Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CH-232-A
Date: 2020-11-20
Due: 2020-11-27

ICS 2020 Problem Sheet #11

Problem 11.1: *fork system call* (2+3 = 5 points)

Consider the following C program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void action(int m, int n)
{
    printf("(%d,%d)\n", m, n);
    if (n > 0) {
        if (fork() == 0) {
            action(m, n-1);
            exit(0);
        }
    }
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        int a = atoi(argv[i]);
        action(a, a);
    }
    return 0;
}
```

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

   (1) `./cnt`
   (2) `./cnt 1`
   (3) `./cnt 2`
   (4) `./cnt 1 2 3`

b) Remove the line `exit(0)` and compile the program again. What is printed to the terminal and How many child processes are created for the following invocations of the program? Explain how you arrived at your answer.

   (1) `./cnt 1`
   (2) `./cnt 2`
   (3) `./cnt 1 2`
   (4) `./cnt 1 2 3`

Problem 11.2: *stack frames and tail recursion* (1+1 = 2 points)

As discussed in class, function calls require to allocate a stack frame on the call stack. A simple recursive function with a recursion depth n requires the allocation of n stack frames, i.e., the memory complexity grows linear with the recursion depths. In order to improve performance, compilers of high-level programming languages try to optimize the execution of recursive functions.

If a function does a function call as the last action of the function, then this function call can reuse the current stack frame. A recursive function that has this behaviour is called tail recursive. (See also Tail Recursion Explained - Computerphile on YouTube.)

Below is a definition of the function `pow :: Integer -> Integer -> Integer` calculating the function $pow(x, n) = x^n$.

```haskell
pow :: Integer -> Integer -> Integer
pow x n
    | n == 0    = 1
    | n == 1    = x
    | otherwise = x * pow x (n-1)
```

a) The function `pow` has a linear time complexity. Define a recursive function `pow'`, which has a logarithmic time complexity. You can utilize the following law:

$$pow(x, n) = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{if } n \text{ is even} \\ x \cdot x^{n-1} & \text{otherwise} \end{cases}$$

b) Define a tail recursive function `pow''` with a logarithmic time complexity.

Problem 11.3: *evaluation of arithmetic expressions* (1+2 = 3 points)

Arithmetic expressions are often represented as a tree structure within a compiler. We use an infix notation:

```haskell
data Ops   = Plus | Minus | Mult
data Exp a = Val a | Op (Exp a) Ops (Exp a)
```

Below is an example of an expression using this above data type definitions.

```haskell
expr = Op (Val 5) Plus (Op (Op (Val 3) Minus (Val 11)) Mult (Val 2))
```

a) Implement a function `display :: (Show a) => Exp a -> String` that returns a proper string representation. For example, `display expr` returns `(5 + ((3 - 11) * 2))`.

b) Modify the data type definitions so that a division can be represented in an arithmetic expression. Implement a function `eval :: (Integral a) => Exp a -> Maybe a` that returns just a value or nothing if a division by zero occurs.