Building a Parser for Fun and Profit: Why 2 + 2 = 4

Alejandro Santos // https://github.com/ALEJOLP

Introduction

Welcome to London Algorithms Meetup 2021!

About Alejandro:

Computer Scientist, and Software Developer. I work in Large Data Systems, and on the side I like to learn about programming languages parsers.

I used to teach Data Structures and Algorithms (UNLP, Argentina), then worked as a scientist at CERN. Now I work with market depth data in London.

Outline

In today's presentation we'll see:

- How to hand-craft a simple parser (for fun and profit!)
- Some examples of this technique
- How to create a parser in your day-to-day job
- Some pointers and signposts on where to look next

I was asked to write an expression evaluator

The "time to write the code" was limited. The P. Statement was not clear.

Given the string "2 + 2" output "4", and "2 + 3 * 4", output "14" (not 20).

It should work with +, -, *, and /, parenthesis, and have ops precedence.

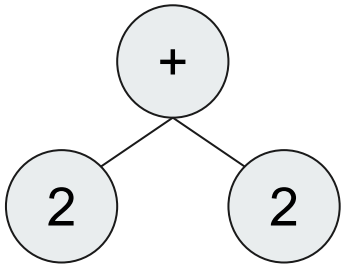
This is a classical parsing exercise. Many different algorithms and approaches. Dijkstra invented a stack-based one (it has limitations).

Recursion to the Rescue

Arithmetic expressions are naturally recursive.

For example 2+2:

It's a tree!

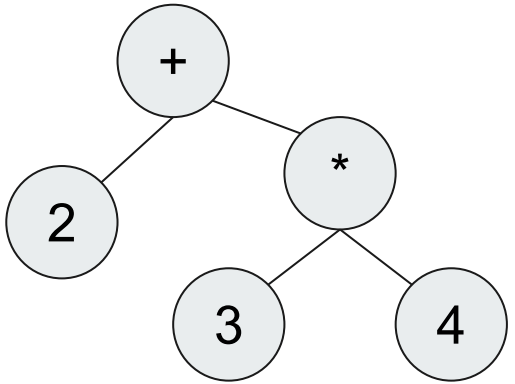



```
def F(Node):
    if Node.Value == '+':
        return F(Node.Left) + F(Node.Right)
    elif Node.Value == '-':
        return F(Node.Left) - F(Node.Right)
    elif IsNumber(Node.Value):
        return int(Node.Value)
```

Another example

Another example: 2 + 3 * 4

The product has precedence over addition.



The Problem

Having the tree already built makes the problem easy.

The problem is to convert the String to a Tree.

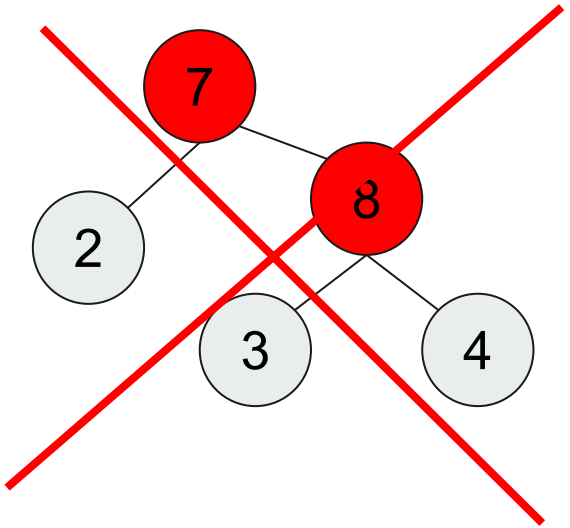
This is where many algorithms exist. Some more complex than others.

The one I like: hand-craft one derived from a EBNF.

EBNF

The Backus–Naur form or Backus normal form (BNF) allows to define what's called a "Grammar": **the rules of the tree**.

For example, every leaf of the **Parse Tree** has to be a number, and every interior node shall not be a number: "7 3 + 2"



Writing the EBNF

Writing a EBNF can get tricky. This is the most difficult step: **careful with recursion loops and ambiguous rules**. Many examples are already available online, do use them.

- Grammar: (the {...} means 0+ repetition)
- **expr**: term | term { '+' term } | term { '-' term }
- **term**: factor | factor { '*' factor } | factor { '/' factor }
- **factor**: '-' factor | '(' expr ')' | **value**

Example: "2 + 3 * 4"

The Recursive-Descent Parser

1. Convert the input string to a list of 'tokens':
   a. '37 * (4 + 12)' → ['37', '*', '(', '4', '+', '12', ')']

For each rule:

2. Write one recursive function, taking the input and start position.
3. Scan the input from left to right, follow a **Greedy** approach.
4. Returns two things: the node, and the next position in the input.

Example: value rule

The 'value' rule builds the node of the tree of numbers. It will always be a leaf of the tree (due to the rules).

```
def Parse_Value(input, pos):
    return (int(input[pos], pos + 1))
```

The 'expr' rule

1. Create a new 'Node'
2. Add the first Term child
3. Greedily match '+' and a new Term. Add both as childs.
4. Stop when something is missing.
5. Return the 'Node'

```
def Parse_Expr(input, pos):
    # expr: term | term {+ term} | term {- term}
    result_node = Node('expr')
    item_node, item_pos = Parse_Term(input, pos)
    result_node.Childs.append(item_node)
    next_pos = item_pos

    while next_pos < len(input):
        t = input[next_pos]
        if t in ['+', '-']:
            other_node, other_pos =
Parse_Term(input, next_pos + 1)
            result_node.Childs.append(t)
            result_node.Childs.append(other_node)
            next_pos = other_pos
        else:
            break

    return (result_node, next_pos)
```

The Recursive-Descent Parser (2)

Finally, at the end, check that all the input was consumed.

This parser is simple to write in code. The difficult step is the EBNF.

It's recursive, and prone to Stack Overflow errors.

It's can be slow, because it tries all the rules in sequence.

Error checking (ie, syntax errors) are also difficult.

Some open-source code

The parser code in Java, including tokenizer and test-cases:

https://github.com/alejolp/simple-expr-eval-java/

Parsing Python EBNF rules from code, a Parser Parser:

https://github.com/alejolp/pppp

The Despacito Programming Language

Despacito is a programming language following the Latin-American culture. It is open source and free software (BSD license). Despacito is **Turing-complete**, compiled, statically typed, and supports arrays, functions and recursion.

The Despacito compiler is implemented in Python 3 in about 1000 lines of code, without using any external libraries (ie, parser generators) or regular expressions. Source code is available at GitHub:

https://github.com/Despacito-Lang/Despacito

How to Create a Parser in your day-to-day Job

The best library I know is **ANTLR4**. Do use it !

The Generator is Java, but the runtime supports: Java, C# (and an alternate C# target), Python (2 and 3), JavaScript, Go, C++, Swift, PHP, DART.

The Parsing Algorithm produced is both fast and efficient, https://www.antlr.org/papers/allstar-techreport.pdf

In their GitHub repo there's a very large collection of Grammars! https://github.com/antlr/grammars-v4

Parsing Python with AST

Python exposes its own parser with the AST module. This is the best way to parse Python code, should you have to.

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
   body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

Other Interesting Examples

SLY: SLY is a lexing and parsing library for Python. Under the covers, it's based on the same LALR(1) algorithm as yacc, bison, PLY and similar tools.

https://github.com/dabeaz/sly

Swig is wrapper generation tool that automatically connects existing C/C++ code to a wide variety of other programming languages including Python, Perl, Tcl, Java, C#, and many others.

http://www.swig.org/

Node.js HTTP Parser

The node.js team wrote their own parser-generator for parsing HTTP.

The generated code is obfuscated C.

https://github.com/nodejs/llhttp

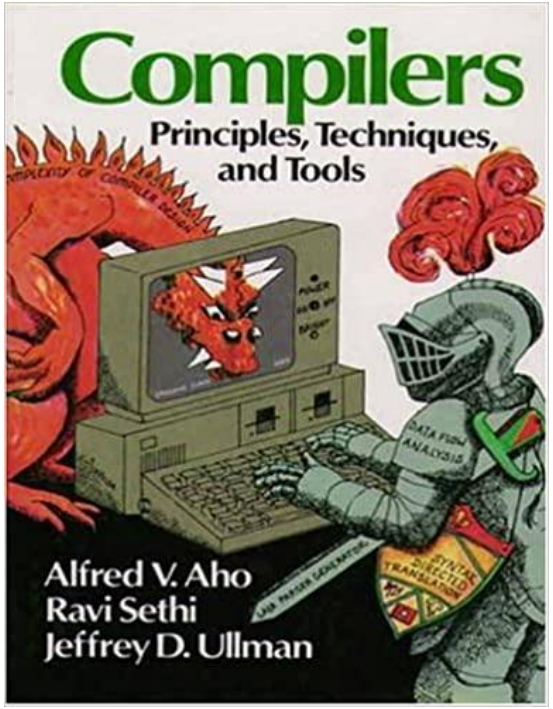
Originally, they were using NGINX's HTTP parser, hand-crafted.

Books

Compilers: Principles, Techniques, and Tools, by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.

Engineering a Compiler, by Keith D. Cooper and Linda Torczon

Crenshaw, let's build a compiler: https://github.com/pepaslabs/crenshaw-lets-build-a-compiler



Backup Slides

Parser Combinators in Functional Programming

Each EBNF rule translated to code returns a Maybe (Node, Remaining).

PCs combine simple rules to more complex ones.

https://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf

Left-Recursion Elimination

Left recursion makes grammars not suitable for Recursive Descent: rewrite

Given:

- A: A X | Y

Write:

- A: Y A'
- A': eps | X A'