# Second Task

**Programming Languages**
LVA 185.208
2025 S
TU Wien

## What to do

Develop an interpreter for a functional language as specified below in a dynamically typed language.

## The Language

A program is an expression that gets evaluated when executing the program. The following language elements shall be supported:

- integers
- structured data (like lists or records containing integers and other structured data)
- functions (e.g., lambda abstractions as in the lambda calculus)
- named entities (as a means to specify names for above language elements and use names to adress them)
- a small set of predefined operations, e.g., functions named `plus`, `minus`, `mult`, `div` and `cond` (conditional execution)

It is recommended to use the lambda calculus as well as Lisp or another simple, interpreted language as a guideline, but to implement only the smallest possible subset of it.

Please feel free to design your own language, but keep the syntax and semantics simple. It is recommended not to use expensive compiler technologies (e.g., tools to generate a lexer, parser, AST, etc.) although it is not forbidden to do so.

## An Example Language

Here is the description of a language as an example. If you want, you can implement this language, but your own language is preferred. Figure 1 shows the syntax.

There are tokens (basic syntactic elements) for integer literals, names, and some symbols and parentheses as syntactic elements. Looking at the next token (or character) in a program shall be sufficient to determine the next element syntactically. Most of the syntax shall be self-explanatory. An expression `x . ...` represents a function (lambda abstraction) with `x` as formal parameter and ... as function body, e.g., `x.mult x x` is a function returning the square of `x`. Expressions can be nested as in `x.y.add(mult x x)y` or equivalently `x.(y.add(mult x x)y)`. In this way nested functions, each with one

```
 <expr> ::= <apply>
          | <name> '.' <expr>

<apply> ::= <basic>
          | <apply> <basic>

<basic> ::= <integer>
          | <name>
          | '(' <expr> ')'
          | '{' [<pairs>] '}'
          | '[' [<pairs>] ']'

<pairs> ::= <name> '=' <expr>
          | <pairs> ',' <name> '=' <expr>
```

Figure 1: Syntax in EBNF

parameter, can be used to specify functions with several parameters (Currying). Round parentheses group syntactic elements. A function application is written as a function followed by its argument. For example, `(x.mult x x) 2` applies the above function to `2`, thereby replacing the parameter by `2`, resulting in `mult 2 2` which should be evaluated to 4. By nesting function applications we can provide several parameters to functions, e.g., `(x.y.add(mult x x)y) 2 3` is equivalent to `((x.(y.add(mult x x)y))2)3`, and the evaluation will, step by step, result in

```
(x.y.add (mult x x) y) 2 3  ⟶
(y.add (mult 2 2) y) 3  ⟶
add (mult 2 2) 3  ⟶
add 4 3  ⟶
7
```

Parameters can be of any kind, i.e., integers, functions, and records.

Expressions like `{d=x.mult x x, v=d 2}` represent *lazy records* of named fields, `{}` is the empty record. In this example, `d` is a named function and `v` stands for the result of applying `d` to `2`. Names in records (to the left of `=`) are introduced from left to right. Therefore, `d` in `v=d 2` refers to `x.mult x x`, but `v` cannot be referred to while defining `d`. The content of a lazy record is not executed immediately. A lazy record not applied to some other expression stands just for itself without being evaluated.

Expressions like `[d=x.mult x x, v=d 2]` represent *eager records* of named fields. Such records are essentially the same as lazy records, `[]` represents even the same value as `{}`. However, the content of an eager record is executed as soon and as far as possible. So the variable `v` in the above example of a record has just `4` as value. Executing `[a=x.y.add(mult x x)y, b=a 2, c=b 3]` results in a record

`{a=x.y.add(mult x x)y, b=y.add 4 y, c=7}`. If nothing can be evaluated in a record, it does not matter if braces or square brackets are used.

In general, eager evaluation is used. Hence, expressions are immediately evaluated as far as possible as can be seen in the above example on nested funtion applications. A function not applied to an argument stands just for itself. Executing `x.y.add(mult x x)y` gives just `x.y.add(mult x x)y` as result. A function applied to an argument is immediately reduced as far as possible. For example, `(x.y.add(mult x x)y) 3` will be evaluated to `y.add 9 y`.

The predefined function `cond` is an exception concerning eager evaluation: In `cond b t f`, the argument `t` is evaluated only if and after `b` has been evaluated to a value representing `true`, and `f` only if and after `b` has been evaluated to a value respresenting `false`. Neither `t` nor `f` (nor any parts of these expressions) are executed before the value of `b` has been determined. To keep it simple, we assume that `0` and `{}` represent `false` while each other integer and each non-empty record represents `true`.

Records (no matter if lazy or eager) specify environments of named expressions when applied to arguments (i.e., a record followed by an expression). For example,

```
[a=x.y.add(mult x x)y, b=a 2, c=b 3] minus (b 5) c
```

evaluates to 2 using the definitions in the record:

```
{a=x.y.add(mult x x)y, b=y.add 4 y, c=7}minus(b 5) c ⟶
minus ((y.add 4 y) 5) 7 ⟶
minus (add 4 5) 7 ⟶
minus 9 7 ⟶
2
```

Hence, `{...}e` and `[...]e` have approximately the same meaning as `let ... in e` in usual functional programming languages. However, `{...}` and `[...]` can be used as arguments and results of functions or as named entities in other records, too. Records are building blocks of larger data structures. An expression `{...}e` can also be seen as the execution of `e` in a program `{...}`. Predefined operations are regarded as existing in each environment.

Figure 2 shows a larger example, where the predefined operations `plus x y` and `minus x y` have usual semantics. The execution starts with `sum (range 3 6)`, applying `range` to `3` and `6`, thereby constructing

```
[val=3, nxt=[val=4, nxt=[val=5, nxt=[]]]]
```

as intermediate result. Most of this work is done by recursive applications of `list`, while `range` just provides appropriate functions as arguments of `list`. By applying `sum` to the intermediate result, `reduce` recursively reduces the list to the sum of integers in the list,

```
{   list=c.f.x.cond (c x)
                    [ val=x, nxt=list c f (f x) ]
                    [] ,
    reduce=f.x.l.cond l
                      (f (reduce f x (l nxt)) (l val))
                      x ,
    range=a.b.list (x.minus b x) (x.plus 1 x) a ,
    sum=l.reduce (x.y.plus x y) 0 l
}
sum (range 3 6)
```

Figure 2: A program example

this is 12. As a detail: `l nxt` in `reduce` is the field `nxt` in the record provided as parameter `l` because the application of `l` to `nxt` sets the environment and thereby hides every other meaning of `nxt` that existed before; accordingly for `l val`. Be careful to use the correct scope in the language implementation.