SEMPL the programing language

Biloshenko Hryhorii (biloshry@fit.cvut.cz)

07.01.2024

1. Introduction

This report is informing about the semestral work done for subject BI-PYT in the winter semester of studying years 2023/2024.

Topic of semester work is self-created programing language and an interpreter for it.

2. Interpretation process

Interpretation is split into 3 parts: lexing, parsing and evaluation.

2.1 Lexing

Lexing or lexical analysis is a process of transforming plain text into tokens. Tokens itself are small, easily categorizable data structures that are then fed to the parser.

2.1.1 Keywords

The defined keywords of SEMPL language are `let`, `if`, `else`, `while`, `fn`, `True`, `False`, `mut`, `return`, `and`, `or`, `not`.

2.2 Parsing

The next step of the interpretation is parsing. Parser turns its input into a data structure representing the input. More precisely, it takes tokens from the lexer and creates an Abstract Syntax Tree (AST for short) representing those tokens.

The parser which was used for SEMPL programing language is a “top-down operator precedence” parser, sometimes called “Pratt parser”. [1] This means that starts with constructing root node of the AST and then descends. It is simple and powerful enough but is not the fastest one.

2.2.1 Statements

The list of statements forms:

- Let statement:
  `"let" [mut] <identifier> <identifier> "=" <expression>`
- Return statement: `"return" [<expression>]`
- Expression statement: `<expression>`
- Block statement: `"{"[<statement>]…"}"`
- If statement:
  `"if" <expression> <block statement> [else <block statement>]`
- While statement: `"while" <expression> <block statement>`
- Function statement:
  `"fn" <identifier> "(" [<identifier>":" <identifier> ["=" <expression>]]…")" "->" <identifier> <block statement>`

2.2.2 Expressions

The list of complex expressions forms:

- Prefix operation: `<prefix operator> <expression>`
- Index operation: `<expression> <infix operator> <expression>`
- Call expression: `<expression>  "("[<expression>]…")"`
- Assignment expression: `<expression> = <expression>`
- Array literal: `"[“[<expression>]…"]"`
- Index operation: `<expression>"["<expression"]"`

Other expressions are identifier, integer literal, float literal, boolean literal and string literal.

2.2.3 Precedence

Operator	Description
x[1], x(1), (1+2)	Index operation, Call operation, Grouped operations
`/, *`	Division, Multiplication
`+, -`	Unary Plus/Minus, Binary Plus/Minus
`==, !=, <, >, <=, >=`	Comparison
`not`	Logical not
`and`	Logical and
`or`	Logical or
`x = 10`	Assignment

2.2 Evaluation

Process of evaluation of the AST happens “on the fly” or in other words without preprocessing. Identifiers are bound to the corresponding scope and can be received from there.

In SEMPL language there are types: `Int, Float, Str, Bool, Arr` and `Singularity`. The last one is used mainly as a return type of void functions, can have only one value – `singularity` and can be aliased as “O”.

Type checking is done at a runtime, so only executed statements and expressions are checked.

The important thing is that the variable of type `Arr` must be always mutable as it is the only mutable type. Also, arrays do not have to contain only one type of objects.

SEMPL language also provides some built-in functions: `print, input, len, append, remove, int` and `float`.

Function parameters default values are evaluated at the function definition time, not at call time. Also, at call time all function arguments are passed as mutable references. That is the reason why `Arr` variables must always be mutable.

3. Testing and error handling

3.1 Testing

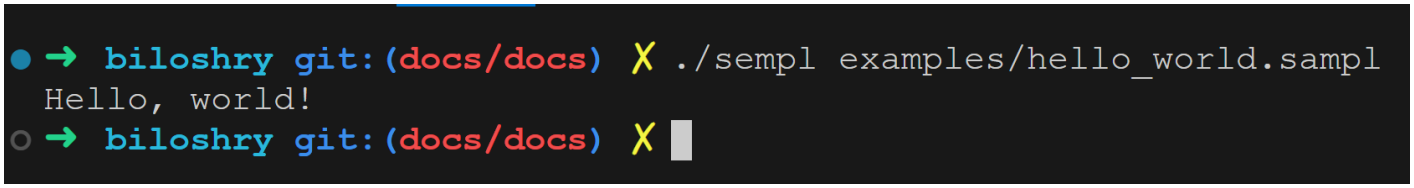
Because of SEMPL project is complex and many possible unexpected situations can happen, the TDD approach was used to ensure less bugs and better architecture definition. Almost two hundred tests were created at the development

process, though they do not cover all possible scenarios and probably twice or trice bigger amount shall be created to cover most of the possible cases.

3.2 Error handling

As for error handling, exceptions were not seen as a good thing to use in such a complex system. Therefor “Result” objects from “result” library were used with custom “Error” object for collecting notes and traceback imitation.

4. Results and possible extensions



The result of my work is a fully functional programing language which can be used either as a REPL (Read-Evaluate-Print-Loop) or with a file containing program code.

In the relation to extensions, there are a lot of place and possibilities for them. Beginning from by constant reference and by value function arguments binding and generic types to support of structs and enums. Also some preprocessing step can be used to faster the code execution.

5. References and links

[1] Writing An Interpreter In Go, Thorsten Ball, 2018. Accessible from https://interpreterbook.com/sample.pdf

[2] Python official documentation, https://docs.python.org/3/

[3] Pytest official documentation, https://docs.pytest.org/en/6.2.x/contents.html

[4] Python result library repository, https://github.com/rustedpy/result

[5] Regex tool, https://regex101.com/