The Open University of Israel

Department of Mathematics and Computer Science

a project report on

Flue

*a **F**luent **L**ibrary for **U**tilizing **E**BNFs*

submitted by
Noam Rotem

The Open University

*under the advisory of Yossi Gil,
Dept. Comp. Sc., the Technion*

(2020-2021)

Acknowledgement

Many thanks to my wife and daughters who tolerated the fact that my nights and weekends are not theirs but this work's…

I wish to thank Yossi for his guidance, his responsiveness and his patience with me along the way…

Contents

1 Introduction **1**
- 1.1 About . . . 1
- 1.2 Code . . . 1
- 1.3 Basic Terminology . . . 1
  - 1.3.1 DSL . . . 2
  - 1.3.2 Groovy . . . 2
  - 1.3.3 Library . . . 2
  - 1.3.4 Grammar . . . 3
  - 1.3.5 EBNF . . . 3
  - 1.3.6 AST . . . 3

2 Grammar Specification **4**
- 2.1 The EBNF class . . . 4
  - 2.1.1 Order of rules . . . 5
- 2.2 Non terminals . . . 5
  - 2.2.1 Non terminals in Flue . . . 5
  - 2.2.2 Non terminal Labels . . . 5
- 2.3 Terminals . . . 6
  - 2.3.1 Deterministic Terminals . . . 6
  - 2.3.2 Pattern Terminals . . . 6
  - 2.3.3 Epsilon (ε) . . . 6
- 2.4 Production Rules . . . 7
- 2.5 Root non-terminals . . . 7
  - 2.5.1 Explicit Root Non terminals . . . 8
  - 2.5.2 Implicit Root non-terminals . . . 8
- 2.6 Expressions . . . 9
  - 2.6.1 Sequences . . . 9
  - 2.6.2 Choices (Alternatives) . . . 10
  - 2.6.3 Options . . . 10
  - 2.6.4 Repetitions . . . 11
  - 2.6.5 Precedence . . . 12

3 How Flue is Implemented **13**
- 3.1 Challenges . . . 13
- 3.2 Rule Capturing . . . 13
- 3.3 Expression Statement . . . 14
- 3.4 Operator Overloading . . . 14

3.5 How Can Expressions “Do Something”? . . . 14
3.6 A Timing Problem . . . 15
3.7 Thread Safety . . . 16
3.8 Precedence . . . 16
3.8.1 Options and Repetitions . . . 17
3.8.2 Terminals . . . 17
3.8.3 Overloading Methods of System Types . . . 17
3.8.4 The +{} Syntax . . . 18

4 EBNF API **19**
4.1 clone() . . . 19
4.2 `nonTerminalGraph()` . . . 19
4.3 entryPoints() . . . 20
4.4 cycles() . . . 20
4.5 inlined() . . . 21
4.6 nullable() . . . 21
4.7 first() . . . 22
4.8 follow() . . . 23

5 Further Tools **26**
5.1 Rule API . . . 26
5.1.1 rules and ruleMap . . . 26
5.1.2 Rule . . . 26
5.1.3 Expression . . . 26
5.1.4 Terminal . . . 27
5.1.5 Non terminal . . . 27
5.1.6 Optional . . . 27
5.1.7 Repeated . . . 27
5.1.8 Then . . . 27
5.1.9 Or . . . 27
5.2 Visitors . . . 27
5.3 ExPath . . . 28

6 The Java Grammar **32**
6.1 Syntax . . . 32
6.2 Cycles . . . 34
6.3 Inlining . . . 34
6.4 Optimization . . . 34

A The follow() Method Code **35**

B Flue Grammar for EBNF **40**

List of Algorithms

1 Finding Grammar's non-terminal Root 9
2 Graph of non-terminals 19
3 Finding Entry Points 20
4 Finding Cycles 20
5 Inlining Grammar 21
6 Evaluating Nullability 22
7 Evaluating Nullability 23
8 FOLLOW Closure Calculation 25

Abstract

Flue is a Groovy library that allows specifying language grammatical rules by a fluent, elegant and clear EBNF-like DSL. When the grammar creation is run, it builds a data structure (AST) that represents the language, and allows studying and manipulating the language rules. Finally, Flue provides a collection of algorithms and tools that make optimizing, investigating and reviewing grammars easy. To test the capabilities of Flue, this work also included the implementation of the Java grammar with Flue, and a demonstration of some of Flue's tools and techniques on this grammar.

Keywords: *Fluent API, groovy, EBNF, Embedded DSL, grammar, language, AST, Java*

Chapter 1

Introduction

1.1 About

Flue is a Groovy library for fluent specification of language grammars. This report will summarize the development work, the challenges in implementing it, the results and the way to use Flue.

Chapter 2 will explain how to use Flue for specifying grammatical rules.

Chapter 3 will then allow a peep into the challenges in building Flue, and tell the story of how Flue overcomes all those obstacles.

Chapter 5 will review and explain the EBNF API that provides handy tools and algorithms for the grammar developer.

Chapter 5 will complete the review with the API for using rules and additional tools that allow working with the rules and expressions created in Flue.

And finally, chapter 6 will touch the work done with the Java grammar, to try out Flue and its tools.

1.2 Code

Flue's source code is in Github: `https://github.com/metormaon/flue`

The code is well documented and tested.

1.3 Basic Terminology

From the abstract of this paper:

Flue is a Groovy library that allows specifying language grammatical rules by a fluent, elegant and clear EBNF-like DSL. When the grammar creation is run, it builds a data structure (AST) that represents the language, and allows studying and manipulating the language rules.

This section will explain the terminology found in this description, and hopefully make it clear.

Note: the definitions in this chapter aim to clarity and readability. There may be more accurate or generic definitions of the terms, but this chapter focuses on the definitions that serve the understanding of this project.

1.3.1 DSL

A domain specific language (DSL) is a computer language for a specific purpose (typically an application domain). As opposed to a general-purpose language like Java or C++, a DSL draws its commands and capabilities from a singular world from which the problems to be solved by this DSL arrive. A dedicated language for game programming makes a good example of a DSL. It focuses on a specific domain (games), and draws its commands and statements from the world of game programming, rather from the generic world of computer languages.

Often, DSLs are embedded within generic programming languages. For example, a DSL for programming games, can be offered as a library of Java, that uses the Java syntax to create a set of functions or other language tools that "speak the language of games". It will probably offer commands for game setting creation, character movement, character-object interactions etc.

1.3.2 Groovy

Apache Groovy is a Java-syntax compatible, object-oriented programming language that runs on Java JVMs. It could be used both as a programming language and a scripting language.

Groovy and Java are quite similar, but Groovy is more dynamic and flexible. It was selected for this project thanks to its DSL-friendliness and its dynamic nature.

DSL-friendliness means that Groovy is flexible enough to allow creating DSLs that look natural, and are relatively free of programming symbols—dots, parentheses etc. In other words, Groovy allows statements that resemble fluent English sentences. The flexibility is not perfect (Scala, for example, is much more flexible), but compared to Java, one can do a lot to make Groovy fluent and English-like.

The dynamic nature of Groovy deserves documents of its own, but specifically for this project, Groovy's operator overloading and system-class extensions were used (among a few more dynamic capabilities), and the fact that the Groovy IDEs adapt their syntax highlighting and apply the "Intellisense" technology to support these dynamic changes immediately when they are punched into the code, made it possible to create this fluent EBNF-like DSL.

1.3.3 Library

A programming library is a packaged set of classes, tools, and programming aids that may be linked to a program and be used by it programmatically.

As opposed to a framework that controls the developer's flow and runs the developer's code, a library provides tools that a developer may run or invoke from his code from within his program flow.

1.3.4 Grammar

Grammar is a level of language syntax that focuses on phrases, expressions, statements etc. A programming language's grammar is a specification of how programs in this language may look like.

1.3.5 EBNF

EBNF—Extended Backus–Naur Form—is a grammar for formally defining grammars.

There is no standard EBNF, and the web is full of variants and implementations of it. However, the concept is quite the same among most of the variants.

EBNF specifies terminals, non-terminals, concatenations, alternations, choices, repetitions and more.

Note: EBNF specification by EBNF (and coded by Flue) can be found in Appendix B

1.3.6 AST

AST—Abstract Syntax Tree—is a tree that represents the syntax of some text that complies with a programming language's specifications.

In this project, the Expression composite tree of the rule API (Section 5.1) is an AST of the EBNF-like language specified by the developer using Flue.

Chapter 2

Grammar Specification

2.1 The EBNF class

The EBNF class encapsulates and represents a grammar. Instantiating an EBNF object is done by specifying a block of production rules:

```
EBNF grammar = ebnf {
    A >> B | C
    B >> "hello" & [C] & "world"
    C >> "."
}
```

Listing 2.1: Specifying Grammar

This piece of code creates an EBNF instance (to be referred to as *grammar*) that represents a grammar of three production rules. The rules define the non-terminals `A`, `B` and `C`:

- `A` is defined as `B` or `C`.
- `B` is defined as a sequence of the token `"hello"` then an optional `C`, then the terminal `"world"`.
- `C` is defined as the terminal `"."`

Flue does not parse, but just for the completion, the rules of *grammar* allow the following phrases:

- hello world
- hello . world
- .

Note: phrases are expected to comply with rule `A`, which is the *implicit root* of the grammar. See section 2.5.2 below.

2.1.1 Order of rules

Order of rules is not important—a grammar is a **set** of rules. Nevertheless, Flue remembers the order by which the rules were specified, so a printout of the grammar, for example, will be deterministic.

2.2 Non terminals

`A`, `B` and `C` in code listing 2.1 are *non-terminals*. *Non terminals* are variables that represent a piece of grammar, and may be used within other pieces of grammar.

Non terminals are expected to be defined within the grammar by rules (see later). In other words, each non-terminal `A` in grammar x is expected to have at least one production rule in grammar x that explicitly specifies its meaning.

2.2.1 Non terminals in Flue

In Flue, *non-terminals* are elements of an *Enum* that must implement the **NonTerminal** trait:

```
enum V implements NonTerminal { A, B, C, D, E, F, G }
```

Listing 2.2: Defining non-terminals

Note: it is recommended to statically import the non-terminals Enum such that the non-terminals `A`, `B`, etc., are recognized by the IDE by their short names, and need not to be prefixed by the Enum name: V.A, V.B, etc.,

```
import static il.ac.openu.flue.test.V.*
```

Listing 2.3: Static Import of non-terminal Enum

2.2.2 Non terminal Labels

The Enum elements representing the non-terminals in Flue, have a **label** that represents them in grammar printouts. By default, the label is taken from the Enum element name. In other words:

```
A.label == "A"
```

However, it is possible to set the label explicitly, if needed. For example:

```
enum V implements NonTerminal { A, B, C, UNDERSCORE("_")}
```

Listing 2.4: Setting non-terminal's Label

When `UNDERSCORE` is to be expressed as a string, it will be expressed as _ and not as `UNDERSCORE`. Since _ is not a valid Enum element name in Groovy, this name/label redundancy is required in this hypothetical implementation.

2.3 Terminals

Terminals are the leaves of the syntax tree—they are elements that explicitly specify their appearance. In code listing 2.1, `"hello"`, `"world"` and `"."` are Terminals.

Terminals usually reflect identifiers, literals, keywords, symbols and operators. Keywords, symbols and operators are explicit at definition time, i.e, their string is deterministic. Literals and identifiers, on the other way, are often defined as a pattern, regular expression, on strings rather than a deterministic string.

2.3.1 Deterministic Terminals

Deterministic terminals are specified in Flue as a quoted string (single or double quotes):

- ”hello”
- ’+=’
- ”else”

2.3.2 Pattern Terminals

Pattern terminals are specified in Flue as a quoted string prefixed by a tilde:

- `~"float|double|int"`
- `~"[_a-zA-Z][_a-zA-Z0-9]+"`

Note: When the quoted pattern is prefixed with a tilde, its content has to comply with the specifications.

2.3.3 Epsilon (ε)

For various implementations, a special Terminal is needed for representing an empty state (empty expression; empty token).

For that purpose, the Epsilon Terminal is defined by Flue, and may be used by grammar rules (see later) this way:

`A >> ε`

Epsilon is declared as a static Terminal in the *Terminal* class of Flue:

```
public static final Terminal ε = new
Terminal("ε")
```

Listing 2.5: Definition of Epsilon in the Terminal Class

Note: it is highly recommended to statically import epsilon:

```
import static il.ac.openu.flue.model.rule.Terminal.ε
```

Listing 2.6: Static Import of Epsilon

Otherwise referring to epsilons can be done by their full name:

```
A >> Terminal.ε
```

2.4 Production Rules

Production rules (usually referred in this document just as **rules**) are grammatical phrases that specify the meaning of *Non Terminals*. This is a rule from code listing 2.1:

```
B >> "hello" & [C] & "world"
```

The rule structure consists of two elements: a non-terminal and an expression. Between these elements separates an arrow symbol (`>>`):

```
non-terminal >> expression
```

Note: The left part and the right part of a production rule are often referred to in academic literature as LHS and RHS, respectively.

Note: Expressions will be explained in section 2.6.

A rule states that the non-terminal on its left stands for the expression on its right, and can be replaced by it when it appears in expressions of the grammar. In other words, the rules in code listing 2.1 define `A` as an expression on `B` and `C`; `B` is defined as an expression on `C`; `C` is defined by an expression that consists of a Terminal only.

Grammars may specify more than one rule for a *non-terminal*. It would mean that the non terminal has more than one expression that may stand for it. Such multiple rules may be grouped into a single rule that defines the non terminal as expression 1 OR expression 2 OR expression 3 etc.

2.5 Root non-terminals

Grammars in Flue must have a *Root non-terminal*. The root is a non terminal which is not referred to by the right side of any rule of the grammar, i.e., no other non terminal is defined by it in the grammar.

Note: The rule(s) that has the Root non-terminal on its left side is often referred to as the Start rule in the academic literature.

Flue can represent a grammar as a directed graph of non-terminals, such that a node is a non terminal and an edge is a dependency of this non terminal in other non terminals by which it is defined in the grammar. In such a graph, a Root non-terminal is a node that has no edges coming in.

Root non-terminals have a practical meaning—they are being used in various methods of the *EBNF* class and could be used by programmers in various use cases. For example, when calculating the FIRST and FOLLOW of a grammar (see sections 4.7, 4.8).

2.5.1 Explicit Root Non terminals

Root non-terminals of grammars may be explicitly specified when the grammar is created. An explicit root non-terminal is simply provided by the programmer as an argument to the EBNF specification:

```
EBNF grammar = ebnf(A) {
    A >> B | C
    B >> "hello" & [C] & "world"
    C >> "."
}
```

Listing 2.7: Grammar with Explicit Root non-terminal

In this example, `A` is passed as a parameter to the EBNF block, declaring that `A` is the Root non-terminal.

An explicit non-terminal declaration is not checked by Flue, assuming the programmer knows what she is asking for. Therefore, the following code is accepted by Flue:

```
EBNF x = ebnf(C) {
    A >> B | C
    B >> "hello" & [C] & "world"
    C >> "."
}

EBNF y = ebnf(D) {
    A >> B | C
    B >> "hello" & [C] & "world"
    C >> "."
}
```

Listing 2.8: Suspicious Explicit Root Declarations

In grammar `x`, `C` is explicitly specified as the Root, which may look wrong. But the programmer may have an idea why this is relevant and it will become clear when the grammar is used.

Same goes with `D` in grammar y. It is not even part of the grammar, but it might become part later, so it is left in the responsibility of the programmer.

2.5.2 Implicit Root non-terminals

When a Root non-terminal is not explicitly provided as an argument to the EBNF specification, it is assumed automatically by Flue.

Flue would assume the root with the following algorithm:

Algorithm 1 Finding Grammar's non-terminal Root

```
1: $e \leftarrow entryPoints()$
2: if $|e| \geq 1$ then
3:     return e[0]
4: else
5:     fail
6: end if
```

- (1) Populate e with all the entry points of the grammar, using the `entryPoint` algorithm 4
- (2), (3) If there are entry points, return the first one, arbitrarily.
- (4), (5) If there are no entry points, fail root finding.

2.6 Expressions

Expressions are what is found on the right side of production rules. They are specifications of a composite (meaning—a syntax tree), that consists of Terminals, non-terminals, and compounds of them.

The very basic expressions are made of a single `Terminal` or a single Non `Terminal`. Here are some rules based on such basic expressions:

- `A >> B`
- `C >> "hello"`

`B` is an expression made of a non-terminal. `"hello"` is an expression made of a `Terminal`.

But expressions may also be compound. There are four compound structures that may be used in expressions:

- Sequences
- Choices
- Options
- Repetitions

These compounds are discussed in the following sections.

Elements in an expression may be either simple—Terminals or non-terminals, or compound. Elements in compounds, may also be simple or compound.

2.6.1 Sequences

A sequence is a specification of expression elements that are expected to appear one after the other.

Flue offers the ampersand operator (&) for sequence specification. Here are some examples of rules that use sequences in their expressions:

1. `A >> "hello" & "world"`
2. `B >> C & D`
3. `C >> D & E & F`
4. `D >> E & "+" & D & "."`
5. `E >> (A | [B & {C}]) & F`

- (1) defines `A` as the `Terminal` `"hello"` and then the `Terminal` ”world”.
- (2) defines `B` as the non-terminal `C`, and then the non-terminal `D`.
- (3) specifies `C` as a sequence of `D`, then `E`, then `F`.
- (4) is a sequence of `E`, then the `Terminal` `"+"`, then `D`, then a period token.
- (5) defines `E` as a sequence of a compound statement and then `F`. The compound statement is a choice between `A` and an optional sequence of `B` and a repeating `C`. Choices, Options and Repetitions will be explained in the following sections.

2.6.2 Choices (Alternatives)

A choice is the suggestion of a few alternatives from which only one should be selected. For example, when a statement in the language represented by a Flue grammar is being parsed, a parser may expect to find x or y or z, as a Choice in the grammar allows.

Some rule examples using Choices:

1. `A >> "a" | B`
2. `B >> A & ("+" | D & ".")`

- (1) specifies `A` as a choice between the token ”a” and the non-terminal `B`
- (2) defines `B` as `A`, then a choice between two options: the plus sign, or a `D` with a period at its end. The need in parentheses and the precedence of operators will be discuss in section 2.6.5.

2.6.3 Options

Elements in Flue expressions may be marked as Optional by surrounding them with square brackets. For a potential parser it would probably mean that an element is expected but is not mandatory when parsing statements by the grammar.

Examples:

- `["x"]` is an optional `"x"` Terminal
- `[B]` is an optional `B` non-terminal
- `A & [B]` is a compound of `A` and then, optionally, also `B`.
- `[A | B]` is an optional choice between `A` and `B`.

2.6.4 Repetitions

Elements in Flue expressions may be marked as repeated, by surrounding them with curly brackets. By default, the repetition is of zero or more times. A potential parser that follows grammar specified by Flue may expect an element to be skipped, to appear once or to appear multiple times, when it is marked as a repetition.

A repetition may be treated as a sequence of zero or more occurrences of the repeated element.

Examples:

- `{"x"}` means zero or more `"x"` Terminals
- `{B}` is a repetition of the `B` non-terminal zero or more times
- `A | {B}` is a compound of `A` and then zero or more `B` occurrences.
- `{A & B}` is a repletion of `A` and then `B`. It could be zero times `A` then `B`, or one time `A` then `B`, or `A` then `B` then again `A` then `B` etc.

One or More

The default form of repetitions reflects zero-or-more repetitions. Often, one needs a way to specify a repetition of one-or-more. It is possible to specify a one or more repetition of `B` this way:

```
A >> B & {B}
```

Flue offers the +{} syntactic sugar to simplify this requirement:

```
A >> +{B}
```

Prefixing a repetition with a unary plus operator marks it as a one-or-more case. The `+{}` syntax may of course apply also to compound repetitions.

List Separators

With repetitions, it is possible to specify lists of one or more elements:

```
A >> B & {"," & B}
```

`A` is defined as a single `B`, or `B` comma `B`, or `B` comma `B` comma `B` etc.

Flue answers the list-with-separator requirement with another syntactic sugar – the /”” syntax:

```
A >> B & {B}/","
```

The slash operator may trail repetitions, to specify the separator between sequential repetitions of the repeating element.

`{B}/","` means therefore: either no occurrence of `B`, or a single `B`, or `B` comma `B`, or `B` comma `B` comma `B` etc.

Using the `{}/""` syntax combined with the + syntax, a list of one or more becomes simple and clean:

```
A >> +{B}/","
```

This would mean: `A` is defined as a one or more occurrences of `B`, with a comma between consecutive `B` occurrences.

Note: the separator must be a Terminal.

2.6.5 Precedence

Sequences in Flue use the & operator, while Choices use the | operator. Groovy gives & a higher precedence than |. Therefore, the following rule means a choice between `A` and the sequence of `B` and then `C`:

```
A >> A | B & C
```

If we want a decision between `A` and `B`, and only then a `C`, Groovy's precedence will require us to put parentheses in the right place:

```
A >> (A | B) & C
```

Flue will respect the parentheses.

Chapter 3

How Flue is Implemented

3.1 Challenges

Flue is designed and implemented to simplify the task of specifying grammars for Groovy implementations. As such, it requires a simple yet powerful syntax with very high readability and general elegance.

On one hand, syntax simplicity may be achieved by adhering to the EBNF specification and keeping rule specification as clean as possible for the user. On the other hand, the syntax should also comply with Groovy's syntax. After all, it is Groovy.

Most of the challenges in implementing the grammar syntax, therefore, came from the tension between Groovy and EBNF.

Flue is a library. As such, its classes and APIs are in use in runtime by implementations. However, one aspect of Flue should be available in development time, i.e.—while writing the rules: the IDE (e.g., IntelliJ) should accept the grammar syntax and not mark it as an error.

The following sections describe some of these challenges and the way Flue overcomes them.

3.2 Rule Capturing

Flue uses the following syntax for specifying grammars:

```
EBNF grammar = ebnf {
    A >> B | C
    B >> "hello" & [C] & "world"
    C >> "."
}
```

Listing 3.1: Sample Grammar Code (repeated from previous chapter)

`ebnf()` is a static method of the EBNF class (that is statically imported to avoid writing

EBNF.`ebnf{})`. It accepts a single argument: a Groovy Closure that is specified as a block of code within curly brackets.

A Closure is an anonymous method body. It may accept parameters and return a value, but the FLue user does not expect to handle parameters and return a value when creating a grammar. The user expects to merely specify, line by line, the production rules of the grammar.

A production rule should therefore be a code statement in valid Groovy.

```
A >> B | C
```

3.3 Expression Statement

The rule above doesn't look like valid Groovy, but in fact it is. The operator `>>` is a shift-right operator in Groovy. The operator `|` means or. `A`, `B` and `C` are Enum elements, i.e.—objects of the Enum type (which are eventually non-terminals). Thus, the rule is parsed by Groovy as:

non-terminal-`A` shift-right non-terminal-`B` or non-terminal-`C`

To begin with, this is an expression. Groovy allows expressions to appear where statements are expected (this is called Expression Statement), but such an expression seems like a calculation of some value, rather than an action. In particular, no assignment is visible. Luckily, while a typical Java IDE would at least warn against an expression that ”does nothing”, in Groovy expressions are valid statements, because the last expression in a code block is the value to return from the code block (the return keyword is optional). Closures may return a value; therefore, the IDE accepts expressions within it.

3.4 Operator Overloading

The next challenge is the meaning of these operations against the objects. How can we shift-right `A` by `B` bits? What does it mean for non-terminals?

Luckily again, Groovy allows operator overloading, and thanks to its non-strict type checking, it even allows setting the return value to be of any type we need.

Note: this type-flexibility with operator overloading is undocumented in Groovy. It required empirical tests to be discovered and mastered. There's no guarantee therefore, that all IDEs will respect it, forever. Currently, IntelliJ allows such flexible overloading, and does not warn once the overloading method is in place—not about the overloading method itself, and not about the statements that use it.

3.5 How Can Expressions “Do Something”?

Expressions are usually immutable entities that calculate a value and “do nothing” by themselves. However, the operator overloading of Groovy allows side effects to take place, while the expression is calculated.

Which overloaded operator should "do the magic" of storing the specified rule in the EBNF instance? Apparently, the method of the operator that every rule specification uses, and uses only once, is the method of the `>>` operator:

```
1 // Inaccurate code. Will be corrected in the following sections
2 trait NonTerminal /*...*/ {
3     Rule rightShift(Expression e) { grammar.add(new Rule(this, e)) }
4     /*...*/
5 }
```

Listing 3.2: First (Inaccurate) Attempt on rightShift

- (line 2) The `rightShift` operator belongs to the `NonTerminal` class (it's a trait and not a class to allow `Enums` to implement it).
- (line 3) `rightShift` operates on the non-terminal on its left, and accepts the expression to its right as a parameter. Then it creates a rule that gets both sides as construction arguments. The rule is then added to the grammar instance.

3.6 A Timing Problem

The above code, though, has a major flaw. Let's refer back to the grammar definition:

```
EBNF grammar = ebnf {
    A >> B | C
    B >> "hello" & [C] & "world"
    C >> "."
}
```

The *grammar* instance is not accessible from the Closure block, because it is declared outside of the block. Furthermore, it is the call to `ebnf {}` that instantiates the grammar instance, so *grammar* does not even exist when the closure is executed.

Therefore, the rules should be added to a static object that is accessible from within the Closure, and after the entire Closure is executed, the `ebnf{}` method should take the rules from this static object and instantiate with it the grammar object.

```
1 trait NonTerminal implements Expression /*...*/ {
2     Rule rightShift(Expression e) { EBNF.add(new Rule(this, e)) }
3     /*...*/
4 }
```

Listing 3.3: Method `rightShift` of `NonTerminal`

(line 3) add() is a static method of the EBNF class, and therefore is accessible from the Closure.

3.7 Thread Safety

EBNF.add() may be used in parallel by multiple threads. If add is a static method that operates on a shared object, the rules of the parallel constructions will get mixed up. How can that be prevented?

Flue's work around the challenge is by using a `ThreadLocal`:

```
class EBNF {
    private static final ThreadLocal<EBNF> context
        = new ThreadLocal<>()

    static Rule add(Rule r) {
        /*..*/
        context.get().rules += r
        /*..*/
    }

    /*..*/
}
```

Listing 3.4: Using `ThreadLocal` in EBNF

The `EBNF.ebnf{}` method initiates the `ThreadLocal` grammar object of the current thread. After the rule block is executed and all the rules are processed, it cleans the `ThreadLocal` so it is usable for this thread for the next grammar.

Note: Flue implementations are expected to use a single thread along the block that builds the grammar. This is reasonable to assume, but—as learned during this project in the hard way—this is not guaranteed. The Groovy testing framework Spock does not use a single thread throughout the rule-building Closure, and therefore was not used for testing grammars in this project. Other frameworks may do the same.

3.8 Precedence

Back to the rule processing, the challenges are yet to be over. The following statement needs to overcome precedence challenges as well:

`A >> B & C`

One may expect that everything to the right side of the `>>` operation will be parsed as an expression, and passed as an expression to the `rightShift` method of `NonTerminal` (code listing 3.3). However, the precedence of `rightShift` is higher than of the & operator. Therefore, the parsing order of statement is done as if this was the statement: `(A >> B) & C`

`A >> B` is processed first. The operator & with the argument C will run as a method of whatever `A >> B` returns. Therefore, Flue makes `A >> B` return the generated rule.

What is left to process is therefore:

rule `& C`

& is implemented as the overloaded operator `and()` of the class Rule:

```
class Rule {
    Rule and(Expression e) { definition &= e; this }
    /*...*/
}
```

Listing 3.5: Rule's and() method

The rule object holds a `NonTerminal` (its left side) and an `Expression` (its right side). The operator and() adds what comes after the `&` to that expression.

To sum it up, the rule `A >> B & C & D & E` will evaluate `A >> B` and return a rule object. The rule object's and() method will be called with `C` as an argument. Then it will be called again with `D`. Then with `E`.

3.8.1 Options and Repetitions

Flue expects Options in square brackets. For Groovy, the expression [B] is an instantiation of a new list, with `B` as a member. ["abcd"] is a list of Strings. [~"abcd"] is a list of Patterns.

It means that EBNF.add() and many other methods need to have an overloaded method that accept List<?>, should check what the member type is, and other complications.

Luckily, all of this was proven to be achievable in Groovy (though sometimes in tricky ways).

As for Repetitions, {B} is a Closure that returns `B`. To support it, EBNF.add() and many other methods need an overloaded version that accepts Closure<?>.

3.8.2 Terminals

Deterministic Terminals and Pattern Terminals are Strings and Patterns respectively in Flue.

So EBNF.add() and many other methods need an overloaded version that accepts a String and a version that accepts a Pattern.

3.8.3 Overloading Methods of System Types

Consider the following rule:

`A >> B | [C] & "break"`

As before, `A >> B` is evaluated first by a call to `NonTerminal`'s `rightShift` method that accepts `B` and returns a rule instance.

But now there's a change from the previous examples. Since `&` precedes |, the next thing to get evaluated is `([C] & "break")`. This evaluation calls the and() method of List, with a String argument!

List is not a class. It's an interface. But further than that—it's a system interface.

Slightly changing the rule makes it even more scary –

```
A >> B | "break" & [B]
```

Now `("break" & [B])` is an expected overriding of String's and() method. And String is a **final** system type.

Luckily again—this is Groovy and not Java, and Groovy is more flexible than Java. It allows introducing an Extension class, that allows overriding system class methods.

An extension class requires special registration in Groovy's configuration, and extra care in development, but eventually, Flue manages to override `and(String, List)`, `and(List, String)`, `and(String, String)` etc. Same goes with `or(...)`.

3.8.4 The +{} Syntax

As a final example of the challenges in Flue, it is interesting to look into the one-or-more paradigm expressed by the +{} syntax.

```
A >> +{B & C}
```

The one-or-more attribute of a Repetition is captured in a field of the class Repeat. This class is instantiated in various places, and in these places it is important to know if it's a +{} or a regular {}.

Flue's solution is the class `AtLeastOneClosure` that extends Closure. The above mentioned `Extension` class specifies the method `positive(Closure<?>)` that overrides the unary plus operator when it prefixes a Closure. It creates an instance of `AtLeastOneClosure`, and returns it as a Closure. So +{} returns the closure within the curly brackets, but the fact that it's not an ordinary Closure but an `AtLeastOneClosure`, helps raising the `atLeastOne` flag in `Repeated` when it is instantiated.

Chapter 4

EBNF API

Once an object of the EBNF class is instantiated and representing a grammar, it is possible to use its capabilities and apply actions on the grammar.

The next sections will describe EBNF object's methods and the various grammar tools they provide.

4.1 clone()

grammar.clone() deep copies the grammar. It uses a *visitor* for traversing the grammar's expressions and instantiating copies of expressions.

Terminals and non-terminals which follow the FlyWeight[1] design pattern, do not need to get copied.

4.2 `nonTerminalGraph()`

`grammar.nonTerminalGraph()` creates a directed graph from a grammar. Each node is a non-terminal in the grammar. Each edge from node x to node y means that non-terminal x is defined using non-terminal y (i.e., x is dependent on y).

The graph is returned as a map from `NonTerminal` to a set of `NonTerminals` it is dependent of. The algorithm is straightforward:

Algorithm 2 Graph of non-terminals

```
(1) For each rule in the grammar that specifies non-terminal x by Expression y:
    (a) Visit y with a non-terminals finding Visitor (about visitors—see section 5.2)
    (b) Add all the found non-terminals to the Set mapped from x.
```

The graph is useful for investigating the relationship between non-terminals in the grammar, for finding entry points, cycles etc.

[1]https://en.wikipedia.org/wiki/Flyweight_pattern

4.3 entryPoints()

`grammar.entryPoints()` is a method for retrieving all the potential root `NonTerminals` of a grammar. An entry point is a `NonTerminal` that no `NonTerminal` (including itself) is dependent on in its definition.

For efficiency, if a `nonTerminalGraph` was already generated, it can be passed to the method. Otherwise, the method will generate such a graph.

Algorithm 3 Finding Entry Points

```
(1) Make a set of the graph nodes, i.e., a set of all the non-terminals that have a
    definition rule.
(2) Make a set of all the non-terminals that have edges coming into them, i.e., non-
    terminals that are used in definitions
(3) Subtract (2) from (1) to get all the defined non-terminals that are not involved in
    definitions.
```

Entry points are used in cycle searches, and other traversals of the grammar graph that need to start with entry nodes.

4.4 cycles()

grammar.cycles(), as its name implies, finds all the cycles in the graph. A cycle could be of one non-terminal—if it is defined by itself, or of multiple non-terminals, if they point to one another, until one points back to the first one.

The returned cycles are unique—If A points to `B` and `B` points to `C`, the returned cycle may be [A, B, C], but [B, C, A] and [C, A, B] will not be returned, since they describe the same cycle. Function cycles() uses the DFS algorithm to find cycles, and once a loop is discovered, it uses an additional stack to map the cycle members.

Algorithm 4 Finding Cycles

```
(1) Prepare a stack. Prepare a way to keep for each non-terminal node—its status.
    Initialize all statuses to Not Visited. Prepare a non-terminal graph.
(2) For each rule in the grammar, refer to its non-terminal. If its status is Not Visited:
    (a) Put it in the stack
    (b) Change its status to In Stack
    (c) Let's assume the top of the stack (without popping) is x.
    (d) For each non-terminal y pointed by x in graph:
         (i) If the status of y is In Stack—report a cycle
        (ii) If the status of y is Not Visited:
             1 Push y to the stack
             2 Set y's status to In Stack
             3 Repeat from (2).c
    (e) Set x's status to Visited.
    (f) Pop the stack.
```

Reporting a cycle that starts at x means using a temporary stack and pushing into it pops from the first stack until x is found in the stack. Then, starting pushing back into the stack pops from the temp stack, while reporting the node.

Cycle finding is used when inlining grammars, and for various analysis purposes.

4.5 inlined()

grammar.inlined() returns an optimized copy the grammar by eliminating redundant rules that can be skipped.

For example, in the following set of rules:

```
A >> B & C

B >> [D]

C >> "x"

D >> ~"a*"
```

The following optimization actions may be takes:

(1) `B` can be optimized into `B >> [~"a*"]` and `D` may be deleted

(2) `A` may be optimized into `A >> [~"a*"] & C` and `B` may be deleted

(3) `A` may be optimized into `A >> [~"a*"] & "x"` and `C` may be deleted

Algorithm 5 Inlining Grammar

```
(0) Prepare a non-terminal graph; prepare a reversed graph—from a non-terminal to
    all the non-terminals that are dependent on it; prepare a set of grammar entry
    points.
(1) While there are changes:
    (a) Find in the graph node x that has no edges (= a non-terminal that depends
        on no non-terminals)
    (b) For each node y in the dependency graph that is dependent on x:
          (i) Replace in y the reference to x with the definition of x
    (c) Delete x from the grammar unless it's an entry point
```

The method returns a clone of the original grammar and does not change it.

4.6 nullable()

grammar.nullable() is a method that maps each non-terminal into the Boolean answer to the question—could it resolve to null? Null, in this context, means either an explicit epsilon, or an empty resolution.

Examples:

Rule	nullable() of `A`
`A >> "abc"`	false
`A >> [B]`	true
`A >> {B | "a"}`	true
`A >> +{B | "a"}`	false
`A >> B & [C]`	nullable() of B
`A >> "a" & [B]`	false
`A >> B | [C]`	true
`A >> "a" | B`	nullable() of B

Algorithm 6 Evaluating Nullability

```
(1) Prepare a map from non-terminal to Boolean nullability. Set all to be false.
(2) While there are still any changes to the map:
    (a) For each rule in the grammar:
        i  Use a dedicated Visitor to evaluate the rule definition's nullability. Visitor's
           logic:
           1 Terminal—false, unless the Terminal is epsilon
           2 non-terminal—whatever the nullability map says
           3 Optional—true
           4 Repeated—true, unless the atLeastOne flag is set to true
           5 Choice—true if any of the options is true
           6 Sequence—true if all the elements are true
        ii Set the evaluated nullability of the non-terminal in the nullability map.
```

Nullability is used for follow() calculation (see following sections) and may serve automata algorithms and more.

4.7 first()

grammar.first() maps each non-terminal to all the Terminals that may appear first in its definition.

```
EBNF grammar = ebnf {
    A >> B
    A >> "g"
    B >> "e" | C
    C >> "f"
    C >> A | ε
    D >> {C} & [E] & "m" & "m" | ["r"]
    E >> "q"
}
```

Listing 4.1: Grammar for FIRST Algorithm

first() mapping from non-terminal to Terminals:

```
A: ["e", "f", "g", " "]
B: ["e", "f", "g", " "]
C: ["f", "e", "g", " "]
D: ["f", "e", "g", " ", "q", "m", "r"]
E: ["q"]
```

Algorithm 7 Evaluating Nullability

```
(1) Prepare an empty map from non-terminal to a set of Terminals
(2) While there are still any changes to the map:
    (a) For each rule in the grammar:
        i Use a dedicated Visitor to evaluate its non-terminal's FIRST closure. Visitor's logic:
          1 Terminal—add the Terminal to the FIRST closure
          2 non-terminal—add the non-terminal's closure to the FIRST closure
          3 Optional—add epsilon as well as the closure of the child, to the FIRST closure.
          4 Repeated—add the closure if the child to the FIRST closure. If atLeastOne is not true, add also epsilon.
          5 Choice—add to the FIRST closure the closures of all the options.
          6 Sequence—add to the FIRST closure the closures of the child elements one by one. If a child's closure does not contain epsilon, do not continue to the next child.
        ii Add the calculated FIRST closure to the non-terminal's FIRST map entry.
```

first() is used for follow() calculations (see next), but may serve for automata algorithms, parsing algorithms etc.

4.8 follow()

grammar.follow() maps every non-terminal in the grammar to the set of Terminals that might appear right after it.

To support the algorithm, a special Terminal is defined in the EBNF class, to represent end-of-input—the emptiness after the last character in the language's statements:

```java
public static final Terminal ṩ = new Terminal("ṩ")
```

Listing 4.2: End-of-Input Symbol Declaration

Note: it is not exactly the dollar symbol. It's another Unicode character, since $ is not a valid constant name, and surprisingly, ṩ is.

The *FOLLOW closure* of a non-terminal is made of all the Terminals that might appear right after the resolution of the Non-Terminal. In other words – the potential next Terminals of the non-terminal. Calculating the FOLLOW closure uses the *NULLABLE table* and the *FIRST closure* of the grammar. For efficiency, it is possible to provide these data structures to the `follow()` method, if they were already calculated before (see sections 4.6 and 4.7 for nullable() and first() respectively).

The discovery loop runs while there are changes to the *FOLLOW closure* map. The map keeps accumulating additional values due to the recursive nature of the rules (in other words: the calculation process is a bootstrap).

Based on the NULLABLE and FIRST data, *visitors* will be able to answer specific questions.

The algorithm needs a visitor and not merely the nullable and first maps, because the maps tell the nullability or the FIRST closure of a non-terminal, but the algorithm needs to know also if a **sub-expression** is nullable, or what is the FIRST closure of a sub-expression. Not only of non-terminals.

```
EBNF grammar = ebnf(A) {
    A >> B & C
    C >> "+" & B & C | ε
    B >> D & E
    E >> "*" & D & E | ε
    D >> "(" & A & ")" | "id"
}
```

Listing 4.3: Grammar for follow() Evaluation

follow() mapping from non-terminal to Terminals:

```
A: ["$̣", ")"]
B: ["$̣", ")", "+"]
C: ["$̣", ")"]
D: ["$̣", ")", "+", "*"]
E: ["$̣", ")", "+"]
```

Algorithm 8 FOLLOW Closure Calculation

```
(1) Prepare an empty FOLLOW map from non-terminal to a set of Terminals
(2) While there are still any changes to the map:
    (a) For each rule in the grammar, apply 3 rules:
        (i) Case #1: S -> $ => FOLLOW(S) += $
            What follows the root, is $—end of input. If the rule's non-terminal is
            the root—add $ to its FOLLOW closure.
        (ii) Case #2: A -> $\alpha$B$\beta$ => FOLLOW(B) += FIRST($\beta$)
            Regardless of A, if B is followed by some $\beta$ in any rule, then the follow of
            B should include also the first of $\beta$
        (iii) Case #3: A -> $\alpha$B || (A -> $\alpha$B$\beta$ and NULLABLE($\beta$) => FOLLOW(B) += FOLLOW(A)
            In this situation, if a non-terminal ends a rule or what comes after it is
            nullable, it means that its FOLLOW should contain the FOLLOW of the
            non-terminal of that rule.
    (b) Add the calculated FOLLOW closure of the rule's non-terminal' FOLLOW
        closure.
```

Note: the algorithm uses the grammar's root. It could have just the same used entryPoints() set.

Note: The algorithm uses visitors to evaluate case #2 and case #3.

Note: The full code of follow() is enclosed in Appendix A, and may be also found here: `https://bit.ly/3lISwC8`

Chapter 5

Further Tools

5.1 Rule API

The rule API allows looking into grammar rules, running *Visitors*, manipulating grammars, etc.

Note: instead of manipulating the ruleset of a grammar, it is possible to clone the grammar and manipulate the clone.

5.1.1 rules and ruleMap

ebnf.rules is a list of rule objects that preserves the order of the rules specified when the grammar was created. It may include, naturally, rules for the same non-terminal duplicates, etc.

ebnf.ruleMap is a map from non-terminal to a set of rule objects that define it. It is generated for efficiency, but needs manual recreation whenever the rules list is changed.

5.1.2 Rule

The rule object is a POGO (Plain Old Groovy Object) that represents a rule. It refers to the rule's non-terminal and to the rule's right-side expression.

5.1.3 Expression

Expression is a trait (similar to interface) for all the elements of the rule composite.

All expression elements know how to print themselves (toString) and how to compare themselves (hash and equals).

Unary

`Unary` is an abstract class for unary expressions—Optional, Repeated. A descendant of this class will have a single child.

Multinary

`Multinary` is an abstract class for multinary expressions—Then, `Or`. A descendant of this class will have a list of ordered children.

5.1.4 Terminal

Terminal represents a deterministic string or a pattern within a grammar. It has a terminal field that holds the string, and a Boolean pattern field to indicate if the terminal is a pattern.

5.1.5 Non terminal

non-terminal represents a variable in the expression composite. It has to be defined as an Enum element.

Non-terminals have labels that by default are equal to their variable name, but may be overwritten in construction.

5.1.6 Optional

Optional is a `Unary` that marks its single child as non-mandatory in the syntax.

5.1.7 Repeated

Repeated is a `Unary` that marks its single child as repeating. The `atLeastOnce` Boolean flag tells if this is a zero-or-more repetition (when the flag is false) or one-or-more (when it's true).

`Repeated` has a separator field that might be null, but if not—it represents a separator Terminal between repetitions.

5.1.8 Then

Then is a `Multinary` that represents a sequence and has multiple children.

5.1.9 Or

`Or` is a `Multinary` that represents a choice and has multiple children.

5.2 Visitors

Traversing an expression composite could be done with loops and switches, but it is convenient to use Visitors for that.

Following the Visitor[1] design pattern, every Expression element has an accept(Visitor) method that expects an instance of the Visitor<T> interface, which is included in the rule API.

[1]https://en.wikipedia.org/wiki/Visitor_pattern

The generic T is the expected return value from the visit method. Here's an example from the code: the `clone()` method from the EBNF class:

```
/** Deep copies a grammar */
EBNF clone() {
    EBNF copy = new EBNF()
    copy.root = root
    copy.rules = rules.collect {Rule rule ->
        new Rule(rule.nonTerminal, rule.definition.accept(
                new Visitor<Expression>() {
            @Override Then visit(Then t) {
                new Then(t.children.collect {it.accept(this)})
            }

            @Override Or visit(Or o) {
                new Or(o.children.collect {it.accept(this)})
            }

            @Override Optional visit(Optional o) {
                new Optional(o.child.accept(this))
            }
            @Override Repeated visit(Repeated r) {
                new Repeated(r.child.accept(this),
                    r.separator, r.atLeastOne)
            }
            @Override Expression visit(NonTerminal nonTerminal) { nonTerminal }
            @Override Expression visit(Terminal terminal) { terminal }
        }))
    }
    /*...*/
    copy
}
```

Listing 5.1: clone() Implementation with a Visitor

The anonymous `Visitor<Expression>` deep copies a full expression. Each of its visit() methods handles the deep copy of a single Expression descendant. Each method returns an Expression (hence the Visitor's generic is Expression).

5.3 ExPath

`ExPath` holds a followable path to a sub-expression within an expression. It consists of `PathNodes`, which represent the nodes of a path in the expression composite tree. Each node points to the sub-expression it represents. If the sub-expression is a `Multinary` expression, and it is not the last node in the path, it also remembers the serial of the child through which the path continues.

For example, the path to `D` in the expression `B | (C & [D])`, is made of the following nodes:

(1) `Multinary` node that points to an `Or` expression, and keeps the position of the next node: 1 (0 is `B`; 1 is `C &` [D]).

(2) `Multinary` node that points to a Then expression, and keeps the position of the next node: 1 (0 is `C`; 1 is [D]).

(3) A `PathNode` that points to an `Optional` (i.e., to [D]).

(4) A `PathNode` that points to `D`.

An `ExPath` may also hold an info object that provides custom data. Most probably the info better-describes the last node—the target. If the path points to a Multinary, the info may specify, for example, the exact effective children of it (i.e - the reason why it was pointed at).

Often the info will provide all the additional data needed to transform the expression by the `ExPath`—peripheral or contextual info, without which the user of the `ExPath` will need to re-search for the referred-to sub-expression.

The method match() uses a closure parameter to find all the sub-expressions in a provided expression, that match some specified pattern or behavior. The function will traverse the expression composite tree, apply the closure on each node, and build an `ExPath` to each node that matched the criterion (i.e., for which the closure has evaluated to a non-null). The closure is expected to return an info object with extra information about the found location.

Note: even if no info is needed, match() will need some object to be used as a match indication, so one may use Boolean for the info. The value of that Boolean in this case will be ignored, and the match indication will be Boolean object or null.

The following example uses `ExPath`.match() to find all the terminals in the grammar, including terminals as separators of Repeated expressions.

```
static class TerminalInfo { // An info class
    Rule rule
    String terminal
    boolean isSeparator
}

EBNF grammar = ebnf { // Some grammar
    A >> B | C | +{D}/"."
    B >> ["a"] & "."
    C >> D & ".." & ["."]
    D >> "-"
}

List<ExPath<TerminalInfo>> matches = []

// Matching for each rule. Capturing the information.
grammar.rules.each { Rule r ->
    matches += ExPath<TerminalInfo>.match(r.definition) {
        switch (it) {
            case Terminal:
                return new TerminalInfo(
                    rule: r,
                    terminal: (it as Terminal).terminal
                )
            case Repeated:
                if ((it as Repeated).separator) {
                    return new TerminalInfo(
                        rule: r,
                        terminal: (it as Repeated).separator.terminal,
                        isSeparator: true
                    )
                } else return null
            default:
                null
        }
    }
}

// Using the matches to print the information:
matches.each { ExPath<TerminalInfo> path ->
    TerminalInfo info = path.info
    NonTerminal nonTerminal = path.info.rule.nonTerminal
    String token = (info.isSeparator? "/" : "") + info.terminal

    println("$nonTerminal: $token ($path)")
}
```

Listing 5.2: ExPath Example

Running the code above prints the following:

```
A: /.   (*/|2/{})
B: a    (*/&0/[]/'a')
B: .    (*/&1/'.')
C: ..   (*/&1/'..')
C: .    (*/&2/[]/'.')
D: -    (*/'-')
```

The `ExPaths` captured the information and also the path to where it was found.

Note: more examples of the usage of `ExPath` may be found in `ExPathUsageTest.groovy` within the project code.

Chapter 6

The Java Grammar

To experiment with Flue and demonstrate its capabilities, the Java EBNF was ported to Flue, and some experiments were applied to it.

6.1 Syntax

The Java grammar specification was taken from Oracle's documentation: `https://docs.oracle.com/javase/specs/jls/se16/html/jls-19.html` and was rewritten in Flue. In fact, it was a cyclic work: once the Java syntax was captured, it required more support from Flue. Once the support was added, it was tested on the Java syntax etc.

The full Java syntax may be found here: `https://github.com/metormaon/flue/blob/main/src/test/groovy/il/ac/openu/flue/JavaEbnf.groovy`

The syntax contains about 230 rules.

Here is a short quote:

```
EBNF java = ebnf {
    Identifier >> IdentifierChars

    IdentifierChars >> JavaLetter & { JavaLetterOrDigit }

    JavaLetter >> ~"[A-Za-z]"

    JavaLetterOrDigit >> ~"[A-Za-z0-9_]"

    TypeIdentifier >> Identifier

    UnqualifiedMethodIdentifier >> Identifier

    Literal >> IntegerLiteral
             | FloatingPointLiteral
             | BooleanLiteral
             | CharacterLiteral
             | StringLiteral
             | NullLiteral

    Type >> PrimitiveType | ReferenceType

    PrimitiveType >> { Annotation } & NumericType | { Annotation }
        & "boolean"

    NumericType >> IntegralType | FloatingPointType

    IntegralType >> "byte" | "short" | "int" | "long" | "char"

    FloatingPointType >> "float" | "double"

    ReferenceType >> ClassOrInterfaceType | TypeVariable | ArrayType

    ClassOrInterfaceType >> ClassType | InterfaceType

    ClassType >> { Annotation } & TypeIdentifier & [TypeArguments]
             | PackageName & "." & { Annotation } & TypeIdentifier
                 & [TypeArguments]
             | ClassOrInterfaceType & "." & { Annotation }
                 & TypeIdentifier & [TypeArguments]

    /*...*/
}
```

Listing 6.1: Java Grammar (Partial)

6.2 Cycles

The cycles() method was tested on the Java syntax and found 123 different cycles. A random sample of 20 cycles were manually tested and found to be correct.

6.3 Inlining

As part of the experiments with Flue and with the Java syntax, the syntax was inlined. The experiment helped testing the inlining method, but unfortunately, due to the highly cyclic nature of the Java syntax, inlining it had a very small effect on the grammar. Before inlining, there were 228 rules. After inlining—there were 210. Only 18 rules were eliminated by inlining.

Further investigation may find other reasons for that, but until then the assumption is that the high amount of cycles prevents inlining from being significant, since cycles cannot be inlined the way inlining works currently in Flue.

It is possible to enhance the inlining method such that it inlines also cycles to some extent, then to experiment again with the Java syntax.

6.4 Optimization

To experiment with `ExPath`, an optimization was performed on the Java rules.

There are some rules in the Java grammar that specify lists with separators this way:

`TypeArgumentList >> TypeArgument & { "," & TypeArgument }`

Here are some additional examples:

```
ModuleDirective >> "requires" & { RequiresModifier } & ModuleName & ";"
    | "exports" & PackageName & ["to" & ModuleName & { "," & ModuleName }] & ";"
    | "opens" & PackageName & ["to" & ModuleName & { "," & ModuleName }] & ";"
    | "uses" & TypeName & ";"
    | "provides" & TypeName & "with" & TypeName & { "," & TypeName } & ";"

SwitchLabel >> "case" & CaseConstant & { "," & CaseConstant }
    | "default"

LambdaParameterList >> LambdaParameter & { "," & LambdaParameter }
    | Identifier & { "," & Identifier }
```

Using `ExPath.match()`, a test program found all the occurrences of this pattern (21 occurrences), and replaced:

`A & {"," & A}` With

`+{A / ","}`

A demonstration of this `ExPath` may be found here: https://bit.ly/3oMsS1f

Appendix A

The follow() Method Code

The code of the method follow() of the EBNF class may be found also here: https://bit.ly/3lISwC8

```
/**
  Calculates the follow closure for each non-terminal in the rule set. The
  follow closure of a non-terminal is made of all the terminals that might
  appear after the resolution of the non-terminal. In other words---the
  potential next terminals of the non-terminal. Calculating the follow closure
  uses the nullable table and the first-closure of the non-terminals. If
  nullable and first are provided, the method will use them. Otherwise it will
  calculate them.  This option is meant for efficiency. The discovery loop runs
  while there are changes to the follow-closure map. The map keeps accumulating
  additional values due to the recursive nature of the rules (in other words:
  the calculation process is a bootstrap).
 */
Map<NonTerminal, Set<Terminal>> follow(
    Map<NonTerminal, Set<Terminal>> first = null,
    Map<NonTerminal, Boolean> nullable = null
  ) {
    //IF first and/or nullable are not provided as parameters, calculate them:
    if (first == null) {
        first = this.first()
    }

    if (nullable == null) {
        nullable = this.nullable()
    }

    /** Based on the nullable and first table, visitors will be able to answer
     * specific questions.  We will need a visitor and not merely the nullable
     * and first maps, because the maps tell us the nullability or the first of
     * a non-terminal, and we will need to know if a sub-expression is
     * nullable, or what is the first of a sub-expression. Not only of
     * non-terminals. */
    FirstVisitor firstVisitor = new FirstVisitor(first)

    NullableVisitor nullableVisitor = new NullableVisitor(nullable)

    Map<NonTerminal, Set<Terminal>> follow = [:]
```

```
def copyOfFollow

do {
    copyOfFollow = follow.clone()

    rules.forEach {rule ->
        // Rule #1: S -> $ // What follows the root, is $. End of input. So
        // add $ to the follow of the root.
        if (rule.nonTerminal == root) {
            follow.merge(root, [$].toSet(),
            (
              Set<Terminal> oldFirst,
              Set<Terminal> newFirst
            ) ->
              oldFirst + newFirst)
        }

        // Rule #2: A -> αBβ => FOLLOW(B) +=
        // FIRST(β) Regardless of A, if B is followed by some
        // β in any rule,
        // then the follow of B should include also the first of

        // For the follow calculation we will need a visitor that could
        // find all the non-terminals to which an expression resolves
        // (expression x resolves to non-terminal y if and only if one of
        // the resolution options of x is x -> y. Without any prefix or
        // suffix).  It will be used later in the code.

        Visitor<Set<NonTerminal>> nonTerminalResolver
                = new Visitor<Set<NonTerminal>>() {
            @Override
            Set<NonTerminal> visit(Then then) {
                // Collect all the children expressions that are NOT
                // nullable

                List<Expression> nonNullables = then.children.findAll{
                    !it.accept(nullableVisitor)
                }

                switch(nonNullables.size()) {
                    // If all the children are nullable, then each one may
                    // be resolvable to a non-terminal, while all the
                    // others are null. So try to resolve every child.
                    case 0:
                        return then.children.inject([].toSet(), { set, e ->
                            set + e.accept(this)
                        })

                    // One non nullable child? Try to resolve it to non
                    // terminal, assuming the others are null

                    case 1:
                        return nonNullables[0].accept(this)

                    // More than one non nullable child means that this
                    // Then expression cannot be resolved into a
                    // non-terminal, because there are more than one
                    // adjacent elements in
                    the resolution
                    default:
                        return []
                }
            }

            @Override Set<NonTerminal> visit(Or or) {
              or.children.inject([].toSet())
             {a, b -> a + b.accept(this)}
            }
            @Override Set<NonTerminal> visit(Optional optional) {
              optional.child.accept(this)
            }
            @Override Set<NonTerminal> visit(Repeated repeated) {
              repeated.child.accept(this)
            }
            @Override Set<NonTerminal> visit(NonTerminal nonTerminal) {
              [nonTerminal]
            }
            @Override Set<NonTerminal> visit(Terminal terminal) { [] }
        }

        // Another visitor we will need---a visitor that detects Rule #2
        // situations, i.e.: Bβ, and adds the first of β to
        // the follow of B.
        Visitor<Void> sequenceVisitor = new Visitor<Void>() {
            @Override
            Void visit(Then then) {
                // For each child in the Then expression
                then.children.init().eachWithIndex{ Expression e, int i -> {
                    // Find the set of non-terminals to which the child
                    // resolves

                    Set<NonTerminal> childNonTerminalResolution =
                      e.accept(nonTerminalResolver)

                    // For each non-terminal to which this child resolves,
                    // make a Then expression with all the children that
                    // come after it
                    childNonTerminalResolution.forEach{NonTerminal v ->
                        Expression restOfSequence =
                          new Then(then.children.drop(i+1))

                        // Calculate the first of the rest of the sequence,
                        // and add to the follow of the current child
                        follow.merge(
                          v,
                          restOfSequence.accept(firstVisitor) -
                            [Terminal.],
                          (
                            Set<Terminal> oldFirst,
                            Set<Terminal> newFirst
                          ) ->
                            oldFirst + newFirst
                        )
                    }
                }}
                null
```

```
  }

  @Override Void visit(Or or)
   { or.children.forEach{e -> e.accept(this)}
    null }

  @Override Void visit(Optional optional)
  { optional.child.accept(this)
   null }

  @Override
  Void visit(Repeated repeated) {
    // Repeated is zero or more times. So the follow of {A} is
    // the follow of A by itself, plus the follow of A in
    // AA.
    repeated.child.accept(this)
    new Then(repeated.child, repeated.child).accept(this)
    null
  }
  // NonTerminal and Terminal cannot be resolved into
  // $\alpha$B$\beta$?  sequences, so we use the base
  // implementation that returns null

}

// Apply the sequence visitor on the rule. It will detect and
// update follow closures.

rule.definition.accept(sequenceVisitor)

// Rule #3: A -> $\alpha$B || A -> αBβ &&
// NULLABLE(β) => FOLLOW(B) += FOLLOW(A) In this situation,
// if  a non-terminal ends a rule or what comes after it is
// nullable, it means that its follow should contain the follow of
// the non-terminal of that rule.
// This helping visitor finds all the non-terminals at the end of
// an expression

Visitor<Set<NonTerminal>> nonTerminalsAtEndVisitor
    = new Visitor<Set<NonTerminal>>() {
  @Override
  Set<NonTerminal> visit(Then then) {
    // In a Then expression, find all the non-terminals to
    // which the *last* child resolves
    Set<NonTerminal> nonTerminalsAtEnd =
      then.children.last().accept(this)

    // If the last child is nullable
    if (then.children.last().accept(nullableVisitor)) {
      if (then.children.size() > 2) {
        // Create a new Then expression with all but the
        // last child, and recurse over it
        nonTerminalsAtEnd += new Then(then.children.take(
          then.children.size()-1)).accept(this)
      } else {
        // One child. Recurse over it.
        nonTerminalsAtEnd += then.children[0].accept(this)
```

```
                    }
                }

                nonTerminalsAtEnd
            }

            @Override Set<NonTerminal> visit(Or or) {
                or.children.inject([].toSet())
                    {a, b -> a + b.accept(this)}
            }
            @Override Set<NonTerminal> visit(Optional optional) {
                optional.child.accept(this)
            }
            @Override Set<NonTerminal> visit(Repeated repeated) {
                repeated.child.accept(this)
            }
            @Override Set<NonTerminal> visit(NonTerminal nonTerminal) {
                [nonTerminal]
            }
            @Override Set<NonTerminal> visit(Terminal terminal) { [] }
        }

        // Use the visitor to find the non-terminals the current
        // rule ends with
        Set<NonTerminal> nonTerminalsAtEnd = rule.definition.accept(
            nonTerminalsAtEndVisitor)

        // For each of these non terminals, add to their follow the
        // current rule's non-terminal's follow,
        // as Rule #3 suggests
        nonTerminalsAtEnd.forEach { endingNonTerminal ->
            follow.merge(endingNonTerminal, follow.get(rule.nonTerminal,
                new HashSet<Terminal>()) - [Terminal.ε],
                    (Set<Terminal> oldFirst, Set<Terminal> newFirst)
                        -> oldFirst + newFirst)
        }
    }

} while (copyOfFollow != follow)

follow
}
```

Appendix B

Flue Grammar for EBNF

The following grammar specified by Flue, describes the grammar of EBNF itself.

Note: since there’s no single standard for EBNF, this is based on an unattributed random specification found on the web.

```
        EBNF ebnfGrammar = ebnf {
            lower >> ~"[a-z]"
            upper >> ~"[A-Z]"
            digit >> ~"[0-9]"
            special >> ~"[\\-_\"&’()*+,./:;<=>]"
            character >> lower | upper | digit | special
            string >> "\"" & character & {character} & "\""
            empty >> Terminal.ε
            lhs >> lower & {["_"] & lower}
            option >> "[" & rhs & "]"
            repetition >> "{" & rhs "}"
            sequence >> empty | {string | lhs | option | repetition}
            rhs >> sequence & { "|" & sequence}
            ebnf_rule >> lhs & ":=" & rhs
            ebnf_description >> {ebnf_rule}
        }
```

Bibliography

[1] Groovy Language Documentation, Version 3.0.9, viewed 12 July 2020, <http://docs.groovy-lang.org/docs/latest/html/documentation/>

[2] Oracle, Java Language Specifications, viewed 28 July 2020, <https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html>

[3] Roth O, Gil, Y, Fling—A Fluent API Generator (Github Repository), viewed 2 June 2020, <https://github.com/OriRoth/fling>

[4] Gil, Y, Tsoglin, Y, JAMOOS—A Domain-Specific Language for Language Processing, viewed 23 August 2020, <https://hrcak.srce.hr/file/69426>

[5] Hayun, A, Eyal, B, Zur-Lotan, L, LL(1) Parsers, viewed 7 March 2021, <https://www.cs.bgu.ac.il/ comp171/wiki.files/ps5.pdf>

[6] Wikipedia—EBNF, viewed 20 July 2021,

<https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form>