IPS, Assignment 1

Helga Rykov Ibsen <mcv462> Hold 1

2. maj 2023

Task 1

1. I am a DIKU Computer Science BSc student (general profile).

2. My self-perceived level of proficiency in functional programming with F# is intermediate. I do remember the course PoP, but could use some more update than what we've got now.

3. I am familiar with assembly programming from CompSys, but do need some brushing up. We had to use RISC-V architecture in CompSys.

4. My expectation to the course is to get a better understanding of how the compiler works, brush up functional programming with F# and get an idea of the grammar behind high-level programming languages.

Task 2

1

As far as the completness of the program is concerned, `rec eval()` is implemented completely (cf. <calculator.fsx>) and includes the following cases: `CONSTANT, VARIABLE, OPERATE, LET_IN and OVER`. The operator `op` comprises three arithmetical operations: addition, subtraction and multiplication. The operator `rop` involves the range operations of summing up the values in an interval `RSUM`, multiplying up the values in the interval `RPROD`, finding the maximum value in the interval for a specific expression `RMAX` and finding the maximum value and returning its index `RARGMAX`.

However, at the moment the program does not handle negative integers as input: it throws the `"parse error"` exception for the inputs like "1 * -30". We are not expected to optimize the parser, but a quick fix to this is to write zero before the minus and put the expression into the paranthesis: as "2 * (0 - 30)". If the negative term is not put into the paranthesis as in "2 * 0 - 30", the program will return an incorrect result -30, because multiplication operation has a higher priority than subtraction: 2 * 0 = 0 in 0 - 30 = -30.

2

Speaking about the correctlness of the functionality, the program works correctly. I have tested it on the handed-in program0 - program5 and a couple of other inputs:

Input	Output
0. 1 - 2 - 3	INT -4
1. let x = 4 in x + 3	INT 7
2. let x0 = 2 in let x1 = x0 * x0 in let x2 = x1 * x1 in x2 * x2	INT 256
3. sum x = 1 to 4 of x*x	INT 30
4. max x = 0 to 10 of 5 * x - x * x	INT 6
5. argmax x = 0 to 10 of 5 * x - x * x	INT 2
2 * (0 - 30)	INT -60
30 * 56 *(0-2)	INT -3360
let x0 = 10 in let x1 = x0 + x0 in let x2 = x1 * x1 in x2 * x2	INT 160000
let x0 = 10 in let x1 = x0 + x0 in let x2 = x1 * x1 in x2 - x0	INT 390

3

I evaluate my `eval()` to have linear worst-case running time, O(n). I analyze the functions in accord with each branch/case and choose the one with the worst time complexity:

- CASE 1: CONSTANT n -> O(1): it takes constant time to evaluate ans return a constant.
- CASE 2: VARIABLE v -> O(n): it takes linear time for `eval()` to evaluate a variable v because it calls `lookup()` that runs in O(n), where n is a number of entries in a symbol table.
- CASE 3: OPERATE -> O(1): it takes constant time for each of the arithmetic operations BPLUS, BMINUS and BTIMES.
- CASE 4: LET_IN -> O(n). In principle, this branch should run in O(1), because `bind()` runs in constant time. But if `e1` is evaluated to a variable, then the worst-case for this branch would be linear.

- CASE 5: OVER -> O(n). The case comprises four subcases RSUM, RPROD, RMAX and RARGMAX. All of them include a recursive `range()` function that will determine the overall complexity of CASE 5. The recursive funs worst-case running time corresponds to the number of elements in the array or n = e2 = value2 = endValue. In other words, the total worst-case running time for CASE 5 is O(n).

Summing up, the overall time complexity of `eval()` is O(n). The space `eval()` uses corresponds to the length of the array it takes as input and is maximum as large as the number of elements in the array n, i.e. O(n).

4

At the moment `eval()` contains repeated code as OVER includes a recursive renage function for each subcase: RSUM, RPROD, etc. to to perform the iteration and accumulation within each of the subcases.

To improve the program so that it does not contain repeated code lines, one should add helper-functions for each of the subcases and define them outside of `eval()`. That would be in accord with good coding principles, namely that about avoiding redundancy and repetition. And that would make `eval()` considerably shorter and more elegant, with the four subcases of OVER calling a respective recursive range function defined below.

Task 3

Non-obvious implementation choices in the Fasto program:

1. The implementation of `mul()`: one of the ways to handle multiplication with non-positive integers is to increment the second parameter and to substract the first parameter (cf. assign1.fo & mul.fo)
   `if y < 0 then mul(x, y+1) - x .`
   Another way of handling negative integers is to flip the sign of both terms: `if y < 0 then mul(-x, -y)`

2. The implementation of `map()`: in order to implement the diff operation on the elements in a list, we need to define the anonymous fun that has a base case for the first element `if i == 0 then arr[0]` and the

rest: calculates the difference between two neighbouring elements `else arr[i] - arr[i-1]`.

3. The implementation of `potensPlus`: `reduce()` returns a scalar, which is what the program should do. But it is defined as a plus operation `reduce(plus, 0, 1,..., n-1)`. In our case, the program should compute the accumulated value of each element in array raised to the power of 2.
   We define an additional function `potensPlus()` outside of `main`, which we pass in as the argument to `reduce()`.

   First, I defined `fun int potensPlus(int a, int b) = mul(a,b) + mul(b, b)` and got a very large number as output, because `reduce()` would add the product of two elements `mul(a,b)` to the next element in the array per each element. To fix this, we need to redefine `potensPlus` so that only the second element is raised to the power of 2 as in `fun int potensPlus(int a, int b) = a + mul(b, b)`. That fixes the bug and returns the expected output.

Tests of `mul()`: The `mul()` has been tests on various input: two positive ints, positive & negative int, negative & positive int, two negative ints, a zero & an int, a zero & a zero. See `<mul.fo>, <mul.in>, <mul.out>`. The program returns as expected.

`map()` is defined within `main` and would require more effort to test multiple inputs that `mul()` does. However, it returns as expected for the input defined in the assignment {4, 3, 7, 2, 4} and returns 54. See the files `<assign1.fo>, <assign1.in>, <assign1.out>`.