ISIS-4217 Paradigmas de programación
FP project
Date: October 15, 2025

The objective of this project is to use Mozart to build a very tiny functional programming language (the base to build any functional language). As a matter of fact, the project will develop the initial approach to implement functional programming, which consists of a graph reduction technique called template instantiation.

The idea of template instantiation is to represent expressions (program instructions) as a graph, and apply the outermost reductions,1 to evaluate the expression. The language accepts programs composed of function expressions (*i.e.,* function definitions) and function expressions (*i.e.,* function applications). We will a very reduced language to evaluate the language consisting of built-in primitives (*i.e.,* mathematical operations) and supercombinator redexes consisting of combinations of built-in primitives. The following code snippet shows an example program. For simplicity, we will express a single function definition/call per line of code, and use spaces a separator between names/values/variables.

```
fun twice x = x + x
twice 5
```

The process to reduce a graph expression consists of three steps, which you will have to develop

Task 1. The step 0 in the implementation of template instantiation is to build the graph to represent the program. The graph consists of a tree describing the structure of the program with pointers referencing a same node (to avoid repeated evaluation). The graph is composed of two types of nodes (in its simplest representation):

1. Leaf nodes: representing constants (numbers) or variables
2. @ nodes (Application nodes): representing function applications

Implement the function `GraphGeneration` that takes a program string and returns the program's graph. Put it on the single file `mozart-fp.oz` with your solution.

Task 2. Find the next expression to reduce. The expression to reduce must always be the outermost expression in the tree. 1. Follow the left branch of the application nodes, starting at the root, until you get to a supercombinator or built-in primitive.

Implement the function `NextReduction` that takes a graph, and obtains the reduction to apply. Put it on the single file `mozart-fp.oz` with your solution.

Task 3. Check how many arguments the supercombinator or primitive takes and go back up that number of application nodes; you have now found the root of the outermost function application. Now, reduce the expression (a.k.a evaluate). For built-in primitives you have to evaluate them, for supercombinators replace their definition into the tree

Implement the function `Reduce` that takes a graph, and evaluates the outermost reduction, performing the modifications on the graph. Put it on the single file `mozart-fp.oz` with your solution.

Task 4. Update the expression with the result of the evaluation Note that not all programs need to be reducible (for example if the evaluation is not complete as variables are not known; the reduction of the expression x + x is itself if a value for x is unknown).

Implement the function `Evaluate` that takes a graph, and obtains its value or reducedexpression. Put it on the single file `mozart-fp.oz` with your solution. This function is the evaluation API of the program. Programs should be evaluated as {Evaluate {GenerateGraph P}} to return their resulting value

Example programs:

```
fun square x = x * x
square square 3
%%Res: 81
```

```
fun fourtimes x = var y = x*x in y+y
fourtimes 2
%%Res: 8
```

The process of template instantiation is explained in the Implementing Functional Languages tutorial. Additionally, if it is of any help, there is a mozart program that takes a function definition and generates an equivalent function in prefix notation. A file with example programs an their generated graphs is given as examples

The input to the project is composed of single function expressions (definitions) (*e.g.,* `fun square x = x * x`). Function definitions can have any number of parameters (*e.g.,* `fun sum_n x y z n = (x + y + z)* n`), while function bodies can use any combination of arithmetic expressions (multiplication, sum, subtraction, division) (*e.g.,* `fun arithmetic x y = ((x + y)/ ( x - y))* 2`).

Additionally functions may introduce internal variables (using the `in` construct) (any number of variables) to use within the function (*e.g.,* `fun var_use x = var y x+x in var z = y * 2 in z - 3`

Together with the function definition, programs will have the function application, which can combine (nest) any number of function calls. *e.g.,* :

- `square 3`
- `sum_n 1 sum_n 1 1 1 2 3 2`
- `arithmetic arithmetic 5 6 arithmetic 2 11`
- `var_use var_use 16`

parenthesis may be used if convenient