LINGI2132

Languages and Translators

Emilyen Laffineur

Teacher :
Nicolas Laurent

Assistant:
Mathieu Jadin & Alexandre Dubray

Ecole Polytechnique de Louvain
Universtié catholique de Louvain
Belgium
Scholar year : 2020-2021

Contents

Chapter 1

Introduction

This class is about compiler. The goal of compiler is to implement a programming language. It take source code and make it executable (or pass it to a program that execute it). We may think that language like Python or JavaScript does not use compilers however they use it as well! Even if they are dynamical. Note that this class is not about programming paradigms, it's truly about implementing languages in general.

We will learn about compilers and their parts, allowing us to understand how languages works. We will also learn some useful patterns (Domain Specific Language (DSL), trees, interpreter pattern).

Chapter 2

Compilation Pipeline

Here is the classical compiler pipeline :

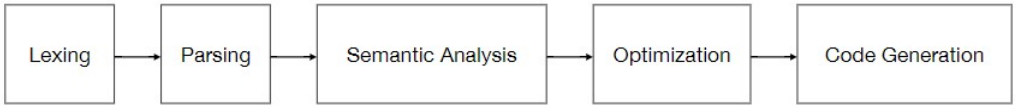

Figure 2.1: Compiler pipeline

We will go through each step.

2.1 Lexing

Definition 2.1.1 (Lexing)**.** It's a lexical analysis, tokenization, scanning. It take the text and produce tokens (= "words"). e.g : foo = bar + 42 will produce |foo| = |bar| + |42|.

2.2 Parsing

Definition 2.2.1 (Parsing)**.** It take the output of lexing (tokens) and extract it's structure (by producing an Abstract Syntax Tree (AST)). It also catches syntax errors (like missing parenthesis around if in Java). Other kind of errors (like something that is not defined by the language) is not a syntax error, thus, the parser cannot catch it.

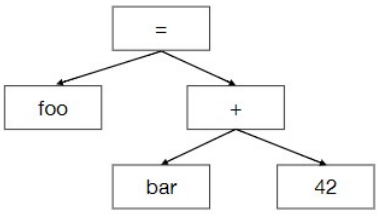

Figure 2.2: Abstract Syntax Tree

Note that lexing is optional while parsing (called scanner-less parser) for others parsing is actually lexing AND parsing.

2.3 Semantic Analysis

It consists of a series of check that is perform on the AST, two mains are Name binding et type checking.

2.3.1 Type checking

Definition 2.3.1 (Type checking)**.** Verify that the type passed is what is expected.

```
1 void print(String str)
2 print(42)
```

Definition 2.3.2 (Type inference)**.** Let the compiler guess the type of the variable given the context.

Note that type checking is inference. When :

```
1 String x = "foobar"
```

is used, it first need to guess what is "foobar", then check the consistency with the type.

Sometimes, we need to know the return value's type of a function to perform the check. That's what Name Binding is.

2.3.2 Name Binding

Definition 2.3.3 (Name binding)**.** Allow us to infer the type of a variable given the return type of a function.

Note that it becomes much more complex with OOP.

2.3.3 Rest

It also performs flow check (missing return for example), access control (public, private, etc.).

2.4 Optimization

Actually, the pipeline we saw in figure 2.1 it's only the old school C compiler. New C compiler, do optimization on the produced machine code (LLVM, Low Level Virtual Machine)! Do not get confuse, it does not use a virtual machine at all... LLVM generate IR (intermediate representation), optimize it and then generate machine code.

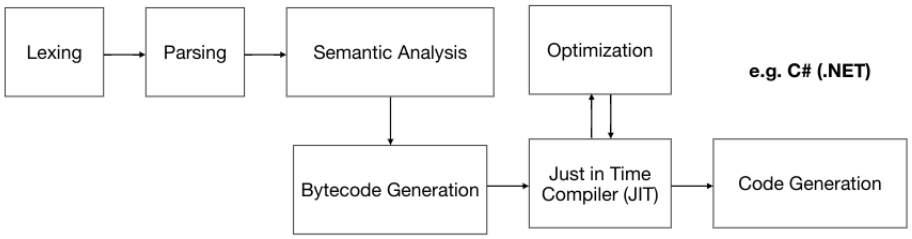

Figure 2.3: C# pipeline

This one generate bytecode (Java name), that use usable but that won't be directly use by the machine. After the generation of the bytecode, it is passed to a Just In Time Compiler (JIT).

What is done until JIT is done at compilation time, JIT and the following is done while running the program!

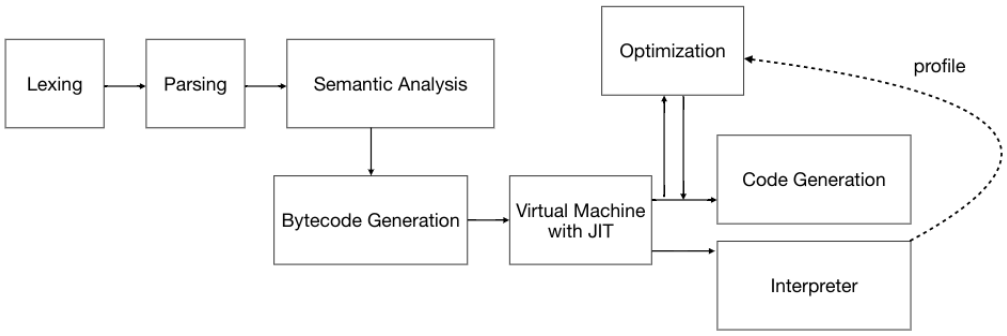

Figure 2.4: Java and Python pipeline

That's the architecture that most recent languages uses. The bytecode is passed to a VM executing a JIT. This VM can generate machine code (code generation) or pass it to in interpreter which will run a runtime profile that is much better for optimization! That the reason why Java can be as fast (or faster) the C. Pypy (Python), Java, TruffleRuby use this kind of architecture!

We could also not optimize it! We could directly use the Tree-Walk interpreter, Ruby before 1.9 did that. CPython (standard one), and Ruby generate the bytecode and run it in a VM.

2.4.1 Tree-Walk Interpreter

Let's take the following program :

```
1 var foo = 42 + 52
2 print(foo)
```

The AST is the following :

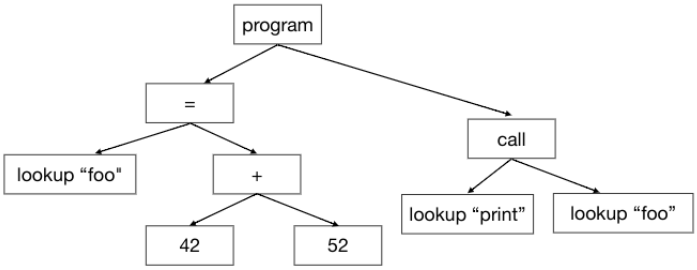

Figure 2.5: Resulting AST

(Note that print has is own AST). We have the scope, containing the value of foo saved in the store and the definition of print. The procedure in order to execute is recursive, it will lookup for foo, perform the addition, store it. Lookup at the print definition, lookup for foo and foo will be passed as a parameter to print.

2.5 Code Generation

It could be machine or bytecode generation. Example of Machine Code (e.g. x64) :

```
1 int square(int num) {
2     return num*num
3 }
```

Becomes :

```
square(int):
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rpb−4], edi
    mov eax, DWORD PTR [rpb−4]
    imul eax, eax
    pop rbp
    ret
With optimization it becomes :
    mov eax, edi
    imul eax, edi
    ret
```

While bytecode generation is:

```
public class Hello {
    int square(int num) {
        return num∗num
    }
}
```

Which becomes (stacked base instead of registry based as before):

```
public class Hello {
    public <init>()V
      L0 LINENUMBER 1 L0
       ALOAD 0
       INVOKESPECIAL java/lang/Object.<init> ()V
       RETURN
      L1
       LOCALVARIABLE this LHello; L0 L1 0
       MAXSTACK = 1
       MAXLOCALS = 1
      square(I)I
       L0
         LINENUMBER 3 L0
         ILOAD 1
         ILOAD 1
         IMUL
         IRETURN
       L1
         LOCALVARIABLE this LHello; L0 L1 0
         LOCALVARIABLE num I L0 L1 1
         MAXSTACK = 2
         MAXLOCALS = 2}
```

2.6 Optimizations

As we have seen, we can optimize on tree or on target code. They are two types of optimization that are the base of everything :

- Inlining : pulling a function in another one
- Partial evaluation : propagating known information (e.g constant-folding)

Some other exists like loop unrolling, etc.

2.6.1 Inlining

Take the following program :

```
1 var foo = false
2 int a(){
3     if (foo) return somethingLongAndBoring()
4     else return b(true)+1
5 }
6 int b(boolean bar) {
7     return bar ? 42 : 52
8 }
```

If we optimize the function a. It we online somethingLongAndBoring and b. However, if we do this and do not enter the condition, this could be bad for cache locality. Indeed, it will lead to a cache miss, with a lot of code, the time to retrieve the actual code will be huge. However, we know that foo is always false so we can get rid of the first condition! Then we can inline b. Thanks to the inlining of b, then constant folding it will lead on only this :

```
1 int a(){
2     return 43;
3 }
```

Which is much more efficient! Thanks to that we can see that there is a complementary between constant folding and inlining.

Chapter 3

Formal Grammars

3.1 Language vs Grammar

Definition 3.1.1 (Grammar)**.** The (formal) definition (*description*) of a language

Definition 3.1.2 (Language)**.** A (potentially infinite) *set* of sentences, in programming language, a sentence is a source file, REPL expression, etc.

Definition 3.1.3 (Alphabet)**.** Composed of tokens, lexemes.

Let's take an example which is JSON. The language is all valid JSON expression. Sentences : e.g.

```
{
    "version": 17,
    "bundles": [
        { "name" : "org.graalvm.component.installer.Bundle" },
        { "name" : "org.graalvm.component.installer.commands.Bundle" },
        { "name" : "org.graalvm.component.installer.remote.Bundle" },
        { "name" : "org.graalvm.component.installer.os.Bundle" }
    ]
}
```

Figure 3.1: JSON Sentence

Grammar :

```
VALUE   ::= STRINGLIT / NUMBER / OBJECT / ARRAY
OBJECT  ::= "{" (PAIR ("," PAIR)* )? "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE ("," VALUE)* )? "}"
```

Alphabet (Tokens)

Figure 3.2: JSON Grammar

3.2 Grammar notation

3.2.1 PEG

Parsing Expression Grammar. The "?" denotes optional, "*" means 0 or more times.

```
VALUE   ::= STRINGLIT / NUMBER / OBJECT / ARRAY
OBJECT  ::= "{" (PAIR ("," PAIR)* )? "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE ("," VALUE)* )? "}"
```

Figure 3.3: PEG notation

PEG is a grammar formalism and a notation.

Definition 3.2.1 (Formalism)**.** Mathematical system to define the language (set of sentences)

Definition 3.2.2 (Notation)**.** A way to denote a grammar to be interpreted by the formalism.

3.2.2 EBNF and BNF

Note that (E)BNF is a notation for CFG (Context-Free Grammars).

EBNF

Extended Backus-Naur Form. "[]" means optional, "{}" means 1 or more times, so in order to make "0 or more" with have to combine symbols.

```
VALUE   ::= STRINGLIT | NUMBER | OBJECT | ARRAY
OBJECT  ::= "{" [ PAIR [{ “,” PAIR }] ] "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE [{ " ," VALUE }] ] "}"
```

Figure 3.4: EBNF notation

BNF

Backus-Naur Form. It is at the base of EBNF, it does not contain "{}" or "[]", so it uses recursion and a lot more rules. Note that ϵ is the mathematical notation for "nothing"

```
VALUE   ::= STRINGLIT | NUMBER | OBJECT | ARRAY
OBJECT  ::= "{" PAIRS? "}"
PAIRS?  ::= ε | PAIRS
PAIRS   ::= PAIR TPAIRS
TPAIRS  ::= ε | "," PAIRS
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "[" VALUES? "]"
VALUES? ::= ε | VALUES
VALUES  ::= VALUE TVALUES
TVALUES ::= ε | "," VALUES
```

Figure 3.5: BNF notation

3.2.3 Summary

CFG and PEG are different formalism. In some case, PEG and CFG can be the same (like in JSON) because it is a very simple definition. By it is not true in general. Sometimes, we can use PEG notations like +, *, ? in CFGs (coming from REGEx, which are related to CFGs.). Actually we can use any notation that we want, we just need to define it!

Chapter 4

Parsers

Definition 4.0.1 (Parser)**.** A parser is a program that :

- Accepts/rejects input a sentence in the language (recognizer)
- Extracts a(n) (abstract) syntax tree

4.1 Parsing Tools

A parsing tool lets you create parsers. We can denote two kinds : Generator an Library. Note that is just a generalization. The main distinction to retain is the generation vs interpretation.

Parsing tools are often called "parsers", it is acceptable, but incorrect, parsing tools create and/or run parsers.

Parsing tools are compilers!

- Language = grammar notation
- Code generation (parser generators) or interpretation (parsing libraries) (depending on the used tool)
- Domain Specific Language (DSL) (depending on how we define it. DSL is opposed to Turing-Complete language like Java, Ruby, Python, etc. A DSL could be XML, JSON, Grammar, SQL, etc.)

4.1.1 Parsing Generator

Note : in the following figure, all solid boxes denote programs.

Definition 4.1.1 (Parsing Generator)**.** A Parsing Generator is :

- Outputs a parser program (typically source code)
- Typically a command line tool
- Typically uses own grammar language

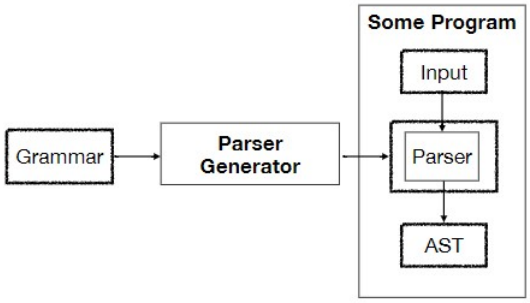

Figure 4.1: Parser Generator

4.1.2 Parsing Library

Definition 4.1.2 (Parsing Library)**.** A Parsing Library is :

- Let us "interpret" a grammar
- Grammar typically defined with a DSL.

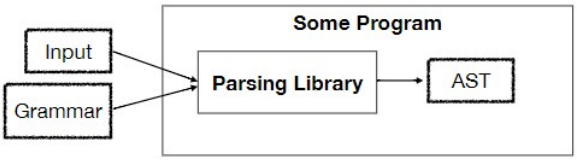

Figure 4.2: Parser Library

Note that input AND grammar could be in the "Some Program" box.

4.2 (Abstract) Syntax Tree

4.2.1 Syntax Tree

Remember the notation of figure 3.2 and 3.1. To make it easier for Syntax Trees we will use the figure 3.5. Doing so, we can extract the following ST :

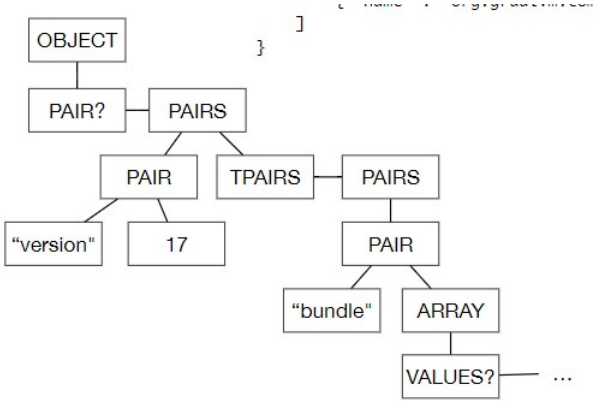

Figure 4.3: Syntax Tree

Syntax Tree is the following : for each phrase in the sentence we follow the grammar. It is the only thing we can do!

4.2.2 Abstract Syntax Tree

Here is the same Syntax Tree but abstracted :

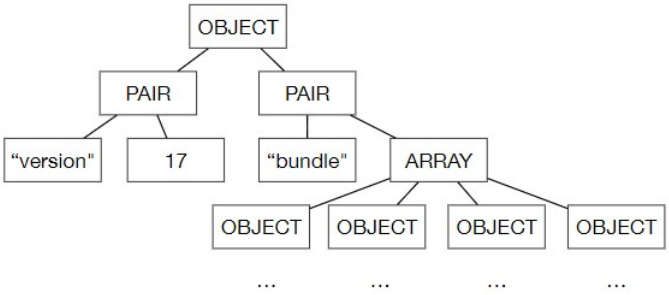

Figure 4.4: Abstract Syntax Tree

As we can see, it is much more clear than the classical Syntax Tree.

4.3 Summary 1

Parsing Tools that allow us to use grammar definition like PEG or EBNF (with more symbols that denotes more explicit stuffs) will generally produce better Syntax Tree than the others. However AST let us decide precisely what we want as node/leaves in the tree. Let's take an example using Java :

```
1 list.forEach(x -> System.out.println(x));
2 list.reduce((x,y) -> x + y);
```

As we can see, the difference is that with 1 identifier we do not need parenthesis. That must be take into account when generating the Syntax Tree. It would lead to the following Syntax Trees :

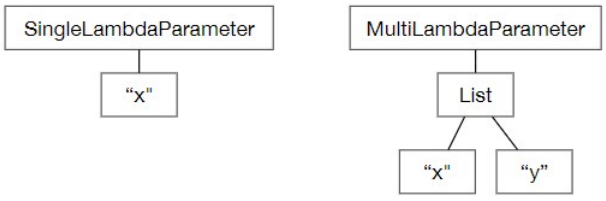

Figure 4.5: First ST

However, as we have seen, parsing is just the start of the pipeline! We have to do a lot of work with it (check for errors, generate bytecode, etc.). With this kind of Syntax Tree, we would have to deal with different kind of node for the same lambda principle. Using AST we can have the following :

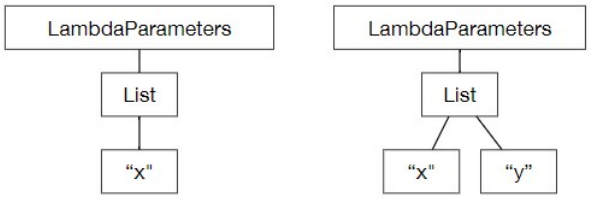

Figure 4.6: Second AST

Which allow us to use the same code to deal with both!

4.4 Coding a parser

General principles :

- One function per symbol (non-terminals & terminals);
- Model in the input as globally accessible character array;
- The current input position is globally accessible;
- Calling a function = attempting to parse the symbol it denotes at the current input position;
- A parsing function returns :
  - true if the symbol was matched, updating the input position past the matched input.
  - false if they failed to match, the input remains unchanged.

See parser.java file. Note that, in order to be sure our implementation really work, we can use the debugger and check for the pos cursor at the end, or to go further we could also inspect the parse tree if we have build one.

As we can see, the parser we wrote is verbose and cumbersome. Also it does not have any bells or whistles, if it fails it just fails, it does not tell us where it fails or why. What we have done is implementing a PEG semantics for the grammar (hint, a* a is empty). Backtracking in PEG is to reset the counter before the start of the sequence. CFG also have a backtracking technique but it is not the same.

We can also add AST to our parser. In order to do that in Java, we just need to add a Dequeue and modify only a few methods. See parser_ast.java for that.

4.5 Parser Combinator

The parser we have implemented from PEG grammar. As we have seen, this parsers have a lot of duplication (each "expression" is handled the same way). The idea to solve that is to build an AST where each node is an "expression" and interpret it.

The idea, is that each combinator is a node/parsing expression in the AST. See file Combinators.java. Using combinators allow us to cut down on verbosity & code duplication (across all grammars and within a specific grammar). However it still have some drawbacks :

- Harder to debug
- It's slower because of megamorphism (we'll come back later on this)

Note that theses issues can be eliminated by using combinators for code generation. Good frameworks (like Autumn) will mitigate usability issues.

Chapter 5

PEG & CFG Semantics

5.1 Recap

Definition 5.1.1 (Grammar)**.** The (formal) definition of a language

Definition 5.1.2 (Language)**.** A (potentially infinite) set of sentences

Definition 5.1.3 (Parser)**.** Recognizes a sentence in the language + extract a syntax tree

Definition 5.1.4 (Parsing Tool)**.** Generate and/or runs parsers

Two types of formalism :

- CFG (Context-Free-Grammar)
- PEF (Parsing Expression Grammar)

Notations :

- (E)BNF (usually for CFG)
- PEG Notation (usually for PEG)

Definition 5.1.5 (Non-Terminal)**.** Things that we define

Definition 5.1.6 (Terminals)**.** Tokens, strings, etc. that cannot be extended further.

5.2 Context-Free Grammars

Usually a CFG is defined as a tuple of 4 components Grammar $= (N, \Sigma, P, S)$:

- N: Non-Terminals
- Σ: Alphabet (Terminals)
- P: Production Rules $(P : N \to (\Sigma \cup N)*)$
- Starting Symbol $(S \in N)$

In order to have a CFG from a "full" grammar, we first need to replace all syntactic sugars by the complete recursive rules, then eliminate all choices by introducing as many rules as we have choices.

5.2.1 Semantics Derivation

- The language defined by a CFG is the set of all sentences that can be derived from its rules
- Start from the start symbol, replace it by the right-hand side of one its production
- At each step, replace a non-terminal from the current string symbol by its definition (until no non-terminals are left in the string)
- Any terminal, the order does not matter

Note that by doing that, we define the language, not a parsing algorithm and we're doing it in a generative way grammar → sentences.

Also, if we would have used characters derivation instead of tokens we would have need to continue to derive which would of potentially lead to an infinite derivation. So, we can say that for a sentence to be part of a language in need to be derivable however we cannot describe the all derivation table because in many case, it would be infinite.

5.3 PEG Semantics

We can see in the slide (9 of PEG - CFG semantics's pdf) the use of a top-down recursive descent parsers that produce production rules (with lookahead operator as an exception).

5.4 PEG vs CFG

5.4.1 Semantics

CFG are generative (their semantics is given by constructing the language set by the grammar through derivation). On the other hand PEG are recognition based : a sentence is in the language defined by a PEG grammar only if it is recognized by the language recognizer. In order to formalize PEG grammar we need to formalize the recognizer for the grammar.

These two approaches are very different in the mathematical point of view. Also, CFG is easier to defined in the mathematical language, on the other hand the PEG is very much related to the practice.

5.4.2 PEG vs CFG

- CFG has unordered choices, while derivating we can pick any non-terminal we want
- PEG has ordered choices, the first matching is the "correct" one

5.4.3 The differences

Unordered and ordered has a consequence for parsing algorithm. PEG does not test anything after a fail (prefix capture) if the parse failed while CFG can make an other choice!

PEG: Single parse Rule

Once a choice have been made, we never visit it again. As repetition can be desugared to choice it has an impact on them too. Repetition are greedy. (A ::= a* a is empty as a* will consume everything).

Backtracking

Let's take the following grammar :

- A ::= B y z
- B ::= v | x y | x
- input: "xyz"

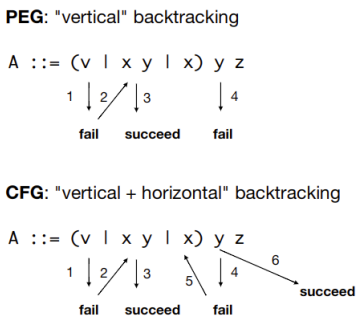

Figure 5.1: PEG vs CFG backtracking

This is not the real working way, be it does illustrate the principle. Indeed doing that would lead to exponential time complexity. CFG has horizontal backtracking thanks to is possibility to backtrack on x after trying y.

Ambiguity

It seems that CFG are always better because of what we have seen before. However there is a flip-side : ambiguity. By construction PEG cannot suffer from ambiguity.

As a remainder :

- Prefix capture (The rule B will consume all y)
  - A ::= B y z
  - B ::= x y | z
- Ambiguity (B can be xy or x, C can be z or yz):
  - A ::= B C
  - B ::= x y | x
  - C ::= z | yz

Ambiguity make the AST creation harder and also has impact on the parser performances.

Performances

The best parsing algorithm for CFG are $\mathcal{O}(n^3)$, deterministic parts of the grammar run in $\mathcal{O}(n)$ (most of useful grammar are deterministic).

For PEGs, the regular algorithm is exponential (in theory). Still, it is almost impossible to write an $\mathcal{O}(x^n)$ exponential parser. However, one often use operation is very inefficient : infix.

An example can be this one :

- S ::= P '+' S | P '-' S | P

- P ::= N '*' P | N '/' P | N
- N ::= [0-9]+

(Note that PEG does not allow left recursion by definition.) Let's assume we have a parseN function, if we try to parse "42", parse N will be called 9 times! 3 times for P, 3 times for S, 3x3=9. In general : $\mathcal{O}((P+1)^L)$ times with L : precedence levels (2 here), P : operators at each level (3 here).

A solution for that can be found, for example in Autumn we can write :

```
rule P = left_expression()
                .operand(N)
                .operand('*')
                .operand('/'); //Same for S however, operand is P in S, for precedence
```

This rewrite the grammar as :

- S ::= P ('+' S | '-' S)*
- P ::= N ('*' P | '/' P)*
- N ::= [0-9]+

P and N will be only called once! Performances are good thanks to that! However we can note that the parse tree won't be nice (not a problem with Autumn as we build an AST explicitly (we give a function to create the nodes)). If we were not using Autumn, we should create the parse tree like that and then, rewrite it

Packrat Parsers The single parser rule make memoization very easy! Packrat Parser are PEG parser with memoization. However, some practical experiments have been done in Java shown it is slower. Unless maybe if your language is very slow, or unless your infix expressions are improperly implemented.

Expressivness

- Some PEGs definition cannot be defined with CFG (A ::= $a^n b^n c^n$ for any same n we want any a b c)
- Some CFGs definition cannot be defined with PEGs (A ::= a A a | b A b | a | b | ϵ)
- Traditional PEG can't use left recursion (use repetition or Autumn to solve that, as they are the only two big case where we need left recursion)

PEG : Lookahead operators

- &<expression>, succeeds if the expression succeeds but does not consume any input
- !<expression>, succeeds if the expression fails, does not consume any input
- In theory, &X == !!X (not true in Autumn)

PEG misc

PEGs are often use in scannerless parsing (without lexer), ordered and lookahead are very useful at lexical level. PEG allow us to define "reserved works", hence lexing can still be advantageous.

PEGs are easy to extends thanks to recursive descent (with new combinators).

5.5 Summary

PEG :

- Intentional language = sentences recognized
- Similar to handwritten top down recursive descent parsers
- Desugared to CFG + lookahead
- Single parse rule
- Suffer from prefix capture
- Vertical backtracking
- Deterministic
- Ordered
- Potentially exponential, in practice largely linear (careful with infix)

CFG :

- Extensional language = set of sentences obtained by derivation
- Suffer from ambiguity
- Vertical and horizontal backtracking
- Non deterministic
- Unordered
- $\mathcal{O}(n^3)$ but often linear in practice

In the end, both formalisms are good enough, the real difference is tooling, what is available in the language we want to use, ease of use, features, performances, etc.

Chapter 6

Chomsky's Hierarchy

6.1 The hierarchy

Chomsky which is a famous linguist also known for his political engagement has defined a hierarchy of grammar :

- Type 3: Regular grammars (regular expressions, single non terminal at the end)
  - $P : N \to \Sigma * N$
- Type 2: Context-Free grammars (non terminal string of symbols)
  - $P : N \to (\Sigma \cup N)*$
- Type 1: Context-Sensitive grammars (alpha and beta represent the context)
  - $P : \alpha N \beta \to \alpha \gamma \beta$ where $(\alpha, \beta, \gamma \in (\Sigma \cup N)*)$
- Type 0: Unrestricted grammars
  - $P : (\Sigma \cup N)+ \to (\Sigma \cup N)*$

6.2 Regular vs CFGs

In regular we have a final non-terminal makes regular languages capable of expression repetition and optionality. Regular cannot express nesting.

6.3 Type 0 and 1

Let's be honnest, they are pretty much useless. Still, the principle behind context sensitivity is useful! Indeed, is func((T) * x) a multiplication or a cast? However CSGs are bad for that (not like Autumn).

6.4 Automaton mapping

Definition 6.4.1 (Automaton)**.** Formalisation of an abstract machine

A nice thing with the hierarchy is that it can be mapped to automaton :

- Type 3 (Regular grammars) : Deterministic Finite Automaton

- Type 2 (Context-Free grammar) : Nondeterministic (can have multiple transition) Pushdown (use a stack, a transition can push on it or look at it to take a decision) Automaton
- Type 1 (Context-Sensitive grammar) : Linear Bounded Automaton (Finite-tape Turing Machine)
- Type 0 (Unrestricted grammar) : Turing machine

Chapter 7

Lexing with regular expression

Chapter 8

Pumping lemma

Chapter 9

LL and LR algorithms

9.1 LL

9.2 LR

Chapter 10

Semantic Analysis

10.1 The Hindley-Milner Type System

10.2 Using Uranium