IPS - Assignment 1

Christian Bendix Fjordstrøm(wjx230)

27. april 2023

Task 1

I am a DIKU Computer Science BSc general profile student. I have a decent proficiency with functional programming; I remember the curriculum from PoP, and I have read through (most of) Learn You a Haskell For Great Good. I am familiar with assembly programming using the RISC-V architecture from CompSys earlier this year. In general I find functional programming interesting, and specifically I find parsing using functional languages interesting.

Task 2

From the admittedly very limited testing done in `calculator.fsx`, all the desired functionality is there and the evaluator is complete and working correctly. The tests can be seen below.

```
(* Tests *)
let eval_test_constant = (eval [] (CONSTANT (INT 4)) = INT 4)
let eval_test_lookup = (eval [("x", INT 4)] (VARIABLE "x") = INT 4)

let program0 = "1 - 2 - 3"
let program1 = "let x = 4 in x + 3"
let program2 = ("let x0 = 2 in \
                 let x1 = x0 * x0 in \
                 let x2 = x1 * x1 in x2 * x2")
let program3 = "sum x = 1 to 4 of x*x"
let program4 = "max x = 0 to 10 of 5 * x - x * x"
let program5 = "argmax x = 0 to 10 of 5 * x - x * x"
let program6 = "prod x = 1 to 5 of x"

let eval_test_arithmetic = (run program0 = INT -4)
let eval_test_let = (run program1 = INT 7)
let eval_test_nested_let = (run program2 = INT 256)
let eval_test_sum = (run program3 = INT 30)
let eval_test_max = (run program4 = INT 6)
let eval_test_argmax = (run program5 = INT 2)
let eval_test_prod = (run program6 = INT 120)

printfn "test results:"
printfn " eval_test_constant = %b" eval_test_constant
printfn " eval_test_lookup = %b" eval_test_lookup
printfn " eval_test_arithmetic = %b" eval_test_arithmetic
printfn " eval_test_let = %b" eval_test_let
printfn " eval_test_nested_let = %b" eval_test_nested_let
printfn " eval_test_sum = %b" eval_test_sum
printfn " eval_test_max = %b" eval_test_max
printfn " eval_test_argmax = %b" eval_test_argmax
printfn " eval_test_prod = %b" eval_test_prod
```

As can be seen in the following code snippet showing the case for **rmax**, there are no extra evaluations done, each expression is evaluated once, and when the end of the recursion is reached, it returns. Therefore, the time and space usage should reasonably match what is expected from an evaluator of mathematical expressions written in F#.

```
| RMAX ->
  if n1 > n2 then
    failwith "upper bound must be greater or equal to lower bound"
  else
    let rec max x currentMax =
      if x > n2 then currentMax
      else
        let vtab1 = bind var x vtab
        let newMax =
```

```
10          match ((eval vtab1 e3), currentMax) with
11            | (INT r, INT cMax) -> if r > cMax then INT r else INT cMax
12        max (match x with INT x -> INT (x+1)) newMax
13    max n1 (INT -2147483648)
```

Code sharing was attempted as much as possible. For example, every operation in **OVER** needs to have **e1** and **e2** evaluated so they can check if the bounds are correct. Therefore, the two expressions are evaluated and saved before matching the operators. **RSUM** and **RPROD** have also been combined in an attempt to share code, but rather than making it easier to maintain, this probably just makes it harder to understand what is going on when looking back at it. This could also be done with **RMAX** and **ARGMAX** where it could be checked at the end which one of the operators it is, and based on that return either the maximum value found or the index of the maximum value found.

In **RMAX** and **ARGMAX**, a current max is checked against in each recursion. This current max is initially set to the minimum value of a 32 bit signed integer, there could probably be a cleaner way of implementing this.

Task 3

The idea behind **mul** is to check if one of the parameters is 0, and if it is return 0, otherwise add the other paramater to a call to mul and decrement the parameter that is not added. To deal with potential negative numbers, if the parameter that changes is negative, the number that gets added needs to be negated, and the parameter that changes needs to be incremented instead.

```
1  fun int mul(int x, int y) =
2    if y == 0 then 0
3    else
4      if 0 < y then x + mul(x, y-1)
5      else 0 - x + mul(x, y+1)
```

When computing **difs**, I wanted to map over **arr**, but I could not see a way of accessing consecutive elements, therefore the array containing the values from the call to **iota(n)** is mapped over. Mapping over **iota_arr** gives the different indices in **arr** that can then be used to index into **arr**. **mul** is then used to compute the array containing the squares of **difs** and a separate **plus** function is defined to give to reduce to sum the array of squares.

```
1  fun int plus(int x, int y) =
2    x + y
3
4  fun int main() =
5    let n = read(int) in
6    if n < 1 then
7      let t = write("Incorrect Input!") in
8      0
9    else
10     let iota_arr = iota(n) in
11     let arr = map(fn int (int i) => read(int), iota_arr) in
12     let difs = map(fn int (int i) =>
13                                  if i == 0 then arr[0]
14                                  else arr[i] - arr[i-1]
15                                  , iota_arr) in
16     let squares = map(fn int (int i) => mul(i, i), difs) in
17     let result = reduce(plus, 0, squares) in
18     write(result)
```

Running the program with the interpreter, compiler and `runtests.sh` all produces the expected result.