Interpreters and bite sized codes

To execute a list of instructions, a program, called an interpreter, is used. Interpreters are distinguished into categories depending on their execution strategy. Common examples are tree walking interpreters, the other strategy most modern interpreters use are byte code interpreters. To take a look at byte code interpretation with a register based approach, we need to establish a baseline of terms we use a lot in this article: byte code refers to a byte representing an instruction for the byte code virtual machine, such as `0x1` (could represent any instruction). A byte code virtual machine takes this byte, interprets it (could be an argument or could be an instruction) and executes the instruction.

```
125.9*15+3.1415
```

Consider the above arithmetic expression. How would you represent this inherently tree like execution priority into a linear list of bytes? - We will do a deep dive into the interpretation pipeline in the next section, thereby answering this question. Most snippets will be in rust, refer to the rust[1] or the go implementation[2] if you are interested in the complete projects.

Interpretation pipeline To evaluate and therefore interpret the example, our interpreter has to convert the expression into a machine readable format. The next step is parse this stream of tokens into a representation including expression precedence, known as an abstract syntax tree. Afterwards we compile this ast into instructions encoded as bytes.

Tokenisation Refers to the process of converting a stream of characters, such as `125.9*15` into tokens with meaning, taking a look at our example we can at first notate the token kinds we will need and afterwards define a list of tokens representing our example.

```
enum Token {Plus, Asteriks, Number(f64)}
vec![Token::Number(125.9),Token::Asteriks,
Token::Number(15),Token::Plus,
Token::Number(3.14)];
```

Parsing

Compilation

Evaluation

[1] https://github.com/xNaCly/calcrs
[2] https://github.com/xNaCly/calculator