A simple functional programming language

Goal for this part of the exercise is building a simple functional programming language, only implementing a minimal subset while getting in touch with the skillset necessary for creating a new language.

Requirements

- Linux or Unix based operating system (Windows can work but is not supported)
- Python 3.8 or newer

Running programs

Using a virtual environment

It is recommended to create a new virtual environment before running the interpreter for this language. Use `python3 -m venv venv` to create a new venv, and `source venv/bin/activate` to activate it.

Installing dependencies

To install all dependencies required for running the interpreter, use `pip install -r requirements.txt`, preferably with a virtual environment activated.

Running a program

To run a program with the provided interpreter, run:

```
python3 interpreter.py /path/to/code/file
```

The file specified has to be structured according to the Language design section.

Running binary version

If python is not installed on the target system, you can run the bundled binary version of this program in `./dist`. Note: this version is build for Linux

```
./dist/interpreter /path/to/file
```

Language design

The basic building blocks of this langauge are records (multiple key value pairs), integers and functions. Names are strings which can be evaluated to each of the above building blocks. The notation for function calls is strictly prefix.

Example: `add 5 2` is evaluated to `7`

Only one main function can be executed, but this function can call other functions given in a record.

Functions

There are 5 predefined operations:

Operation	Number of arguments	Operation
`add`	2	integer addition
`sub`	2	integer substraction
`mult`	2	integer multiplication
`div`	2	integer division
`cond`	3	conditional statements, returns 2nd argument if 1st is true, 3rd otherwise

New functions can be defined by separating the parameters, and the function body with `->` arrows not unlike lambda notation. Example: `x->mult x x` is a function that represents integer squaring. It has one parameter, `x`, the value of which is then inserted in the expression that represents the function body.

Records

Records are key value pairs, which can be used as data holder, as well as context for a following expression. They can save any citizen of this language as value, but has to use name keys (in essence strings). Entries within a record are separated by a comma `,`.

Example: `{a=x->mult x x} a 4` is a record which defines the variable `a` as a function (which squares the parameter given). The variable `a` is then used to call this function. The variables within a record are only available for the expression immediatly afterwards, but they can also be used within other records.

More examples of this language can be found in the demo folder.

Testing

The tests for this language are automated integration tests, which can be run with pytest.

To run the tests, execute `python -m pytest` in the folder (after following the steps in installing dependencies).

These tests use different code examples (mostly taken from the assignment descriptions) to test the interpreter and check its result against a predefined value.

Test protocol:

#	Test case	Calculated result	Expected result	Test result
1	`add (mult 40 50) 3050`	5050	5050	OK
2	`((x->(y->add(mult x x)y))2)3`	7	7	OK
3	`(x->mult x x) 32`	1024	1024	OK
4	`(x->y->add(mult x x)y) 2 3`	7	7	OK
5	`{a=x->mult x x} a 4`	16	16	OK
6	`{x=5} sub x 1`	4	4	OK
7	`{a=x->y->add(mult x x)y, b=a 2, c=b 3} sub(b 5)c`	2	2	OK
8	`{append = x1->y1->cond x1 {head=x1 head, tail=append(x1 tail)y1} y1,gen = x2->cond x2 (append (gen(sub x2 1)) {head=x2, tail={}}) {}}gen 3`	`{head=1, tail={head=2, tail={head=3, tail={}}}}`	`{head=1, tail={head=2, tail={head=3, tail={}}}}`	OK

Application design

The application is split in 3 parts.

- First, we have the lexer. It tokenizes the input string, so the parsing is easier. It assigns each token a type, and stores it together with the substring of the code the token references.
- Then, we have the parser. The parser parses the tokens, and puts them into a datastructure, which can be evaluated. The datastructure is loosely based on an abstract syntax tree (AST), but also contains the context of the statements and its children.
- Then, we have the datastructures themselves. When calling eval on them, they resolve its dependencies out of the context they got from the parser (and parent nodes of the tree), and evaluate all children until it can evaluate itself.

The lexer is mostly there to make the parsing easier and split two separate parts. Due to that, the parser only has to differentiate between different token types, and does not have to do string operations.

The reason for the AST with integrated context and evaluation functions was initially to reduce complexity, and being able to reuse parsed elements (by copying them out of the context the parser created). In the end, it became clear that this solution was not the easiest at all, since there were quite some problems with the bounded context of language entities passed as arguments to other functions. Those issues were resolved but if the program was to be redesigned, another approach would be chosen.

Note that the code uses python the way it should be, as a dynamically typed language. However, type information is hinted at various locations, to support better tooling like static type analysis, tab completion and so on for development of this interpreter. As much as they helped, they often made “type: ignore” hints necessary to supress warnings.

Benchmarks

Results of benchmarking different scenarios can be seen in the following table. Note that the time given is always for 100 consecutive runs. All measurements are taken with the zsh time command, and only the userspace durations are recorded.

#	Filename	Total time for 100 runs
1	complex_function.fprog	2.16s
2	complex_function_without_parantheses.fprog	2.17s
3	complex_records.fprog	2.15s
4	complex_records_nested.fprog	2.41s
5	simple_expression.fprog	2.15s
6	simple_expression_cond.fprog	2.13s
7	simple_function.fprog	2.15s
8	simple_records.fprog	2.18s

The only signifikant outlier is case #4, which is around 260ms slower than the other cases, due to it being a way more complex case. This leads to the conclusion that the startup of python and initialization of the interpreter takes way longer than the actual evaluation, in our limited test cases.