T-501-FMAL Programming languages, Practice class 6
Spring 2021

The goal of this class is to implement recursive first-order functions with multiple arguments, for example:

```
let ack m n =
  if m = 0 then n + 1
  else if n = 0 then ack (m - 1) 1
  else ack (m - 1) (ack m (n - 1)) in
ack 2 2

let fact n =
  let factAcc acc n =
    if n < 1 then acc else factAcc (n * acc) (n - 1) in
  factAcc 1 n in
fact 6
```

1. Proceed from the file FirstFunLexerParser.fs. This contains a lexer, parser and evaluator for a language supporting recursive first-order functions with a single argument.

   Modify the discriminated union type `expr` to handle functions with multiple arguments rather than just one.

   The modified type `expr` should have cases for the these two expression forms:

   ```
   f e1 ... en
   let f x1 ... xn = e in e'
   ```

   where `f` can only be a variable and `n` is at least 1.

2. Extend the parser to support multiple arguments. To this end, adjust the grammar for concrete syntax. Accordingly modify the parsing of function call in `parseFactor`, and the parsing of let in `parseHead`.

3. Extend the evaluator (the function `eval`) to support multiple arguments. Do this both for the static scope and dynamic scope rules. Function definition in let is meant as recursive. You will need to adjust the type `value`.

4. What does the following expression evaluate to under the static and dynamic scope rules? (You can find out by using your parser and evaluator.) Why?

   ```
   let z = 0 in
   let f x y =
     if x = 0 then y + z else let z = z + 1 in f (x - 1) y in
   f 4 5
   ```

5. To solve this problem, proceed from the file FirstFunTypes.fs containing a type inferrer and evaluator for a slightly modified language where function definitions come with type annotations.

   Extend type inference to functions with multiple arguments. You will need to adjust the types `expr` and `typ`.

   In the modified type `expr`, the case of let for functions should correspond to the type-annotated expression form `let f (x1 : t1) ... (xn : tn) : t' = e in e'`.