Task 1

1. I am a DIKU Computer Science BSc, general profile
2. I have little to no proficiency with F#
3. I have mininal knowledge of assembly programming
4. I'm very interested in the topic of compilers

Task 2

a) Completeness and Correctness

All the requested features have been implemented fully. All 10 possible expressions are accounted for, which can be seen by testing each individually, or together.

The program has shown to give the correct answers in all test cases.

Both completeness and correctness can be shown in these tests, all of which return correct results:

```
    Welcome to the calculator! Type "exit" to stop.
    Input an expression : 1+2+3+4-(5+6)
    Evaluation result   : INT -1
    Input an expression : let x=7 in (x*8+9)
    Evaluation result   : INT 65
    Input an expression : sum x=1 to 10 of (prod y=x to 10 of y)
    Evaluation result   : INT 9864100
    Input an expression : max x = 0 to 10 of 5*x-x*x
    Evaluation result   : INT 6
    Input an expression : argmax x = 0 to 10 of 5*x-x*x
    Evaluation result   : INT 2
```

b) Effeciency

The program runs efficiently and with no obvious uses of excessive memory.

Any singular expression, apart from the `OVER` expressions runs in constant time, ignoring any expressions it contains. This means that any expression that contains n non-`OVER` expressions runs in linear time, $O(n)$.

The `OVER` expressions contain a loop, meaning they run in linear time by themselves.

c) Code Sharing/Elegance

The code is made readable and with mininal code duplication. The `OPERATE` and `OVER` expressions have code run before the specific expression is determined, which reduced duplication.

The full `eval` code

```
let rec eval (vtab : SymTab) (e : EXP) : VALUE =
  match e with
    | CONSTANT n -> n
    | VARIABLE v ->
      lookup v vtab
    | OPERATE (op, e1, e2) ->
      let (INT exp1) = eval vtab e1
      let (INT exp2) = eval vtab e2
      match op with
        | BPLUS -> INT (exp1 + exp2)
        | BMINUS -> INT (exp1 - exp2)
        | BTIMES -> INT (exp1 * exp2)
```

```
    | LET_IN (var, e1, e2) ->
      let exp1 = eval vtab e1
      let vtab = bind var exp1 vtab
      eval vtab e2
    | OVER (rop, var, e1, e2, e3) ->
      let (INT exp1) = eval vtab e1
      let (INT exp2) = eval vtab e2
      let results = [
        for i in exp1..exp2 ->
          let vtab = bind var (INT i) vtab
          eval vtab e3
      ]
      match rop with
        | RSUM -> List.fold (fun (INT acc) (INT elem) -> INT (acc + elem)) (INT 0) results
        | RPROD -> List.fold (fun (INT acc) (INT elem) -> INT (acc * elem)) (INT 1) results
        | RMAX -> List.fold (fun (INT acc) (INT elem) -> INT (max acc elem)) results[0] results
        | RARGMAX ->
          let max_elem = List.fold (fun (INT acc) (INT elem) -> INT (max acc elem)) results[0] results
          INT ((List.findIndex ((=) max_elem) results) + exp1)
```

Task 3

a) The `mul` function

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

The `mul` functions is recursive. The base level, when `y` is 0, 0 is returned. In all other cases, `y` is moved 1 step closer to 0, and `x` is either added or subtracked from the final result.

b) The `dif` array

```
let dif = map(fn int (int x) => if x == 0 then arr[x] else arr[x] - arr[x-1], iota(n)) in
```

The `dif` array is constructed using an anonymous function in a `map` function call.

The full Fasto code

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

fun int readInt(int i) = read(int)

fun int squareSum(int x, int y) = x + mul(y, y)

fun int main() =
  let n = read(int) in
  if n == 0 then let a = write("Incorrect Input!") in 0
  else if n < 0 then let a = write("Incorrect Input!") in 0
  else
    let arr = map(readInt, iota(n)) in
    let dif = map(fn int (int x) => if x == 0 then arr[x] else arr[x] - arr[x-1], iota(n)) in
    write(reduce(squareSum, 0, dif))
```