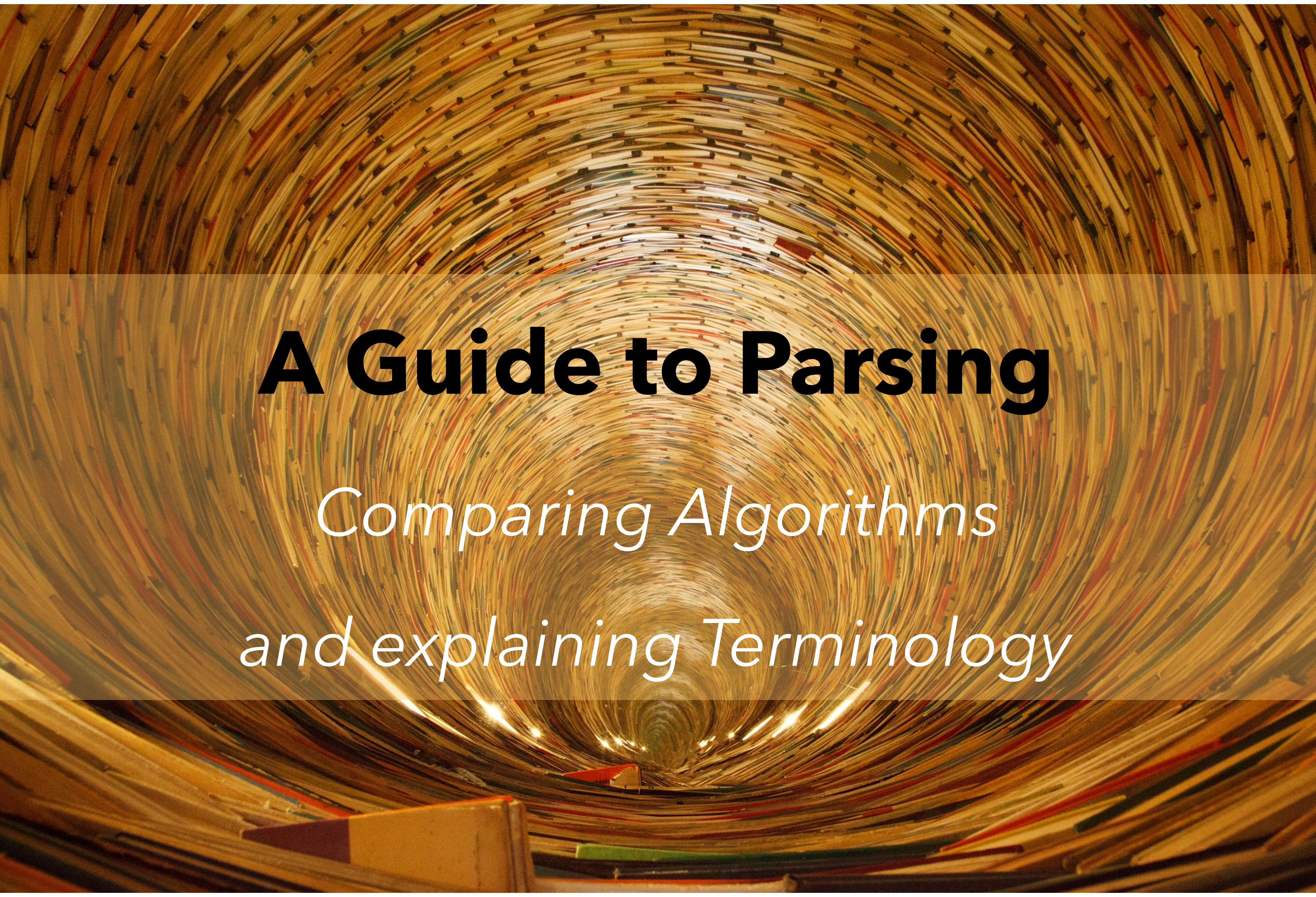


A Guide to Parsing

Comparing Algorithms

and explaining Terminology

FEDERICO
TOMASSETTI
Software Architect

Learn more at tomassetti.me

We have already introduced a few parsing terms, while listing the major tools and libraries used for parsing in Java, C#, Python and JavaScript. In this article we make a more in-depth presentation of the concepts and algorithms used in parsing, so that you can get a better understanding of this fascinating world.

We have tried to be practical in this article. Our goal is to help practicioners, not to explain the full theory. We just explain what you need to know to understand and build parser.

After the definition of parsing the article is divided in three parts:

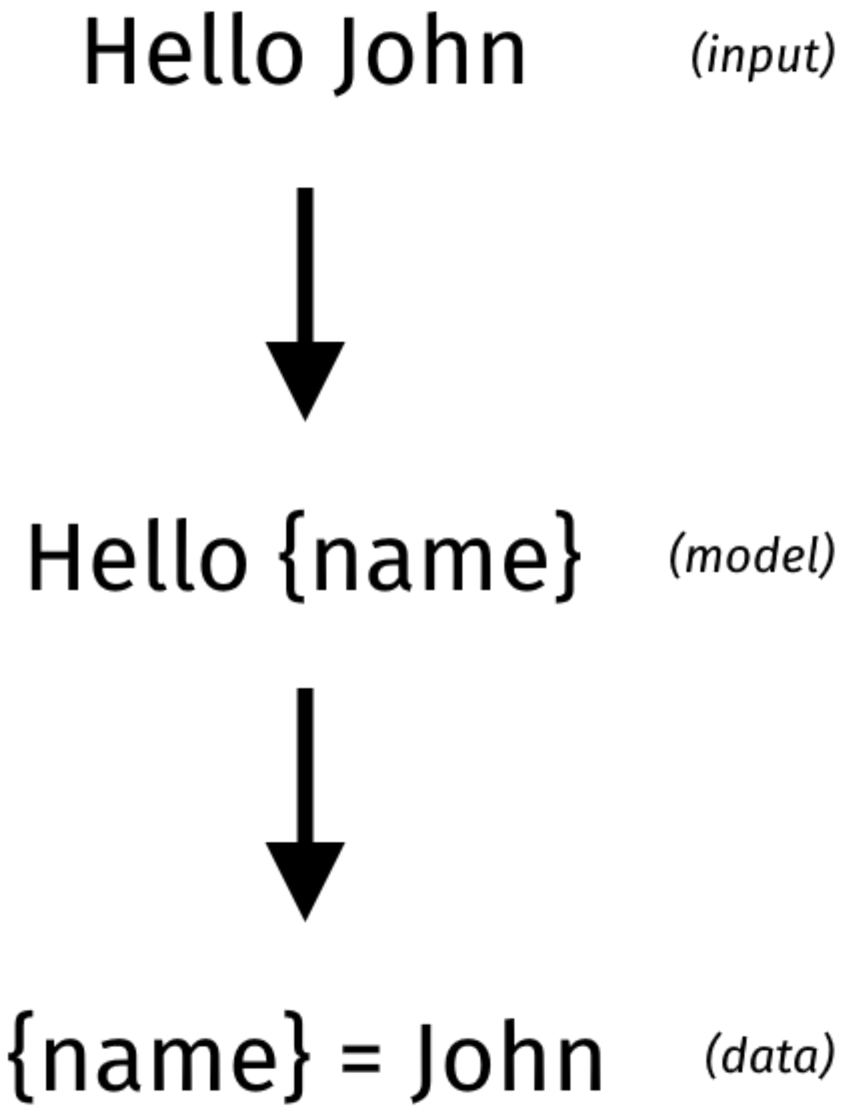
1. **The Big Picture**. A section in which we describe the fundamental terms and components of a parser.
2. **Grammars**. In this part we explain the main formats of a grammar and the most common issues in writing them.
3. **Parsing Algorithms**. Here we discuss all the most used parsing algorithms and say what they are good for.

Table of Contents

Definition of Parsing

The analysis of an input to organize the data according to the rule of a grammar

There are a few ways to define parsing. However the gist remain the same: parsing means to find the underlying structure of the data we are given.

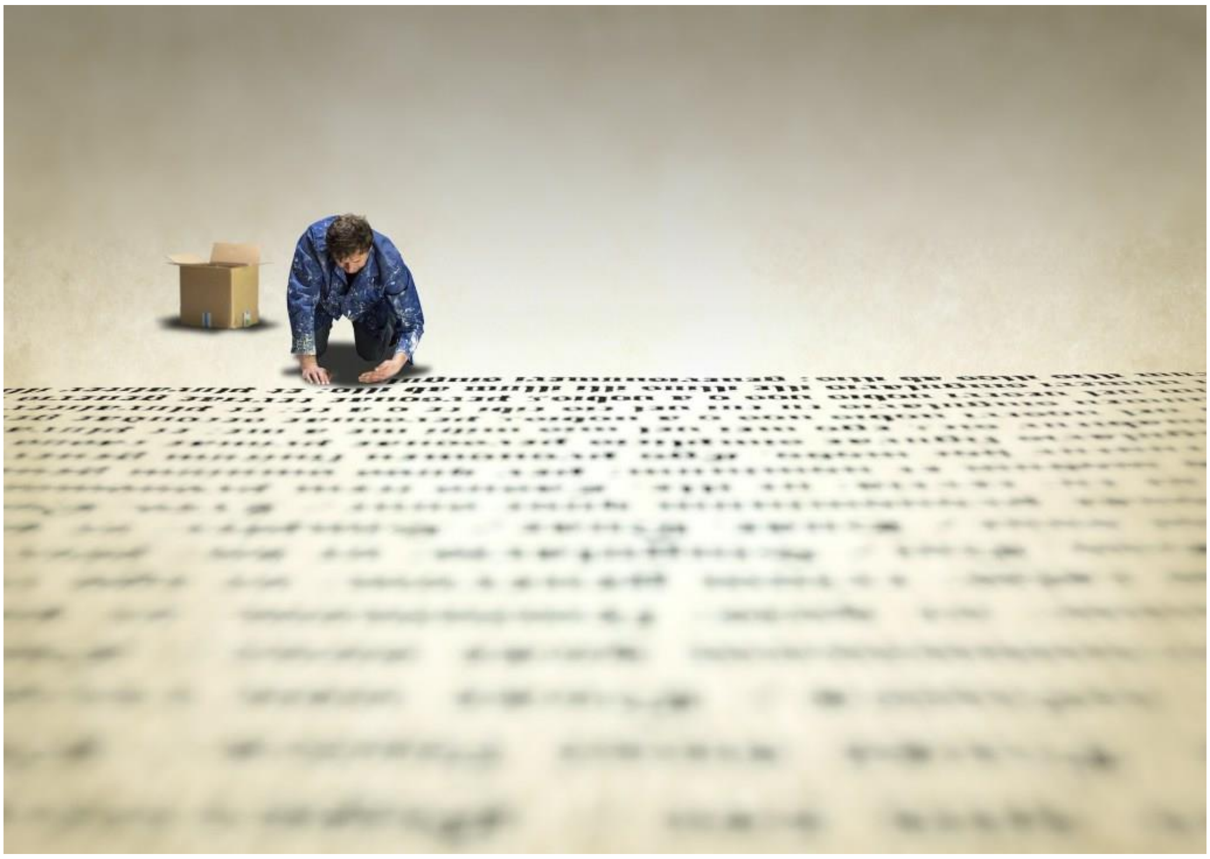


In a way parsing can be considered the inverse of templating: identifying the structure and extracting the data. In templating instead we have a structure and we fill it with data. In the case of parsing you have to determine the model from the raw representation. While for templating you have to combine the data with the model, to create the raw representation. Raw representation is usually text, but it can also be binary data.

Fundamentally parsing is necessary because different entities need the data to be in different forms. Parsing allows to transform data in a way that can be understood by a specific software. The obvious example are programs: they are written by humans, but they must be executed by computers. So humans write them in a form that they can understand, then a software transform them in a way that can be used by a computer.

However parsing might be necessary even when passing data between two software that have different needs. For instance, it is needed when you have to serialize or deserialize a class.

The Big Picture



In this section we are going to describe the fundamental components of a parser. We are not trying to give you formal explanations, but practical ones.

Regular Expressions

A sequence of characters that can be defined by a pattern

Regular expression are often touted as the thing you should not use for parsing. This is not strictly correct, because you can use regular expressions for parsing simple input. The problem is that some programmers only know regular expressions. So they use them to try to parse everything, even the things they should not. The result is usually a series of regular expressions hacked together, that are very fragile.

You can use regular expressions to parse some simpler languages, but this exclude most programming languages. Even the ones that look simple enough like HTML. In fact, languages that can be parsed with just regular expressions are called regular languages. There is a formal mathematical definition, but that is beyond the scope of this article.

Though one important consequence of the theory is that regular languages can be parsed or expressed also by a finite state machine. That is to say regular expressions and finite state machines are equally powerful. This is the reason because they are used to implement lexers, as we are going to see later.

A regular language can be defined by a series of regular expressions, while more complex languages need something more. A simple rule of thumb is: if a grammar of a language has recursive, or nested, elements it is not a regular language. For instance, HTML can contain an arbitrary number of tags inside any tag, therefore is not a regular language and it cannot be parsed using solely regular expressions, no matter how clever.

Regular Expressions in Grammars

The familiarity of a typical programmer with regular expressions lend them to be often used to define the grammar of a language. More precisely, their syntax is used to define the rules of a lexer or a parser. For example, the *Kleene star* (*) is used in a rule to indicate that a particular element can be present zero or an infinite amount of times.

The definition of the rule should not be confused with how the actual lexer or parser is implemented. You can implement a lexer using the regular expression engine provided by your language. However usually the regular expressions defined in the grammar are converted are actually converted to a finite-state machine to gain better performance.

Structure of a Parser

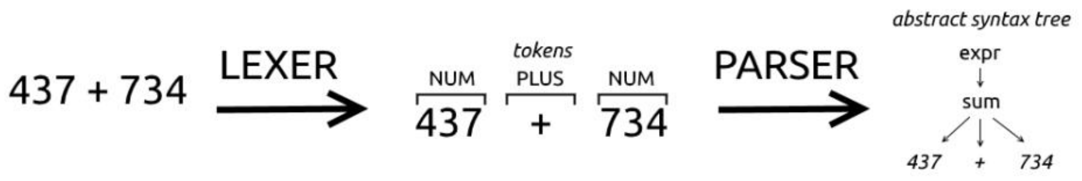
Having now clarified the role of regular expressions, we can look at the general structure of a parser. A complete parser is usually composed of two parts: a *lexer*, also known as *scanner* or *tokenizer*, and the proper parser. The parser need the lexer because it does not work directly on the text, but on the output produced by the lexer. Not all parsers adopt this two-steps schema: some parsers do not depend on a separate lexer and they combine the two steps. They are called *scannerless parsers*.

A lexer and a parser work in sequence: the lexer scans the input and produces the matching tokens, the parser then scans the tokens and produces the parsing result.

Let’s look at the following example and imagine that we are trying to parse an addition.

```
437 + 734
```

The lexer scans the text and find 4, 3, 7 and then a space ( ). The job of the lexer is to recognize that the characters 437 constitute one token of type **NUM**. Then the lexer finds a + symbol, which corresponds to a second token of type **PLUS**, and lastly it finds another token of type **NUM**.



The parser will typically combine the tokens produced by the lexer and group them.

The definitions used by lexers and parsers are called *rules* or *productions*. In our example a lexer rule will specify that a sequence of digits correspond to a token of type NUM, while a parser rule will specify that a sequence of tokens of type NUM, PLUS, NUM corresponds to a **sum** expression.

It is now typical to find suites that can generate both a lexer and parser. In the past it was instead more common to combine two different tools: one to produce the lexer and one to produce the parser. For example, this was the case of the venerable lex & yacc couple: using lex it was possible to generate a lexer, while using yacc it was possible to generate a parser

Scannerless Parsers

Scannerless parsers operate differently because they process directly the original text, instead of processing a list of tokens produced by a lexer. That is to say, a scannerless parser works as a lexer and a parser combined.

While it is certainly important to know for debugging purposes if a particular parsing tool is a scannerless parser or not, in many cases it is not relevant to define a grammar. That is because you usually still define pattern that group sequence of characters to create (virtual) tokens; then you combine them to obtain the final result. This is simply because it is more convenient to do so. In other words, the grammar of a scannerless parser looks very similar to one for a tool with separate steps. Again, you should not confuse how things are defined for your convenience and how things works behind the scene.

Grammar

A formal grammar is a set of rules that describes syntactically a language

There are two important parts of this definition: a grammar describes a language, but this description pertains only the syntax of the language and not the semantics. That is to say, it defines its structure, but not its meaning. The correctness of the meaning of the input must be checked, if necessary, in some other way.

For instance, imagine that we want to define a grammar for the language shown in the paragraph **Definition**.

```
HELLO: "Hello"
NAME: [a-zA-Z]+
greeting: HELLO NAME
```

This grammar accepts input such as `"Hello Michael"` and `"Hello Programming"`. They are both syntactically correct, but we know that “Programming” it is not a name. Thus it is semantically wrong. A grammar does not specify semantic rules, and they are not verified by a parser. You would need to ensure the validity of the provided name some other way, for example comparing it to a database of valid names.

Anatomy of a Grammar

To define the elements of a grammars, let us look at an example in the most used format to describe grammars: the **Backus-Naur Form (BNF)**. This format has many variants, including the **Extended Backus-Naur Form**. The Extended variant has the advantage of including a simple way to denote repetitions. Another notable variant is the **Augmented Backus-Naur Form** which is mostly used to describe bidirectional communications protocols.

A typical rule in a Backus-Naur grammar looks like this:

```
<symbol> ::= __expression__
```

The **<simbol>** is a *nonterminal*, which means that it can be replaced by the group of elements on the right, **__expression__**. The element **__expression__** could contains other nonterminal symbols or terminal ones. *Terminal* symbols are simply the ones that do not appear as a **<symbol>** anywhere in the grammar. A typical example of a terminal symbol is a string of characters, like “Hello”.

A *rule* can also be called *production rule*. Technically it defines a transformation between the nonterminal and the set of nonterminals and terminals on the right.

Types of Grammars

There are mainly two kinds of grammars used in parsing: **regular grammars** and **context-free grammars**. Usually to one kind of grammar corresponds the same kind of language: a regular grammar defines a regular language and so on. However, there is also a more recent kind of grammars called **Parsing Expression Grammar** (PEG) that is equally powerful as context-free grammars and thus define a context-free language. The difference between the two is in the notation and the interpretation of the rules.

As we already mentioned the two kinds of languages are in a hierarchy of complexity: regular languages are simpler than context-free languages.

A relatively easy way to distinguish the two grammars would be that the `__expression__` of a regular grammar, that is to say the right side of the rule, could be only one of:

- the empty string
- a single terminal symbol
- a single terminal symbol followed by a nonterminal symbol

In practice though this is harder to check, because a particular tool could allow to use more terminal symbols in one definition. Then the tool itself will automatically transform this expression in an equivalent series of expressions all belonging to one of the three mentioned cases.

So you could write an expression that is incompatible with a regular language, but the expression will be transformed in the proper form. In other words, the tool could provide syntactic sugar for writing grammars.

In a later paragraph we are going to discuss more at length the different kinds of grammars and their formats.

Lexer

A lexer transforms a sequence of characters in a sequence of tokens

Lexers are also known as *scanners* or *tokenizers*. Lexers play a role in parsing, because they transform the initial input in a form that is more manageable by the proper parser, who works at a later stage. Typically lexers are easier to write than parsers. Although there are special cases when both are quite complicated, for instance in the case of C (see the lexer hack).

A very important part of the job of the lexer is dealing with whitespace. Most of the times you want the lexer to discard whitespace. That is because otherwise the parser would have to check for the presence of whitespace between every single token, which would quickly become annoying.

There are cases when you cannot do that, because whitespace is relevant to the language, like in the case of Python where is used to identify block of code. Even in these cases, though, usually it is the lexer that deals with the problem of distinguishing the relevant whitespace from the irrelevant one. Which means that you want the lexer to understand which whitespace is relevant to parsing. For example, when parsing Python you want the lexer to check if whitespace define indentation (relevant) or space between the keyword `if` and the following expression (irrelevant).

Where the Lexer Ends and the Parser Begins

Given that lexers are almost exclusively used in conjunction with parsers, the dividing line between the two can be blurred at times. That is because the parsing must produce a result that is useful for the particular need of a program. So there is not just one correct way of parsing something, but you care about only the one way that serves your needs.

For example, imagine that you are creating a program that must parse the logs of a server to save them in a database. For this goal, the lexer will identify the series of numbers and dots and transform them in a IPv4 token.

```
IPv4: [0-9]+ "." [0-9]+ "." [0-9]+ "." [0-9]+
```

Then the parser will analyze the sequence of tokens to determine if it is a message, a warning, etc.

What would happen instead if you were developing software that had to use the IP address to identify the country of the visitor? Probably you would want the lexer to identity the octets of the address, for later use, and make IPv4 a parser element.

```
DOT   : "."
OCTET : [0-9]+

ipv4  : OCTET DOT OCTET DOT OCTET DOT OCTET
```

This is one example of how the same information can be parsed in different ways because of different goals.

Parser

In the context of parsing, parser can refer both to the software that perform the whole process and also just to the proper parser, that analyze the tokens produced by the lexer. This is simply a consequence of the fact that the parser takes care of the most important and difficult part of the whole process of parsing. By most important we mean that the user cares about the most and will actually sees. In fact, as we said, the lexer works as an helper to facilitate the work of the parser.

In any sense you mean the parser its output is an organized structure of the code, usually a tree. The tree can be a **parse tree** or an **abstract syntax tree**. They are both trees, but they differ in how closely they represent the actual code written and the intermediate elements defined by the parser. The line between the two can be blurry at times, we are going to see their differences a bit better in a later paragraph.

The form of a tree is chosen because it is an easy and natural way to work with the code at different levels of detail. For example, a class in C# has one body, this body is made of one statement, the block statement, that is a list of statements enclosed in a pair of curly brackets, and so on…

Syntactic vs Semantic Correctness

A parser is a fundamental part of a compiler, or an interpreter, but of course can be part of a variety of other software. For example, in our article Generate diagrams from C# source code using Roslyn we parsed C# files to produce diagrams.

The parser can only check the syntactic correctness of piece of code, but the compiler can use its output also in the process of checking the semantic validity of the same piece of code.

Let us see an example of code that is syntactically correct, but semantically incorrect.

```
int x = 10
int sum = x + y
```

The problem is that one variable (y) is never defined and thus if executed the program will fail. However the parser has no way of knowing this, because it does not keep track of variables, it just looks at the structure of the code.

A compiler instead would typically traverse the parse tree a first time and keeps a list of all the variables that are defined. Then it traverse the parse tree a second time and checks whether the variables that are used are all properly defined. In this example they are not and it will throw an error. So that is one way the parse tree can also be used to check the semantics by the compiler.

Scannerless Parser

A *scannerless parser*, or more rarely a *lexerless parser*, is a parser that performs the tokenization (i.e., the trasformation of sequence of characters in tokens) and the proper parsing in a single step. In theory having a separate lexer and parser is preferable because it allows a clearer separation of objectives and the creation of a more modular parser.

Scannerless parser is a better design for a language where a clear distinction between lexer and parser is difficult or unnecessary. An example is a parser for a markup language, where special markers are inserted in a sea of text. It can also facilitates the handling of languages where traditional lexing is difficult, like C. That is because a scannerless parser can more easily deal with complex tokenizations.

Issues With Parsing Real Programming Languages

In theory contemporary parsing is designed to handle real programming languages, in practice there are challenges with some real programming languages. At least, it might be harder parsing them using normal parsing generator tools.

Context-sensitive Parts

Parsing tools are traditionally designed to handle context-free languages, but sometimes the languages are context-sensitive. This might be the case to simplify the life of programmers or simply because of a bad design. I remember reading about a programmer that thought it could produce a parser for C in a week, but then it found so many corner cases that a year later he was still working on it…

A typical example of context-sensitive elements are the so-called soft keywords, i.e., strings that can be considered keywords in certain places, but otherwise can be used as identifiers).

Whitespace

Whitespace plays a significant role in some languages. The most well-known example is Python, where the indentation of a statement indicate if it parts of a certain block of code.

In most places whitespace is irrelevant even in Python: spaces between words or keywords do not matter. The real problem is indentation, that is used to identify block of code. The simplest way to deal with it is to check the indentation at the start of the line and transform in the proper token, i.e., to create a token when the indentation changes from the previous line.

In practice, a custom function in the lexer produce INDENT and DEDENT tokens, when the indentation increases or decreases. These tokens play the role that in C-like languages is played by curly brackets: they indicate the start and end of code blocks.

This approach makes the lexing context-sensitive, instead of context-free. This complicates parsing and you normally would not want to do it, but you are forced to do in this case.

Multiple Syntaxes

Another common issue is dealing with the fact that a language might actually include a few different syntaxes. In other words, the same source file may contain sections of code that follow a different syntax. In the context of parsing effectively the same source file contains different languages. The most famous example is probably the C or C++ preprocessor, which is actually a fairly complicated language on its own and can magically appear inside any random C code.

An easier case to deal with are annotations, that are present in many contemporary programming languages. Among other things they can be used to process the code before it arrives to the compiler. They can command the annotation processor to transform the code in some way, for instance to execute a specific function before executing the annotated one. They are easier to manage, because they can appear only in specific places.

Dangling Else

The dangling else is a common problem in parsing linked to the *if-then-else* statement. Since the else clause is optional a sequence of *if* statements could be ambiguous. For example this one.

```
if one
   then if two
       then two
else ???
```

It is unclear if the `else` belongs to the first `if` or the second one.

To be fair, this is for the most part a problem of language design. Most of the solutions do not really complicate parsing that much, for example requiring the use of an `endif` or requiring the use of blocks to delimit the `if` statement when it includes an else clause.

However there are also languages that offer no solution, that is to say that are designed ambiguously, for example (you guessed it) C. The conventional approach is to associate the else to the nearest if statement, which makes the parsing context-sensitive.

Parsing Tree and Abstract Syntax Tree

There are two terms that are related and sometimes they are used interchangeably: *Parse Tree* and *Abstract Syntax Tree* (AST). Technically the parse tree could also be called Concrete Syntax Tree (CST), because it should reflect more concretely the actual syntax of the input, at least compared to the AST.

Conceptually they are very similar. They are both **trees**: there is a root that has nodes representing the whole source code. The root have children nodes, that contains subtrees representing smaller and smaller portions of code, until single tokens (terminals) appears in the tree.

The difference is in the level of abstraction: a parse tree might contains all the tokens which appeared in the program and possibly a set of intermediate rules. The AST instead is a polished version of the parse tree, in which only the information relevant to understanding the code is maintained. We are going to see an example of an intermediate rule in the next section.

Some information might be absent both in the AST and the parse tree. For instance, comments and grouping symbols (i.e., parentheses) are usually not represented. Things like comments are superfluous for a program and grouping symbols are implicitly defined by the structure of the tree.

From Parse Tree to Abstract Syntax Tree

A parse tree is a representation of the code closer to the concrete syntax. It shows many details of the implementation of the parser. For instance, usually each rule corresponds to a specific type of a node. A parse tree is usually transformed in an AST by the user, possibly with some help from the parser generator. A common help allows to annotate some rule in the grammar, to exclude the corresponding nodes from the generated tree. Another one is an option to collapse some kinds of nodes if they have only one child.

This make sense because the parse tree is easier to produce for the parser, since it is a direct representation of the parsing process. However the AST is simpler and easier to process by the following steps of the program. They typically include all the operations that you may want to perform on the tree: code validation, interpretation, compilation, etc…

Let us look at a simple example to show the difference between a parse tree and an AST. Let’s start with a look at an example grammar.

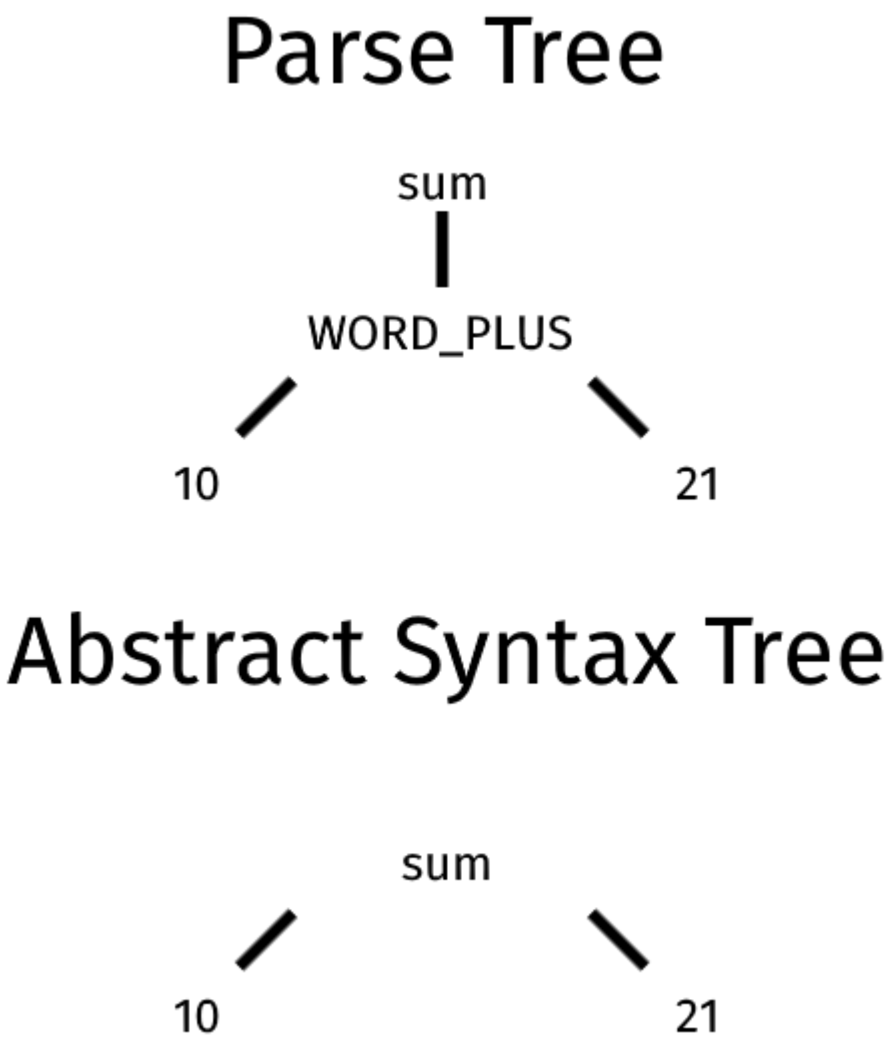
```
// lexer
PLUS: '+'
WORD_PLUS: 'plus'
NUMBER: [0-9]+

// parser
// the pipe | symbol indicate an alternative between the two
sum: NUMBER (PLUS | WORD_PLUS) NUMBER
```

In this grammar we can define a sum using both the symbol plus (+) or the string `plus` as operators. Imagine that you have to parse the following code.

```
10 plus 21
```

These could be the resulting parse tree and abstract syntax tree.



In the AST the indication of the specific operator is disappeared and all that remains is the operation to be performed. The specific operator is an example of an intermediate rule.

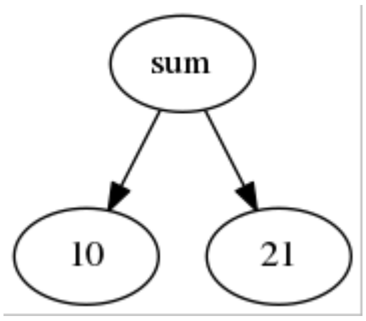
Graphical Representation Of A Tree

The output of a parser is a tree, but the tree can also be represented in graphical ways. That is to allow an easier understanding to the developer. Some parsing generator tools can output a file in the DOT language, a language designed to describe graphs (a tree is a particular kind of graph). Then this file is fed to a program that can create a graphical representation starting from this textual description (e.g., Graphviz).

Let's see a .DOT text, based on the previous sum example.

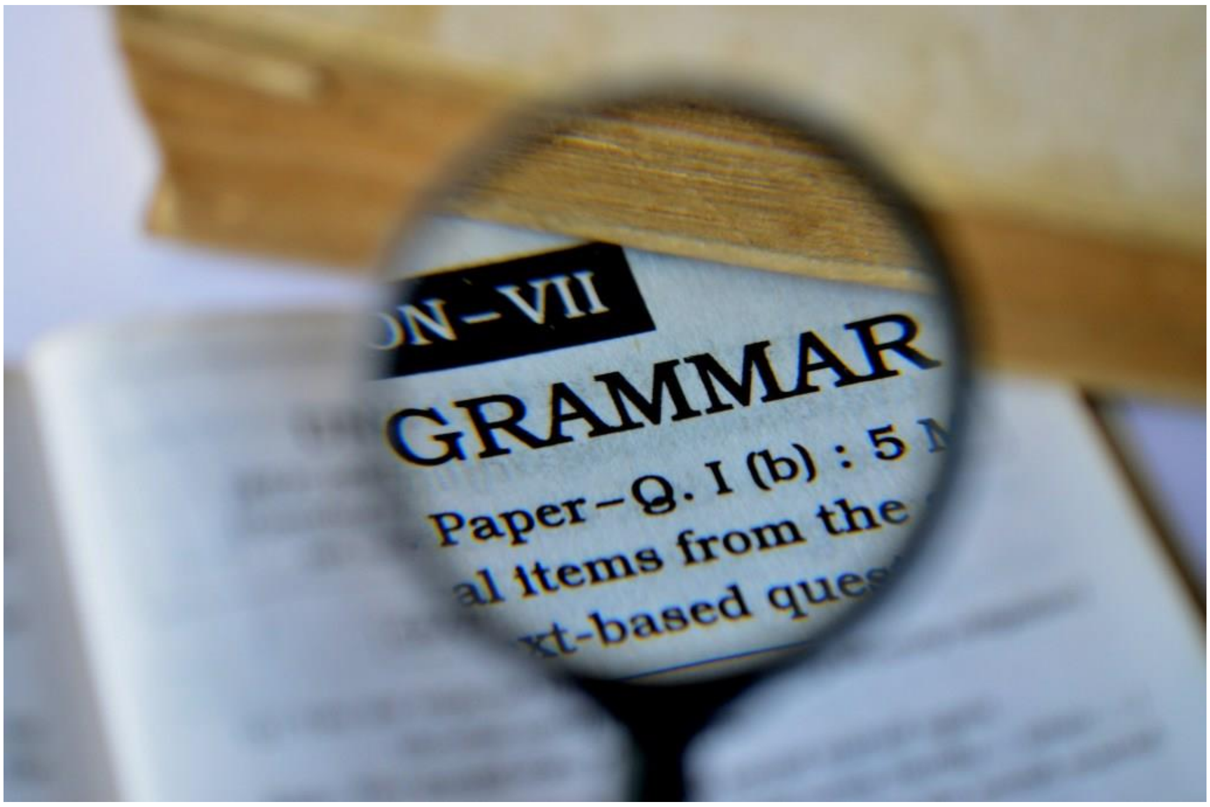
```
digraph sum {
      sum -> 10;
      sum -> 21;
}
```

The appropriate tool can create the following graphical representation.



If you want to see a bit more of DOT you can read our article Language Server Protocol: A Language Server For DOT With Visual Studio Code. In that article we show show how to create a Visual Studio Code plugin that can handle DOT files.

Grammars



Grammars are a set of rules used to describe a language, so it comes naturally to study the formats of the rules. However there are also several elements of a typical grammar that could use further attention. Some of them are due to the fact that a grammar can also be used to define other duties or to execute some code.

Typical Grammar Issues

First we are going to talk about some special rules or issues you might encounter in parsing.

The Missing Tokens

If you read grammars you will probably encounter many in which only a few tokens are defined and not all of them. Like in this grammar:

```
NAME     : [a-zA-Z]+
greeting : "Hello" NAME
```

There is no definition for the token `"Hello"`, but since you know that a parser deals with tokens, you may ask yourself how is this possible. The answer is that some tools generate for you the corresponding token for a string literal, to save you some time.

Be aware that this might be possible only under certain conditions. For instance, with ANTLR you must define all tokens yourself if you define separate lexer and parser grammars.

Left-recursive Rules

In the context of parsers, an important feature is the support for left-recursive rules. This means that a rule starts with a reference to itself. Sometime this reference could also be indirect, that is to say it could appear in another rule referenced by the first one.

Consider for example arithmetic operations. An addition could be described as two expression(s) separated by the plus (+) symbol, but the operands of the additions could be other additions.

```
addition       : expression '+' expression
multiplication : expression '*' expression
// an expression could be an addition or a multiplication or a number
expression     : multiplication | addition | [0-9]+
```

In this example **expression** contains an indirect reference to itself via the rules **addition** and **multiplication**.

This description also matches multiple additions like `5 + 4 + 3`. That is because it can be interpreted as `expression (5) ('+') expression(4+3)` (the rule addition: the first expression corresponds to the option [0-9]+, the second one is another addition). And then `4 + 3` itself can be divided in its two components: `expression(4) ('+') expression(3)` (the rule addition:both the first and second expression corresponds to the option [0-9]+) .

The problem is that left-recursive rules may not be used with some parser generators. The alternative is a long chain of expressions, that takes care also of the precedence of operators. A typical grammar for a parser that does not support such rules would look similar to this one:

```
expression     : addition
addition       : multiplication ('+' multiplication)*
multiplication : atom ('*' atom)*
atom           : [0-9]+
```

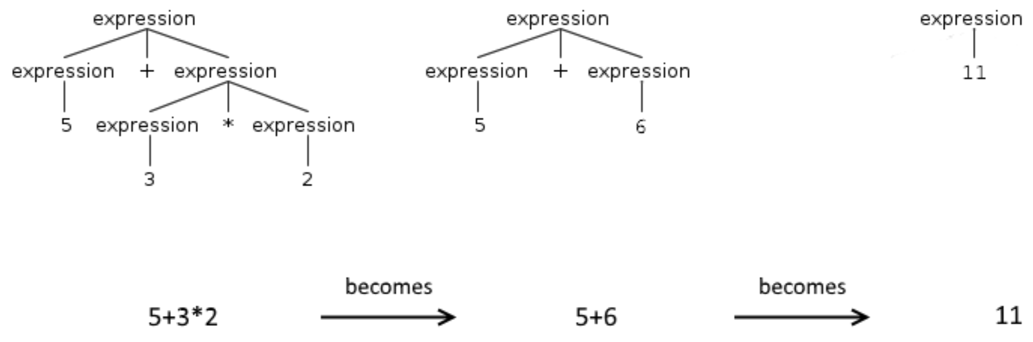
As you can see, the expressions are defined in the inverse order of precedence. So the parser would put the expression with the lower precedence at the lowest level of the three; thus they would be executed first.

Some parser generators support direct left-recursive rules, but not indirect ones. Notice that usually the issue is with the parsing algorithm itself, that does not support left-recursive rules. So the parser generator may transform rules written as left-recursive in the proper way to make it work with its algorithm. In this sense, left-recursive support may be (very useful) syntactic sugar.

How Left-recursive Rules Are Transformed

The specific way in which the rules are transformed vary from one parser generator to the other, however the logic remains the same. The expressions are divided in two groups: the ones with an operator and two operands and the atomic ones. In our example the only atomic expression is a number (`[0-9]+`), but it could also be an expression between parentheses (`(5 + 4)`). That is because in mathematics parentheses are used to increase the precedence of an expression.

Once you have these two groups: you maintain the order of the members of the second group and reverse the order of the members of the first group. The reason is that humans reason on a first come, first serve basis: it is easier to write the expressions in their order of precedence.



However the final form of the parsing is a tree, which operates on a different principle: you start working on the leaves and rise up. So that at the end of this process the root node contain the final result. Which means that in the parsing tree the atomic expressions are at the bottom, while the ones with operators appears in the inverse order in which are applied.

Predicates

Predicates, sometimes called *syntactic* or *semantic predicates*, are special rules that are matched only if a certain condition is met. The condition is defined with code in a programming language supported by the tool for which the grammar is written.

Their advantage is that they allow some form of context-sensitive parsing, which is sometime unavoidable to match certain elements. For instance, they can be used to determine if a sequence of characters that defines a soft keyword is used in a position where it would be a keyword (e.g., the previous token can be followed by the keyword) or it is a simple identifier.

The disadvantages are that they slowdown parsing and they make the grammar dependent on said programming language. That is because the condition is expressed in a programming language and must be checked.

Embedded Actions

Embedded actions identify code that is executed every time the rule is matched. They have the clear disadvantage that make the grammar harder to read, since the rules are surrounded by code. Furthermore, just like predicates, they break the separation between a grammar that describes the language and the code that manipulates the results of the parsing.

Actions are frequently used by less sophisticate parsing generators as the only way to easily execute some code when a node is matched. using these parser generations, the only alternative would be to traverse the tree and execute the proper code yourself. More advanced tools instead allow to use the visitor pattern to execute arbitrary code when needed, and also to govern the traversing of the tree.

They can also be used to add certain tokens or change the generated tree. While ugly, this might be the only practical way to deal with complicate languages, like C, or specific issues, like whitespace in Python.

Formats

There are two main formats for a grammar: BNF (and its variants) and PEG. Many tools implement their own variants of these ideal formats. Some tools use custom formats altogether. A frequent custom format consist in a three parts grammar: options together with custom code, followed by the lexer section and finally the parser one.

Given that a BNF-like format is usually the foundation of a context-free grammar you could also see it identified like the CFG format.

Backus-Naur Form and Its Variants

Backus–Naur Form (BNF) is the most successful format and was even the basis of PEG. However it is quite simple, and thus it is not often used in its base form. A more powerful variant is typically used.

To show why these variants were necessary let's show an example in BNF: the description of a character.

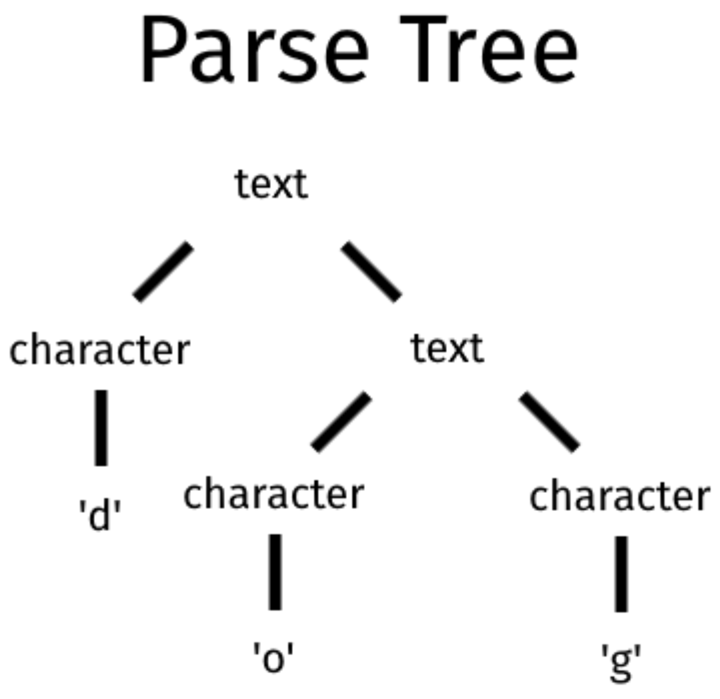
```
<letter>    ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<digit>     ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<character> ::= <letter> | <digit>
```

The symbol **<letter>** can be transformed in any of the letters of the English alphabet, although in our example only lowercase letters are valid. A similar process happens for **<digit>**, which can indicate any among the alternative digits. The first issue is that you have to list all the alternatives individually; you cannot use characters classes like you do with regular expressions. This is annoying, but usually manageable, unless of course you have to list all Unicode characters.

A much harder problem is that there is no easy way to denote optional elements or repetitions. So if you want to do that you have to rely on boolean logic and the alternative (**|**) symbol.

```
<text>      ::= <character> | <text> <character>
```

This rule states that **<text>** can be composed of a character or a shorter **<text>** followed by a **<character>**. This would be the parse tree for the word "dog".



BNF has many other limitations: it makes complicate to use empty strings or the symbols used by the format (e.g., `::=`) in the grammar, it has a verbose syntax (e.g., you have to include terminals between < and >), etc.

Extended Backus-Naur Form

To solve some of these limitations Niklaus Wirth created the **Extended Backus-Naur Form** (EBNF), which includes some concepts from his own Wirth syntax notation.

EBNF is the most used form in contemporary parsing tool, although tools might deviate from the standard notation. EBNF has a much cleaner notation and adopt more operators to deal with concatenation or optional elements.

Let's see how you would write the previous example in EBNF.

```
letter    = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
| "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z" ;
digit     = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
;
character = letter | digit ;
text      = character , { character } ;
```

The **text** symbol now it is described more easily with the help of the concatenation operator (`,`) and the optional one (`{ .. }`). The resulting parse tree would also be simpler. The standard EBNF still presents some problems, like the unfamiliar syntax. To overcome this issue most parsing tools adopt a syntax inspired by regular expressions and also support additional elements like characters classes.

If you need an in-depth coverage of the format you can read our article on EBNF.

ABNF

Augmented BNF (ABNF) is another variants of BNF, mainly developed to describe bidirectional communication protocols and formalized by the IETF with several documents (see RFC 5234 and updates).

ABNF can be as productive as EBNF, but it had some quirks that limited its adoption outside of internet protocols. For example, until recently the standard dictated that strings were matched in a case-insensitive way, which would have been a problem for matching identifiers in many programming languages.

ABNF has a different syntax from EBNF, for example the alternative operator is the slash (/), and sometimes it is plainly better. For instance, there is no need for a concatenation operator. It also has a few more things than standard EBNF. For instance, it allows you to define numeric ranges, such as `%x30-39` that is equivalent to `[0-9]`. This is also used by the designers themselves, to include standard character classes-like basic rules that the end user can use. An example of such rule is ALPHA, that is equivalent to `[a-zA-Z]`.

PEG

Parsing Expression Grammar (PEG) is a format presented by Brian Ford in a 2004 paper. Technically it derives from an old formal grammar called **Top-Down Parsing Language** (TDPL). However a simple way to describe is: EBNF in the real world.

In fact it looks similar to EBNF, but also directly support things widely used, like character ranges (character classes). Although it also has some differences that are not actually that pragmatic, like using the more formal arrow symbol (←) for assignment, instead of the more common equals symbol (=). The previous example could be written this way with PEG.

```
letter    ← [a-z]
digit     ← [0-9]
character ← letter / digit
text      ← character+
```

As you can see, this is the most obvious way a programmer would write it, with character classes and regular expression operators. The only anomalies are the alternative operator (/) and the arrow character, and in fact many implementation of PEG use the equals character.

The pragmatic approach is the foundation of the PEG formalism: it was created to describe more naturally programming languages. That is because context-free grammar has its origin in the work to describe natural languages. In theory, CFG is a generative grammar, while PEG an analytic grammar.

The first should be a sort of recipe to generate only the valid sentences in the language described by the grammar. It does not have to be related to the parsing algorithm. Instead the second kind define directly the structure and semantics of a parser for that language.

PEG vs CFG

The practical implications of this theoretical difference are limited: PEG is more closely associated to the **packrat** algorithm, but that is basically it. For instance, generally PEG (packrat) does not permits left recursion. Although the algorithm itself can be modified to support it, this eliminate the linear-time parsing property. Also, PEG parsers generally are scannerless parsers.

Probably the most important difference between PEG and CFG is that the ordering of choices is meaningful in PEG, but not in CFG. If there are many possible valid ways to parse an input, a CFG will be ambiguous and thus will return an error. By usually wrong we mean that some parsers that adopt CFGs can deal with ambiguous grammars. For instance, by providing all possible valid results to the developer and let him sort it out. Instead PEG eliminate ambiguity altogether because the first applicable choice will be chosen, thus a PEG can never be ambiguous.

The disadvantage of this approach is that you have to be more careful in listing possible alternatives, because otherwise you could have unexpected consequences. That is to say some choices could never be matched.

In the following example doge will never be matched, since dog comes first it will be picked each time.

```
dog ← 'dog' / 'doge'
```

It is an open area of research whether PEG can describe all grammars that can be defined with CFG, but for all practical purposes they do.

Parsing Algorithms

In theory parsing is a solved problem, but it is the kind of problem that keep being solved again and again. That is to say that there are many different algorithms, each one with strong and weak points, and they are still improved by academics.

In this section we are not going to teach how the implement every one of the parsing algorithm, but we are going to explain their features. The goal is that you can choose with more awareness which parsing tool to use, or which algorithm to study better and implement for your custom parser.

Overview

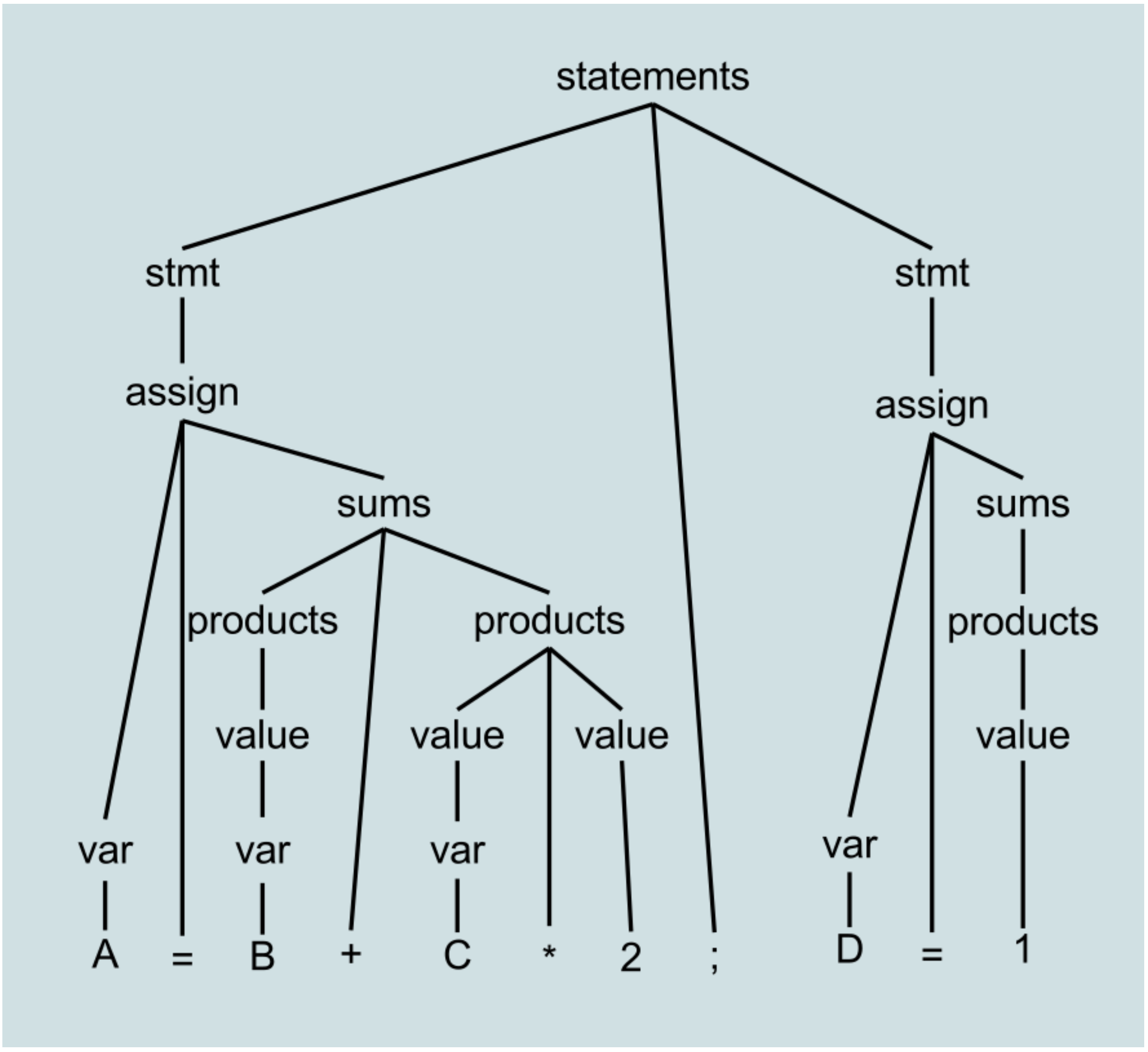
Let’s start with a global overview of the features and strategies of all parsers.

Two Strategies

There are two strategies for parsing: **top-down parsing** and **bottom-up parsing**. Both terms are defined in relation to the parse tree generated by the parser. In a simple way:

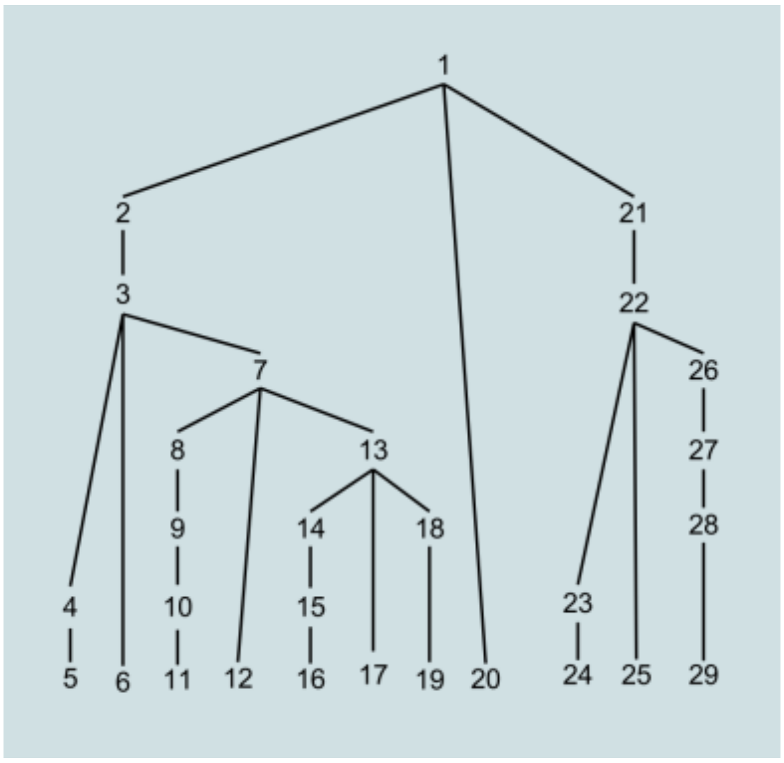
- a top-down parser tries to identity the root of the parse tree first, then it moves down the subtrees, until it find the leaves of the tree.
- a bottom-up parses instead starts from the lowest part of the tree, the leaves, and rise up until it determines the root of the tree.

Let’s see an example, starting with a parse tree.

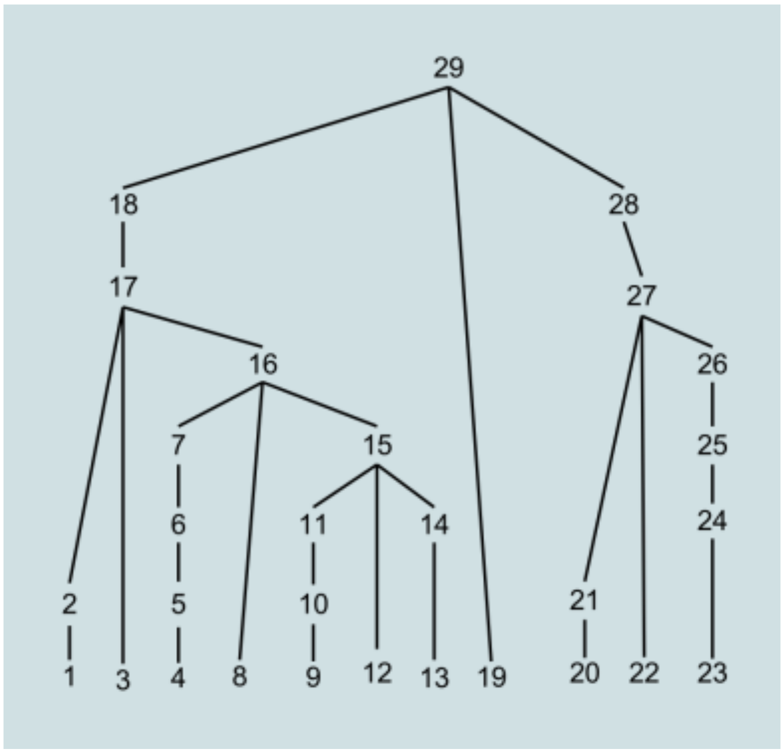


Example Parse Tree from Wikipedia

The same tree would be generate in a different order, by a top-down and a bottom-up parser. In the following images the number indicate the order in which the nodes are created.



Top-down order of generation of the tree (from Wikipedia)



Bottom-up order of generation of the tree (from Wikipedia)

Traditionally top-down parsers were easier to build, but bottom-up parsers were more powerful. Now the situation is more balanced, mostly because of advancement in top-down parsing strategies.

The concept of derivation is closely associated to the strategies. Derivation indicates the order in which the nonterminal elements that appears in the rule, on the right, are applied to obtain the nonterminal symbol, on the left. Using the BNF terminology, it indicates how the elements that appear in __expression__ are used to obtain <symbol>. The two possibilities are: **leftmost derivation** and **rightmost derivation**. The first indicate the rule are applied from left to right, while the second indicate the opposite.

A simple example: imagine that you are trying to parse the symbol `result` which is defined as such in the grammar.

```
expr_one = .. // stuff
expr_two = .. // stuff
result   = expr_one 'operator' expr_two
```

You can apply first the rule for symbol `expr_one` and then `expr_two` or vice versa. In the case of leftmost derivation you pick the first option, while for rightmost derivation you pick the second one.

It is important to understand that the derivation is applied depth-first or recursively. That is to say, it is applied on the starting expression then it is applied again on the intermediate result that is obtained. So, in this example, if after applying the rule corresponding to `expr_one` there is a new nonterminal, that one is transformed first. The nonterminal `expr_two` is applied only when it becomes the first nonterminal and not following the order in the original rule.

Derivation is associated with the two strategies, because for bottom-up parsing you would apply rightmost derivation, while for top-down parsing you would choose leftmost derivation. Note that this has no effect on the final parsing tree, it just affects the intermediate elements and the algorithm used.

Common Elements

Parsers built with top-down and bottom-up strategies shares a few elements that we can talk about.

Lookahead and Backtracking

The terms lookadhead and backtracking do not have a different meaning in parsing than the one they have in the larger computer science field. Lookahead indicates the number of elements, following the current one, that are taken into consideration to decide which current action to take.

A simple example: a parser might check the next token to decide which rule to apply now. When the proper rule is matched the current token is consumed, but the next one remains in the queue.

Backtracking is a technique of an algorithm. It consists in finding a solution to a complex problems by trying partial solutions, and then keep checking the most promising one. If the one that is currently tested fails, then the parser backtracks (i.e., it goes back to the last position that was successfully parsed) and try another one.

They are especially relevant to **LL**, **LR** and **LALR** parsing algorithms, because parsers for language that just needs one lookahead token are easier to build and quicker to run. The lookahead tokens used by such algorithms are indicated between parentheses after the name of the algorithm (e.g., LL(1), LR(*k*)). The notation (*) indicate that the algorithm can check infinite lookahead tokens, although this might affect the performance of the algorithm.

Chart Parsers

Chart parsers are a family of parsers that can be bottom-up (e.g., CYK) or top-down (e.g., Earley). Chart parsers essentially try to avoid backtracking, which can be expensive, by using *dynamic programming*. Dynamic programming, or dynamic optimization, is a general method to break down larger problem in smaller subproblems.

A common dynamic programming algorithm used by chart parser is the Viterbi algorithm. The goal of the algorithm is to find the most likely hidden states given the sequence of known events. Basically given the tokens that we know, we try to find the most probable rules that have produced them.

The name chart parser derives from the fact that the partial results are stored in a structure called chart (usually the chart is a table). The particular technique of storing partial results is called *memoization*. Memoization is also used by other algorithms, unrelated to chart parsers, like **packrat**.

Automatons

Before discussing parsing algorithms we would like to talk about the use of automatons in parsing algorithms. Automaton are a family of abstract machines, among which there is the well known *Turing machine*.

When it comes to parsers you might here the term **(Deterministic) Pushdown Automaton** (PDA) and when you read about lexers you would hear the term **Deterministic Finite Automaton** (DFA). A PDA is more powerful and complex than a DFA (although still simpler than a Turing machine).

Since they define abstract machines, usually they are not directly related to an actual algorithm. Rather, they describe in a formal way the level of complexity that an algorithm must be able to deal with. If somebody says that to solve problem X you need a DFA, he means that you need an algorithm as equally powerful as a DFA.

However since DFA are state machines in the case of lexer the distinction is frequently moot. That is because state machines are relative straightforward to implement (i.e., there are ready to use libraries), so most of the time a DFA is implemented with a state machine. That is why we are going to briefly talk about DFA and why they are frequently used for lexers.

Lexing With a Deterministic Finite Automaton

DFA is a (finite-)state machine, a concept with which we assume you are familiar. Basically, a state machine has many possible states and a transition function for each of them. These transition functions govern how the machine can move from one state to a different one in response to an event.

When used for lexing, the machine is fed the input characters one at a time until it reach an accepted state (i.e., it can build a token).

They are used for a few reasons:

- it has been proven that they recognize exactly the set of regular languages, that is to say they are equally powerful as regular languages
- there are a few mathematical methods to manipulate and check their properties (e.g., whether they can parse all strings or any strings)
- they can work with an efficient online algorithm (see below)

An online algorithm is one that does not need the whole input to work. In the case of a lexer, it means that can recognize a token as soon as its characters are seen by the algorithm. The alternative would be that it needed the whole input to identify each token.

In addition to these properties is fairly easy to transform a set of regular expressions in a DFA, which makes possible to input the rules in a simple way that is familiar to many developers. Then you can automatically convert them in a state machine that can work on them efficiently.

Tables of Parsing Algorithms

We provide a table to offer a summary of the main information needed to understand and implement a specific parser algorithm. You can find more implementations by reading our articles that present parsing tools and libraries for Java, C#, Python and JavaScript.

The table lists:

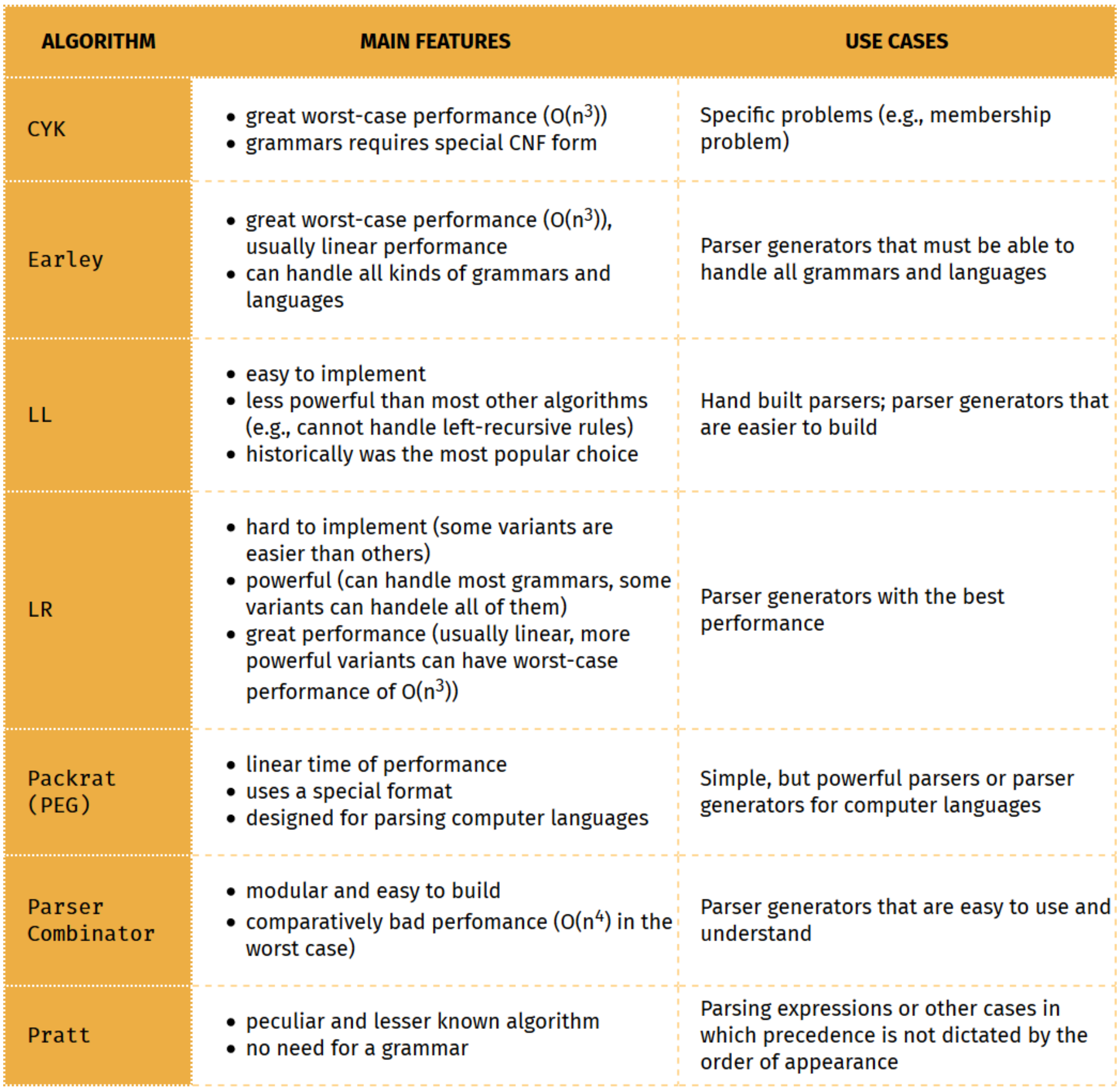
- a formal description, to explain the theory behind the algorithm
- a more practical explanation
- one or two implementations, usually one easier and the other a professional parser. Sometimes, though, there is no easier version or a professional one.

Algorithm	Formal Description	Explanation	Example Implementation
CYK	An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages (PDF)	CKY And Earley Algorithms (PDF)	CYK Algorithm in Golang / Chart-parsers
Earley	An Efficient Context-Free Parsing Algorithm (PDF)	Earley Parsing Explained	Nearley
LL	LL(*): The Foundation of the ANTLR Parser Generator (PDF)	LL Parser on Wikipedia / LL-parser.js	ll(1) parser / ANTLR
LR	On the Translation of Languages from Left to Right (PDF)	Compiler Course	Jison / Bison
Packrat (PEG)	Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking (PDF)	Packrat Parsing: Simple, Powerful, Lazy, Linear TimePackrat Parsing in Scala (PDF)	Companion code of the thesis / Canopy

Parser Combinator	Constructing natural language interpreters in a lazy functional language (PDF)	Parser combinators explained	Understanding Parser Combinators / Sprache
Pratt	Top Down Operator Precedence	Simple Top-Down Parsing in Python	A Pratt Parser in Python / JSLint

To understand how a parsing algorithm works you can also look at the syntax analytic toolkit. It is an educational parser generator that describes the steps that the generated parser takes to accomplish its objective. It implements a LL and a LR algorithm.

The second table shows a summary of the main features of the different parsing algorithms and for what they are generally used.



ALGORITHM	MAIN FEATURES	USE CASES
CYK	• great worst-case performance (O(n^3)) • grammars requires special CNF form	Specific problems (e.g., membership problem)
Earley	• great worst-case performance (O(n^3)), usually linear performance • can handle all kinds of grammars and languages	Parser generators that must be able to handle all grammars and languages
LL	• easy to implement • less powerful than most other algorithms (e.g., cannot handle left-recursive rules) • historically was the most popular choice	Hand built parsers; parser generators that are easier to build
LR	• hard to implement (some variants are easier than others) • powerful (can handle most grammars, some variants can handele all of them) • great performance (usually linear, more powerful variants can have worst-case performance of O(n^3))	Parser generators with the best performance
Packrat (PEG)	• linear time of performance • uses a special format • designed for parsing computer languages	Simple, but powerful parsers or parser generators for computer languages
Parser Combinator	• modular and easy to build • comparatively bad perfomance (O(n^4) in the worst case)	Parser generators that are easy to use and understand
Pratt	• peculiar and lesser known algorithm • no need for a grammar	Parsing expressions or other cases in which precedence is not dictated by the order of appearance

Top-down Algorithms

The top-down strategy is the most widespread of the two strategies and there are several successful algorithms applying it.

LL Parser

LL (**L**eft-to-right read of the input, **L**eftmost derivation) parsers are table-based parsers without backtracking, but with lookahead. Table-based means that they rely on a parsing table to decide which rule to apply. The parsing table use as rows and columns nonterminals and terminals, respectively.

To find the correct rule to apply:

4. firstly the parser look at the current token and the appropriate amount of lookahead tokens
5. then it tries to apply the different rules until it finds the correct match.

The concept of LL parser does not refers to a specific algorithm, but more to a class of parsers. They are defined in relations to grammars. That is to say an LL parser is one that can parse a LL grammar. In turn LL grammars are defined in relation to the number of lookahead tokens that are needed to parse them. This number is indicated between parentheses next to LL, so in the form LL(*k*).

An LL(*k*) parser uses *k* tokens of lookahead and thus it can parse, at most, a grammar which need *k* tokens of lookahead to be parsed. Effectively the concept of LL(*k*) grammar is more widely employed than the corresponding parser. Which means that LL(*k*) grammars are used as a meter when comparing different algorithms. For instance, you would read that PEG parsers can handle LL(*) grammars.

The Value Of LL Grammars

This use of LL grammars is due both to the fact that LL parser are widely used and that they are a bit restrictive. In fact, LL grammars does not support left-recursive rules. You can transform any left-recursive grammar in an equivalent non-recursive form, but this limitation matters for a couple of reasons: productivity and power.

The loss of productivity depends on the requirement that you have to write the grammar in a special way, which takes time. The power is limited because a grammar that might need 1 token of lookahead, when written with a left-recursive rule, might need 2-3 tokens of lookahead, when written in a non-recursive way. So this limitation is not merely annoying, but it is limiting the power of the algorithm, i.e., the grammars it can be used for.

The loss of productivity can be mitigated by an algorithm that automatically transforms a left-recursive grammar in a non-recursive one. ANTLR is a tool that can do that, but, of course, if you are building your own parser, you have to do that yourself.

There are two special kind of LL(*k*) grammars: LL(1) and LL(*). In the past the first kinds were the only one considered practical, because it is easy to build efficient parsers for them. Up to the point that many computer languages were purposefully designed to be described by a LL(1) grammar. An LL(*), also known as *LL-regular*, parser can deal with languages using an infinite amount of lookahead tokens.

On StackOverflow you can read a simple comparison between LL parsers and Recursive Descent parsers or one between LL parsers and LR parsers.

Earley Parser

The Earley parser is a chart parser named after its inventor Jay Earley. The algorithm is usually compared to CYK, another chart parser, that is simpler, but also usually worse in performance and memory. The distinguishing feature of the Earley algorithm is that, in addition to storing partial results, it implement a prediction step to decide which rule is going to try to match next.

The Earley parser fundamentally works by dividing a rule in segments, like in the following example.

```
// an example grammar
HELLO    : "hello"
NAME     : [a-zA-Z]+
greeting : HELLO NAME

// Earley parser would break up greeting like this
// . HELLO NAME
// HELLO . NAME
// HELLO NAME .
```

Then working on this segments, that can be connected at the dot (`.`), tries to reach a completed state, that is to say one with the dot at the end.

The appeal of an Earley parser is that it is guaranteed to be able to parse all context-free languages, while other famous algorithms (e.g., LL, LR) can parse only a subset of them. For instance, it has no problem with left-recursive grammars. More generally, an Earley parser can also deal with nondeterministic and ambiguous grammars.

It can do that at the risk of a worse performance (O(n^3)), in the worst case. However it has a linear time performance for normal grammars. The catch is that the set of languages parsed by more traditional algorithms are the one we are usually interested in.

There is also a side effect of the lack of limitations: by forcing a developer to write the grammar in certain way the parsing can be more efficient. I.e., building a LL(1) grammar might be harder for the developer, but the parser can apply it very efficiently. With Earley you do less work, so the parser does more of it.

In short, Earley allows you to use grammars that are easier to write, but that might be suboptimal in terms of performance.

Earley Use Cases

So Earley parsers are easy to use, but the advantage, in terms of performance, in the average case might be non-existent. This makes the algorithm great for an educational environment or whenever productivity is more relevant than speed.

In the first case is useful, for example, because most of the time the grammars your users write work just fine. The problem is that the parser will throw at them obscure and seemingly random errors. Of course the errors are not actually random, but they are due to the limitations of an algorithm that your users do not know or understand. So you are forcing the user to understand the inner workins of your parser to use it, which should be unnecessary.

An example of when productivity is more important than speed might be a parser generator to implement syntax highlighting, for an editor that need to support many languages. In a similar situation, being able to support quickly new languages might be more desirable than completing the task as soon as possible.

Packrat (PEG)

Packrat is often associated to the formal grammar PEG, since they were invented by the same person: Bryan Ford. Packrat was described first in his thesis: Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking. The title says almost everything that we care about: it has a linear time of execution, also because it does not use backtracking.

The other reason for its efficiency it is memoization: the storing of partial results during the parsing process. The drawback, and the reason because the technique was not used until recently, is the quantity of memory it needs to store all the intermediate results. If the memory required exceed what is available, the algorithm loses its linear time of execution.

Packrat also does not support left-recursive rules, a consequence of the fact that PEG requires to always choose the first option. Actually some variants can support direct left-recursive rules, but at the cost of losing linear complexity.

Packrat parsers can perform with an infinite amount of lookahead, if necessary. This influence the execution time, that in the worst case can be exponential.

Recursive Descent Parser

A recursive descent parser is a parser that works with a set of (mutually) recursive procedures, usually one for each rule of the grammars. Thus the structure of the parser mirrors the structure of the grammar.

The term **predictive parser** is used in a few different ways: some people mean it as a synonym for top-down parser, some as a recursive descent parser that never backtracks.

The opposite to this second meaning is a recursive descent parser that do backtracks. That is to say one that find the rule that matches the input by trying each one of the rules in sequence, and then it goes back each time it fails.

Typically recursive descent parser have problems parsing left-recursive rules, because the algorithm would end up calling the same function again and again. A possible solution to this problem is using tail recursion. Parsers that use this method are called **tail recursive parsers**.

Tail recursion *per se* is simply recursion that happens at the end of the function. However tail recursion is employed in conjuction with transformations of the grammar rules. The combination of transforming the grammar rules and putting recursion at the end of the process allows to deal with left-recursive rules.

Pratt Parser

A Pratt parser is a widely unused, but much appreciated (by the few that knows it) parsing algorithm defined by Vaughan Pratt in a paper called Top Down Operator Precedence. The paper itself starts with a polemic on BNF grammars, which the author argues wrongly are the exclusive concerns of parsing studies. This is one of the reasons for the lack of success. In fact the algorithm does not rely on a grammar, but works directly on tokens, which makes it unusual to parsing experts.

The second reason is that traditional top-down parsers works great if you have a meaningful prefix that helps distinguish between different rules. For example, if you get the token **FOR** you are looking at a **for** statement. Since this essentially applies to all programming languages and their statements, it is easy to understand why the Pratt parser did not change the parsing world.

Where the Pratt algorithm shines is with expressions. In fact, the concept of precedence makes impossible to understand the structure of the input simply by looking at the order in which the tokens are presented.

Basically, the algorithm requires you to assign a precedence value to each operator token and a couple of functions that determines what to do, according to what is on the left and right of the token. Then it uses these values and functions to bind the operations together while it traverse the input.

While the Pratt algorithm has not been overtly successful it is used for parsing expressions. It is also adopted by Douglas Crockford (of JSON fame) for JSLint.

Parser Combinator

A parser combinator is a higher-order function that accepts parser functions as input and return a new parser function as output. A parser function usually means a function that accepts a string and output a parse tree.

A parser combinator is modular and easy to build, but they are also slower (the have O(n^4) complexity in the worst case) and less sophisticated. They are typically adopted for easier parsing tasks or for prototyping. In a sense the user of a parser combinator builds the parser partially by hand, but relying on the hard word done by whoever created the parser combinator.

Generally they do not support left recursive rules, but there are more advanced implementations that do just that. See, for example, the paper Parser Combinators for Ambiguous Left-Recursive Grammars, that also manages to describe an algorithm that has polynomial time of execution.

Many contemporary implementations are called **monadic parser combinator**, since they rely on the structure of functional programming called monad. Monads are a fairly complex concept that we cannot hope to explain here. However basically a monad is able to combine functions and actions relying on a data type. The crucial feature is that the data type specifies how its different values can be combined.

The most basic example is the *Maybe* monad. This is a wrapper around a normal type, like integer, that returns the value itself when the value is valid (e.g., 567), but a special value *Nothing* when it is not (e.g., undefined or division by zero). Thus you can avoid using a *null* value and unceremoniously crashing the program. Instead the *Nothing* value is managed normally, like it would manage any other value.

Bottom-up Algorithms

The bottom-up strategy main success is the family of many different LR parsers. The reason of their relative unpopularity is because historically they have been harder to build, although LR parser are also more powerful than traditional LL(1) grammars. So we mostly concentrate on them, apart from a brief description of CYK parsers.

This means that we avoid talking about the more generic class of Shift-reduce Parser, which also includes LR parsers.

We only say that shift-reduce algorithms works with two steps:

- **Shift**: read one token from the input, that becomes a new (momentarily isolated) node
- **Reduce**: once the proper rule is matched join the resulting tree with a precedent existing subtree

Basically the shift step read the input until completion, while the reduce join the subtrees until the final parse tree is built.

CYK Parser

The Cocke-Younger-Kasami (CYK) it has been formulated independently by the three authors. Its notability is due to a great worst-case performance (O(n^3)), although it is hampered by a comparatively bad performance in most common scenarios.

However the real disadvantage of the algorithm is that it requires the grammars to be expressed in the Chomsky normal form.

That is because the algorithm relies on the properties of this particular form to be able to split the input in half to trying matching all the possibilities. Now, in theory, any context-free grammar can be transformed in a corresponding CNF, but this is seldom practical to do by hand. Imagine being annoyed by the fact that you cannot use left-recursive rules and being asked to learn a special kind of form…

The CYK algorithm is used mostly for specific problems. For instance the membership problem: to determine if a string is compatible with a certain grammar. It can also be used in natural language processing to find the most probable parsing between many options.

For all practical purposes, if you need to parser all context-free grammar with a great worst-case performance, you want to use an Earley parser.

LR Parser

LR (**L**eft-to-right read of the input, **R**ightmost derivation) parsers are bottom-up parsers that can handle deterministic context-free languages in linear time, with lookahead and without backtracking. The invention of LR parsers is credited to the renown Donald Knuth.

Traditionally they have been compared, and competed, with LL parsers. So there is a similar analysis related to the number of lookahead tokens necessary to parse a language. An LR(*k*) parser can parse grammars that need *k* tokens of lookahead to be parsed. However LR grammars are less restrictive, and thus more powerful, than the corresponding LL grammars. For example, there is no need to exclude left-recursive rules.

Technically, LR grammars are a superset of LL grammars. One consequence of this is that you need only LR(1) grammars, so usually the (*k*) is omitted.

They are also table-based, just like LL-parsers, but they need two complicate tables. In very simple terms:

- one table tells the parser what to do depending on the current token, the state is in and the tokens that can possibly follow the current one (l*ookahead sets*)
- the other one tells the parser to which state move next

LR parsers are powerful and have great performance, so where is the catch? The tables they need are hard to build by hand and can grow very large for normal computer languages, so usually they are mostly used through parser generators. If you need to build a parser by hand, you would probably prefer a top-down parser.

Simple LR and Lookahead LR

Parser generators avoid the problem of creating such tables, but they do not solve the issue of the cost of generating and navigating such large tables. So there are simpler alternatives to the **Canonical LR(1) parser**, described by Knuth. These alternatives are less powerful than the original one. They are: **Simple LR parser** (SLR) and **Lookahead LR parser** (LALR). So in order of power we have: LR(1) > LALR(1) > SLR(1) > LR(0).

The names of the two parsers, both invented by Frank DeRemer, are somewhat misleading: one is not really that simple and the other is not the only one that uses lookahead. We can say that one is simpler, and the other rely more heavily on lookahead to make decisions.

Basically they differ in the tables they employ, mostly they change the what "to do" part and the lookahead sets. Which in turn pose different restrictions on the grammars that they can parse. In other words, they use different algorithms to derive the parsing tables from the grammar.

A SLR parser is quite restrictive in practical terms and it is not very used. A LALR parser instead works for most practical grammars and it is widely employed. In fact the popular tools **yacc** and **bison** works with LALR parser tables.

Contrary to LR grammars, LALR and SLR grammars are not a superset of LL grammars. They are not easily comparable: some grammars can be covered by one class and not the other or vice versa.

Generalized LR Parser

Generalized LR parsers (GLR) are more powerful variants of LR parsers. They were described by Bernard Land, in 1974, and implemented the first time by Masaru Tomita, in 1984. The reason for GLR existence is the need to parse nondeterministic and ambiguous grammars.

The power of a GLR parser is not found on its tables, which are equivalent to the one of a traditional LR parser. Instead it can move to different states. In practice, when there is an ambiguity it forks new parser(s) that handle that particular case. These parsers might fail at a later stage and be discarded.

The worst-case complexity of a GLR parser is the same of an Earley one ($O(n^3)$), though it may have better performance with the best case of deterministic grammars. A GLR parser is also harder to build than an Earley one.

Summary

With this great (or at least large) article we hope to have solved most of the doubts about parsing terms and algorithms. What the terms means and why to pick a certain algorithm instead of another one. We have not just explained them, but we have given a few pointers for common issues with parsing programming languages.

For reason of space we could not provide a detailed explanation of all parsing algorithms. So we have also provided a few links to get a deeper understanding of the algorithms: the theory behind them and an explanation of how they work. If you are specifically interested in building things like compiler and interpreters, you can read another of our articles to find resources to create programming languages.

If you are just interested in parsing you may want to read Parsing Techniques, a book that is as comprehensive as it is expensive. It obviously goes much more in depth what we could, but it also cover less used parsing algorithms.

What's next?

If you have questions or doubts you can always look for more content on my blog.

In general you can find articles related to Language Engineering:

https://tomassetti.me/category/language-engineering

If you cannot find an answers write to me and I will try my best to help you.

You will find me at federico@tomassetti.me .