Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CH-232-A
Date: 2019-11-22
Due: 2019-11-29

ICS 2019 Problem Sheet #11

Problem 11.1: *fork system call* (1+3 = 4 points)

Consider the following C (C++) program (let me call the source file `foo.c`).

```
#include <unistd.h>

int main(int argc, char *argv[])
{
    for (; argc > 1; argc--) {
        if (0 == fork()) {
            (void) fork();
        }
    }
    return 0;
}
```

a) Assume the program has been compiled into `foo` and that all system calls succeed at runtime. How many child processes are created for the following invocations of the program? Explain

- `./foo`
- `./foo a`
- `./foo a b`
- `./foo a b c`
- `./foo a b c d`

b) A process returns a number to its parent when it exits and a process stays around until this has happened. If the parent process is not interested in picking up numbers from its child processes, then the child processes stay around as "zombies". Write a C program that creates for every command line argument one zombie process. Then use utilities like `top` or `ps` to find the zombies in the process list and document that you managed to create zombies.

Problem 11.2: *BNF grammar* (2+1 = 3 points)

You are given the following context free grammar in BNF format.

```
<expression> ::= <term> | <expression> "+" <term>
<term>       ::= <factor> | <term> "*" <factor>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
<digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

a) Show how the expression `4 * (2 * x + 7)` can be reduced to the start symbol of the grammar. Apply only one rule in each step and try in every step to replace the leftmost symbol you can replace.

b) Is any additional information needed to reduce this expression in a meaningful way? Explain.

Problem 11.3: *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. An 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 to arbitrary lists where the list elements support the equality operation.

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

b) Write a Haskell unit test module that imports your `ed` function 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.