Interpretation

Syntactic Analysis

Corey Oliver

Why compilers?

"If you don't understand compilers, you can still write programs - you can even be a competent programmer- but you can't be a master" - Hal Abelson of MIT (co author of SICP)

- A good craftsman should know their tools
- Can be used to build DSL’s
- Techniques learned are often useful in other domains
- Great article by Steve Yegge

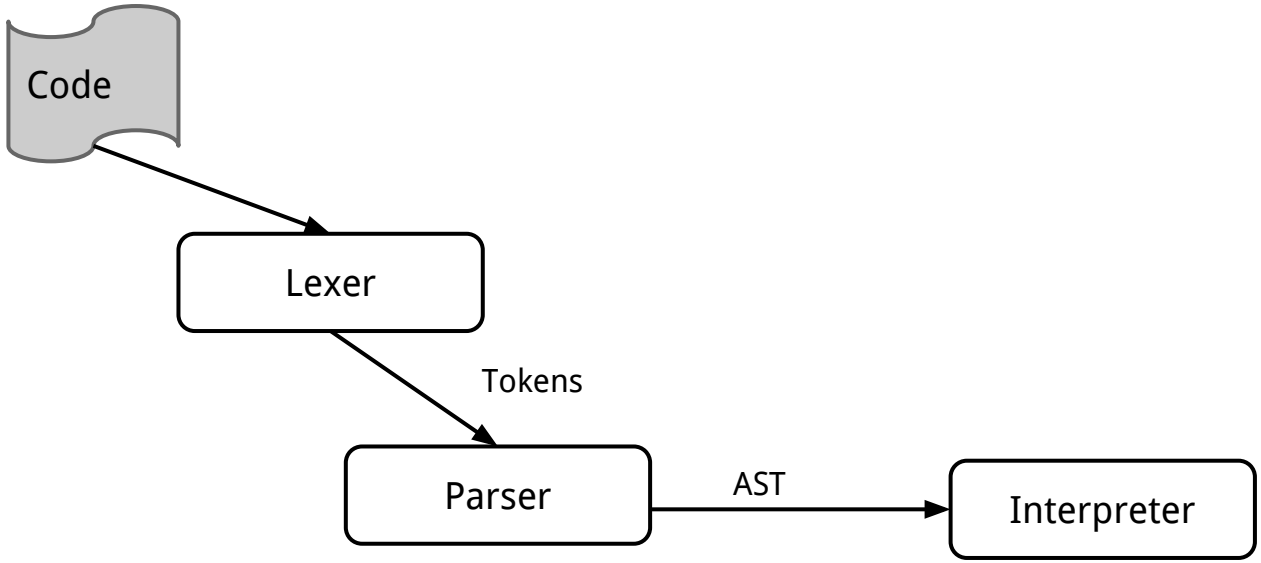
What you did

- Lexical Analysis (Python -> Tokens)

What you will learn today

- Syntactic Analysis (Tokens -> AST)
- Interpretation (AST -> Unit)

Visual



CALC: *a (very) simple language*

- A language which handles simple arithmetic with integers: addition, subtraction, multiplication, and division

CALC *Tokens*

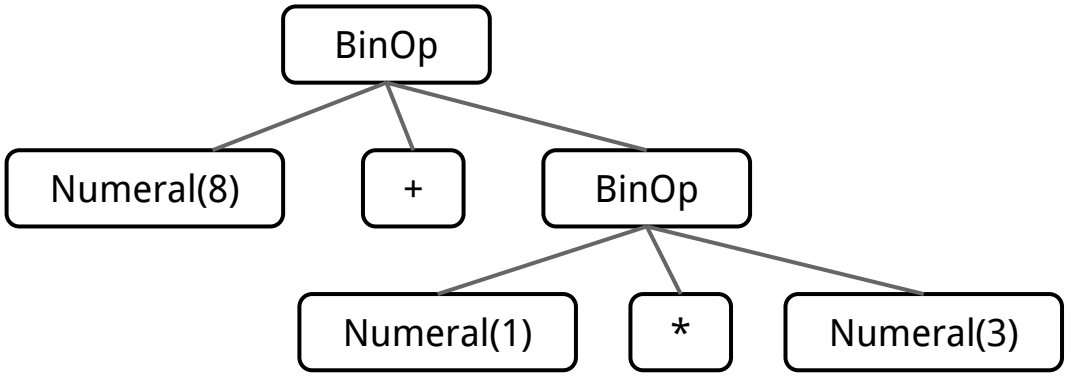
- ['0'-'9']+ {NUMERAL}
- + {PLUS}
- - {MINUS}
- * {TIMES}
- / {DIVIDE}
- ({LPAREN}
-) {RPAREN}

What is syntactic analysis?

- Synonym for parsing
- Takes a string of tokens and transforms them into an abstract syntax tree (AST) according to a grammar

What is an AST?

- A tree representing the structure of the code
- Allows for easy analysis and evaluation
- For example `(8+1*3)` might be parsed as:



What is a grammar?

- A set of rules for rewriting strings
- Each rule dictates how to transform one kind of symbol into another
- Three types of symbols:
  - Start symbol
  - Nonterminal symbol
  - Terminal symbol

Grammar example

```
A -> 0A1
A -> B
B -> #
```

- Symbol types:
  - Start symbol: `A`
  - Nonterminal symbols: `A, B`
  - Terminals: `0, 1, #`
- Example derivation:
  - `A -> 0A1 -> 00A11 -> 00B11 -> 00#11`

CALC *Grammar*

```
E -> EBE | `LPAREN`E`RPAREN` | `NUMERAL`
B -> `PLUS` | `MINUS` | `TIMES` | `DIVIDE`
```

- Valid strings:
  - `(3)`
  - `((8) + 2)`
  - `4 * (9 - 1)`

Construct a recursive descent parser

- Write recursive functions that mimic the rewrite rules of the grammar
- One function for every rule

No left recursion

- A recursive descent parser must use a grammar with no left recursion
- Left recursion occurs when the same symbol to the left of an arrow appears immediately to the right of the arrow:
  - `E -> EBE`
- Leads to infinite parsing loops
  - `E -> EBE -> EBEBE -> EBEEBEBE`
  - There is always an `E` on the left hand side so we can never match any input

CALC *Grammar (with no left recursion)*

```
E -> ABE | A
A -> `NUMERAL` | `LPAREN`E`RPAREN`
B -> `PLUS` | `MINUS` | `TIMES` | `DIVIDE`
```

Let's build!

You build (an evaluator)!