number of operations required to transform one string into the other. A simple algorithm to calculate the edit distance is given here:

> The edit distance between an empty string and a non-empty string is the length of the non-empty string. Given two non-empty strings, the edit distance is (i) the edit distance of the two strings without the first character if the first characters of the two strings are the same or (ii) one plus the minimum edit distance obtained by (1) adding the first character of the second string to the first string, (2) removing the first character of the first string, or (3) replacing the first character of the first string with the first character of the second string.

This is a naive algorithm to calculate the edit distance of two strings with relatively poor performance for large strings. But it is good enough for us right now. Since strings are lists in Haskell, we can generalize this algorithm to work with arbitrary lists where the list elements support equality comparisons.

a) Write a Haskell module `EditDistance` defining a polymorphic function `ed` that takes two lists (of a type supporting equality) and returns the edit distance of the two lists.

b) Write a Haskell unit test module `EditDistanceTest` that imports the `ed` function from the module `EditDistance` and tests it with a suitable collection of test cases.

Ideally, you write the test cases before you write the definition of the `ed` function. Submit your Haskell files as text files.