Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Module: CH-232
Date: 2022-11-18
Due: 2022-11-25

ICS 2022 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+2 = 3 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 `powLin :: Integer -> Integer -> Integer` calculating the function $powLin(x, n) = x^n$.

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

a) The function `powLin` has a linear time complexity. Define a recursive function `powLog`, 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 `powTail` with a logarithmic time complexity.

Below is a template for your solution providing some test cases.

```
module Main (main) where

import Test.HUnit

powLin :: Integer -> Integer -> Integer
powLin x n
    | n == 0 = 1
    | otherwise = x * powLin x (n-1)

powLog :: Integer -> Integer -> Integer
powLog x n = undefined

powTail :: Integer -> Integer -> Integer
powTail x n = undefined

powLinTests = TestList [ map (powLin 0) [0,1,2,3,10] ~?= [1,0,0,0,0]
                       , map (powLin 2) [0,1,2,3,10] ~?= [1,2,4,8,1024]
                       , map (powLin 5) [0,1,2,3,10] ~?= [1,5,25,125,9765625]
                       ]

powLogTests = TestList [ map (powLog 0) [0,1,2,3,10] ~?= [1,0,0,0,0]
                       , map (powLog 2) [0,1,2,3,10] ~?= [1,2,4,8,1024]
                       , map (powLog 5) [0,1,2,3,10] ~?= [1,5,25,125,9765625]
                       ]

powTailTests = TestList [ map (powLog 0) [0,1,2,3,10] ~?= [1,0,0,0,0]
                        , map (powLog 2) [0,1,2,3,10] ~?= [1,2,4,8,1024]
                        , map (powLog 5) [0,1,2,3,10] ~?= [1,5,25,125,9765625]
                        ]

main = runTestTT $ TestList [powLinTests, powLogTests, powTailTests]
```

Students who prefer to write imperative code in C can solve this problem using the following C template.

```
#include <assert.h>

```

```
static int pow_lin(int x, int n)
{
    if (n == 0) {
        return 1;
    }
    return x * pow_lin(x, n-1);
}

static int pow_log(int x, int n)
{
    return -1;
}

static int pow_tail(int x, int n)
{
    return -1;
}

int main(void)
{
    int ns[] = { 0, 1,  2,   3,      10 };
    int t0[] = { 1, 0,  0,   0,       0 };
    int t2[] = { 1, 2,  4,   8,    1024 };
    int t5[] = { 1, 5, 25, 125, 9765625 };

    for (int i = 0; i < sizeof(ns)/sizeof(ns[0]); i++) {
        assert(pow_lin(0, ns[i])  == t0[i]);
        assert(pow_log(0, ns[i])  == t0[i]);
        assert(pow_tail(0, ns[i]) == t0[i]);
        assert(pow_lin(2, ns[i])  == t2[i]);
        assert(pow_log(2, ns[i])  == t2[i]);
        assert(pow_tail(2, ns[i]) == t2[i]);
        assert(pow_lin(5, ns[i])  == t5[i]);
        assert(pow_log(5, ns[i])  == t5[i]);
        assert(pow_tail(5, ns[i]) == t5[i]);
    }
    return 0;
}
```

Problem 11.3: *integer expression simplification (haskell)* (1+1 = 2 points)

Our goal is to simplify integer expressions that may include constants and variables and which are constructed using a sum and a product operator. For example, we want to write an program that can simplify $(2 \cdot y) \cdot (3 + (2 \cdot 2))$ to $14 \cdot y$. We can represent expressions in Haskell using the following data type:

```
data Exp = C Int        -- a constant integer
         | V String     -- a variable with a name
         | S Exp Exp    -- a sum of two expressions
         | P Exp Exp    -- a product of two expressions
         deriving (Show, Eq)
```

We use the following rules to simplify expressions:

- S1 Adding two constants a and b yields a constant, which has the value $a + b$, e.g., $3 + 5 = 8$.
- S2 Adding 0 to a variable yields the variable, i.e., $0 + x = x$ and $x + 0 = x$.
- S3 Adding a constant a to a sum consisting of a constant b and a variable yields the sum of the $a + b$ and the variable, e.g., $3 + (5 + y) = 8 + y$.

P1 Multiplying two constants a and b yields a constant, which as the value $a \cdot b$, e.g., $3 \cdot 5 = 15$.

P2 Multiplying a variable with 1 yields the variable, i.e., $1 \cdot y = y$ and $y \cdot 1 = y$.

P3 Multiplying a variable with 0 yields the constant 0, i.e., $0 \cdot y = 0$ and $y \cdot 0 = 0$.

P4 Multiplying a constant a with a product consisting of a constant b and a variable yields the product of $a \cdot b$ and the variable, e.g., $3 \cdot (2 \cdot y) = 6 \cdot y$.

The usual associativity rules apply. Note that we have left out distributivity rules.

a) Implement a function `simplify :: Expr -> Expr`, which simplifies expressions that do not contain variables. In other words, `simplify` returns the (constant) value of an expression that does not contain any variables.

b) Extend the function `simplify` to handle variables as described above.

Submit your Haskell code plus an explanation (in Haskell comments) as a plain text file. Below is a template providing a collection of test cases.

```haskell
module Main (main) where

import Test.HUnit

data Exp = C Int        -- a constant integer
         | V String     -- a variable with a name
         | S Exp Exp    -- a sum of two expressions
         | P Exp Exp    -- a product of two expressions
         deriving (Show, Eq)

simplify :: Exp -> Exp
simplify _ = undefined

tI0 = TestList
  [ simplify (C 3) ~?= C 3                        -- 3 = 3
  , simplify (V "y") ~?= V "y"                    -- y = y
  ]

tS1 = TestList
  [ simplify (S (C 3) (C 5)) ~?= C 8              -- 3 + 5 = 8
  ]

tS2 = TestList
  [ simplify (S (C 0) (V "y")) ~?= V "y"          -- 0 + y = y
  , simplify (S (V "y") (C 0)) ~?= V "y"          -- y + 0 = y
  ]

tS3 = TestList
  [ simplify (S (S (C 3) (V "y")) (C 5)) ~?= S (C 8) (V "y")     -- (3 + y) + 5) = 8 + y
  , simplify (S (S (V "y") (C 3) ) (C 5)) ~?= S (C 8) (V "y")    -- (y + 3) + 5) = 8 + y
  , simplify (S (C 3) (S (C 5) (V "y"))) ~?= S (C 8) (V "y")     -- 3 + (5 + y) = 8 + y
  , simplify (S (C 3) (S (V "y") (C 5))) ~?= S (C 8) (V "y")     -- 3 + (y + 5) = 8 + y
  ]

tS4 = TestList
  [ simplify (S (S (C 3) (C 5)) (C 8)) ~?= C 16                  -- (3 + 5) + 8 = 16
  , simplify (S (C 3) (S (C 5) (C 8))) ~?= C 16                  -- 3 + (5 + 8) = 16
  , simplify (S (C 5) (V "y")) ~?= S (C 5) (V "y")               -- 5 + y = 5 + y
  , simplify (S (V "y") (C 5)) ~?= S (V "y") (C 5)               -- y + 5 = y + 5
  , simplify (S (V "x") (V "y")) ~?= S (V "x") (V "y")           -- x + y = x + y
  ]
```

```haskell

tP1 = TestList
  [ simplify (P (C 3) (C 5)) ~?= C 15            -- 3 * 5 = 15
  ]

tP2 = TestList
  [ simplify (P (C 1) (V "y")) ~?= V "y"         -- 1 * y = y
  , simplify (P (V "y") (C 1)) ~?= V "y"         -- y * 1 = y
  ]

tP3 = TestList
  [ simplify (P (C 0) (V "y")) ~?= C 0           -- 0 * y = 0
  , simplify (P (V "y") (C 0)) ~?= C 0           -- y * 0 = 0
  ]

tP4 = TestList
  [ simplify (P (P (C 3) (V "y")) (C 2)) ~?= P (C 6) (V "y")    -- (3 * y) * 2) = 6 * y
  , simplify (P (P (V "y") (C 3) ) (C 2)) ~?= P (C 6) (V "y")   -- (y * 3) * 2) = 6 * y
  , simplify (P (C 3) (P (C 2) (V "y"))) ~?= P (C 6) (V "y")    -- 3 * (2 * y) = 6 * y
  , simplify (P (C 3) (P (V "y") (C 2))) ~?= P (C 6) (V "y")    -- 3 * (y * 2) = 6 * y
  ]

tP5 = TestList
  [ simplify (P (P (C 3) (C 5)) (C 8)) ~?= C 120                -- (3 * 5) * 8 = 120
  , simplify (P (C 3) (P (C 5) (C 8))) ~?= C 120                -- 3 * (5 * 8) = 120
  , simplify (P (C 5) (V "y")) ~?= P (C 5) (V "y")              -- 5 * y = 5 * y
  , simplify (P (V "y") (C 5)) ~?= P (V "y") (C 5)              -- y * 5 = y * 5
  , simplify (P (V "x") (V "y")) ~?= P (V "x") (V "y")          -- x * y = x * y
  ]

tM0 = TestList [
  -- (2 * y) * (3 + (2 * 2)) = 14 * y
  simplify (P (P (C 2) (V "y")) (S (C 3) (P (C 2) (C 2)))) ~?= P (C 14) (V "y")
  -- x + (1 + -1) = x
  , simplify (S (V "x") (S (C 1) (C (-1)))) ~?= V "x"
  -- (1 + -1) * x = 0
  , simplify (P (S (C 1) (C (-1))) (V "x")) ~?= C 0
  -- (2 + -1) * x = x
  , simplify (P (S (C 2) (C (-1))) (V "x")) ~?= V "x"
  -- (2 * 2) * (3 + 4) = 28
  , simplify (P (P (C 2) (C 2)) (S (C 3) (C 4))) ~?= C 28
  ]

main = runTestTT $ TestList [tI0, tS1, tS2, tS3, tS4, tP1, tP2, tP3, tP4, tP5, tM0 ]
```