Writting your own interpreter

Felipe Vieira, 2021

is it impossible? / Complexity

We all see interpreters or compilers as this crazy black magic thing created by the gods ages ago. Something impossible to fully understand, with infinite complexity

We will learn that this is not the case

it is a bit tricky tho. not gonna lie.

Why bother? / practicallity

It's pretty probable that nobody ever will use your interpreter. But this isn't the point.

We aren't being practical here. We are being **educational**

List of good stuff that you will achieve by writting a interpreter

- Your knowledge of wtf a interpreter does and how langs work / computer science stuff will grow
- You will practice some pretty damn hard algos
- You will (likely) step out your comfort zone
- You will (hopefully) do some mistakes and you will learn from them
- You will be more critical about shitty langs like **JAVA**
- at the end, you could brag about it

The masterplan

1. Get a really IQ20 (easy) lang to interpret. We will be using LISP
2. Grab some lang that you are comfortable with. I started with Rust and had to go back to JS. Shame
3. Write the interpreter (easy, right?)

LISP

“ LISP has jokingly been described as **“the most intelligent way to misuse a computer”**. I think that description a great compliment because it transmits the full flavour of liberation: it has assisted a number of our most gifted fellow humans in thinking **previously impossible thoughts.**" - E. W. Dijkstra ”

Lisp 101

```
(defun factorial (n)
  (if (< n 1)
      1
      (* n (factorial (- n 1)))
    )
)
```

```
int factorial(int n) {
    if (n < 1)
        return 1;
    else
        return n * factorial( n - 1 );
}
```

Weird right?

LISP is for (LIS)t (P)rocessing

literally every line of code is a list. the full program is just a long list of lists. Also, that means that everything is an expression.

think about the parentesis as just `[]`

```
(+ 1 ( - 2 3)) => ["+", "1", ["-", "2", "3"]]
```

This is good. easy to parse.

Dont forgot this stuff. we will get back to it later

One more thing, Polish notation

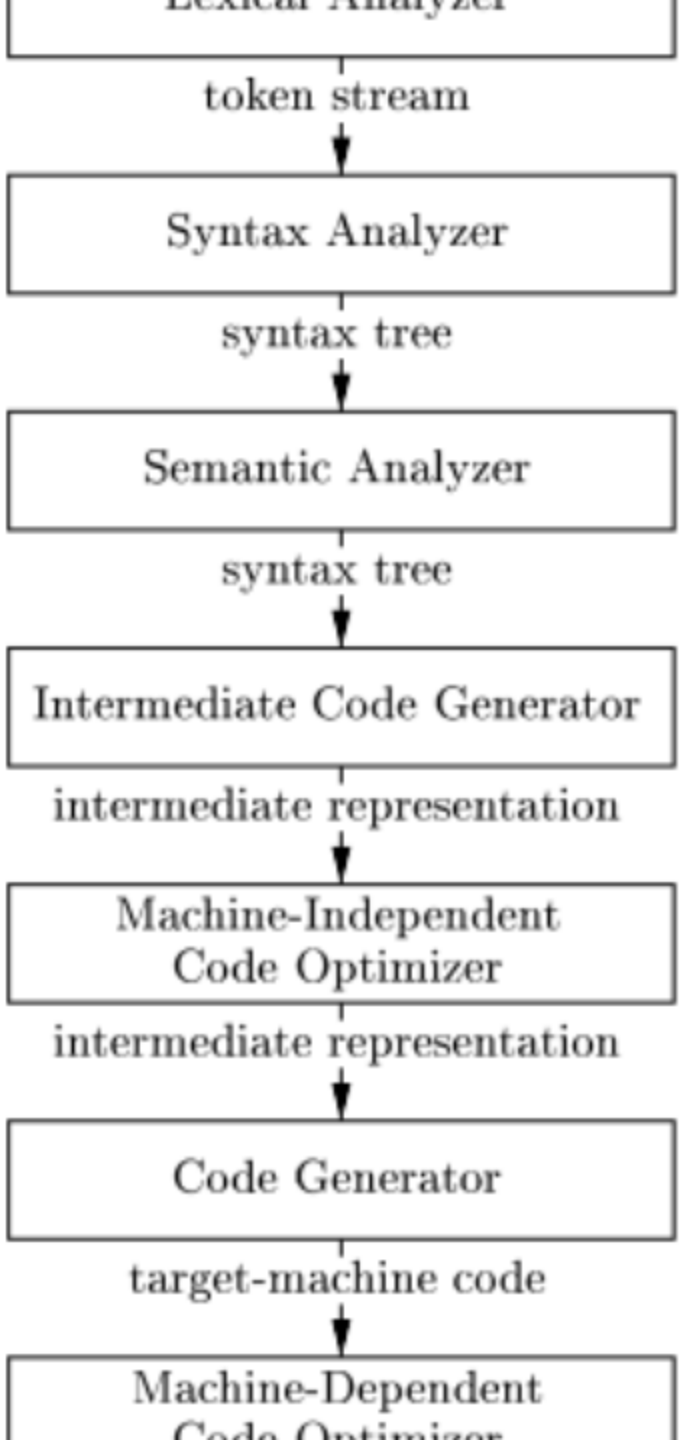
polish notation is just putting the **operator first**. this is weird but if you think about it is pretty smart. We don't have to worry about **operator antecedence**.

Examples

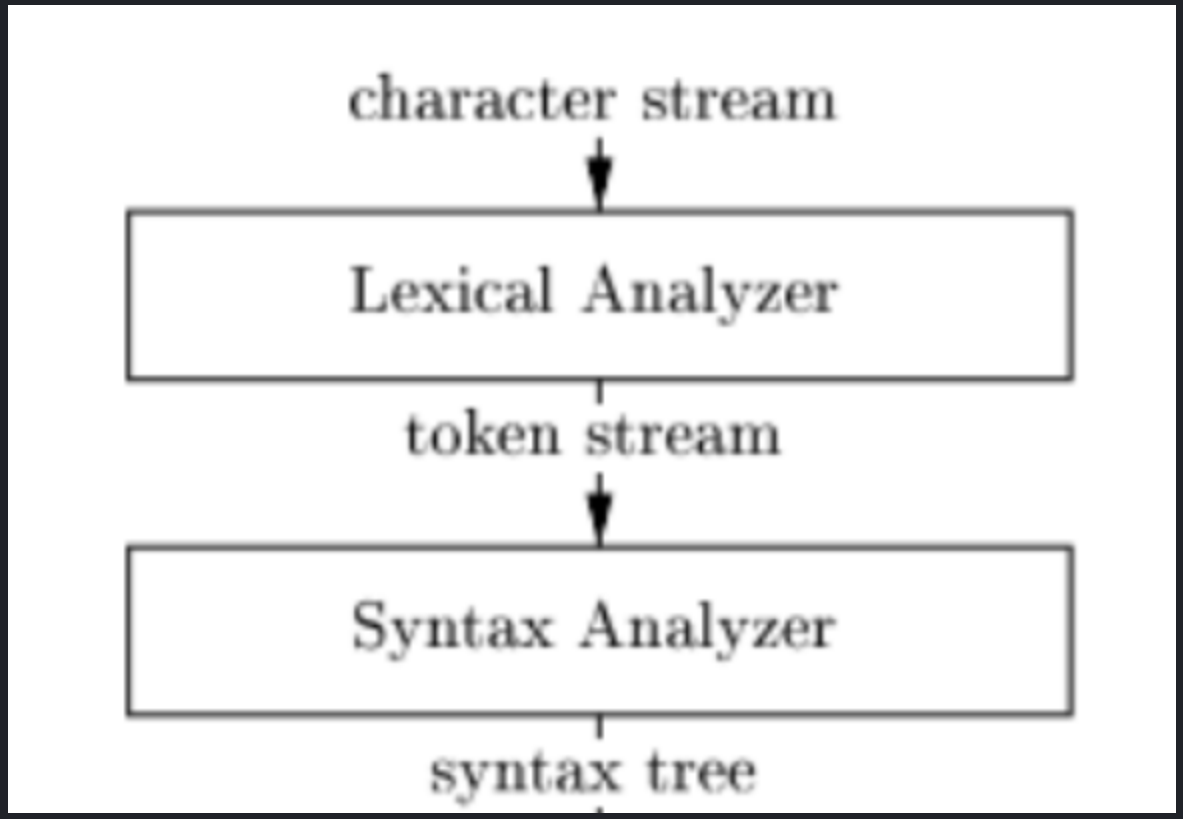
```
+ 1 2 3 => 1 + 2 + 3
+ 1 * 2 3 => 1 + (2 * 3)
```

Interpreters 101

Get ready because this is the real shit 👌


Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent
Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent

Lexical & Syntax Analysis



Lexical Analysis / Tokenizer

our first step here is to split the input program into pieces. What pieces? small pieces. We call this small pieces **tokens**.

It depends a bit on the lang, but normally you want to get symbols like `+=-(){}*&,` and words, like `fn`, `for`, `return`, `factorial` keep them small.

code

```
function tokenizer(text) {
    text = text.split("(").join(' ( ')
    text = text.split(")").join(' ) ')
    return text.split(" ")
}

console.log(tokenizer("(fn (x) (* x x))"))
// [ '(', 'fn', '(', 'x', ')', '(', '*', 'x', 'x', ')', ')' ]
```