

Expression Generator

Bevilacqua Joey

Abstract

Expressions are a fundamental concept of programming languages: they combine values with functions and operators to produce other values. Programming languages define rules that determine how expressions are combined and evaluated to perform tasks. Teaching students how to interpret expressions is fundamental in order to ensure they can understand how a given programming language works. In order to teach and learn expressions, one has to actually write expressions, be it for explaining concepts, studying or making exercises to ensure that knowledge has been acquired. While the process of writing expressions may look like an easy task, it actually is a tedious and very error-prone process due to how a language was defined. We want to develop a tool to aid in the process of writing expressions by automating the whole process: to do so, our *Expression Generator* will be able to create expressions from language grammars definitions. It is possible to customize the expressions that will be generated with a number of constraints applied to the generator, allowing for the teaching of specific topics or scaling the complexity of the generated expression string. An extension for *Expression Tutor*, an online teaching platform by the *Lugano Computing Education research lab*, to enable the generation of expressions to aid in the creation of exercises has been developed and is planned to be deployed.

Advisor

Prof. Matthias Hauswirth

Assistant

Igor Moreno Santos

Advisor's approval (Prof. Matthias Hauswirth):

Date:

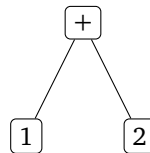
Contents

1	Introduction	2
2	Grammars	4
3	Expression Generator	5
3.1	Constraints	5
3.2	Usage	7
3.2.1	Command line interface	7
3.2.2	Web	7
4	Implementation & design	10
4.1	CFG	10
4.2	Parser generators	10
4.2.1	ANTLR	11
4.3	Generating expressions	12
4.4	Architecture	16
4.5	Code quality	17
5	Validation	19
5.1	Command line interface	19
5.2	Expression Tutor integration	21
6	Conclusion	23
7	Appendix A: example grammars	25
8	Bibliography	28

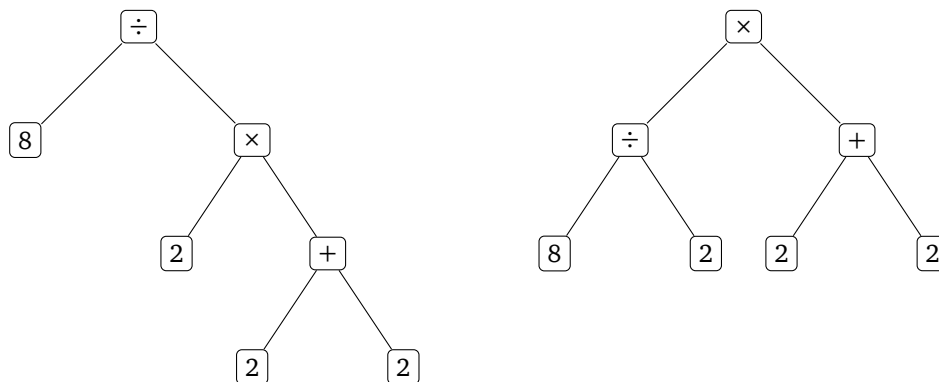
1 Introduction

Learning programming languages requires the understanding of one of their fundamental concepts: the expressions. Expressions are a combination of variables, constants, functions and operators that can yield new values. Multiple expressions can even be combined to make new more complex expressions. Each programming language defines a set of formal rules: these define how expressions can be composed and what syntax they must adhere to in order to be considered valid for the said programming language. This set of rules, commonly called *grammar* of a programming language does not define how the output of expressions is computed, they are only concerned with the how expressions are grouped together to form trees that represent code.

When teaching about them, one of the most commonly used examples is the simple $1+2$ mathematical expression. Expressions are usually represented with trees such as this one:



If we move to a slightly more *complicated* example, we see how operator precedence is fundamental: $8 \div 2 \times (2 + 2)$. Depending on the “priority” of each operator defined by the language grammar, the result of this simple expression will vary:



While this is a simple example, it shows the importance of the grammar definition of a language: the *right* expression tree yields the result 16 and other one yields 1.

Goal of the project

Being able to properly understand how expressions are evaluated is foundational if one wants to be able to write correct programs in a certain programming language. This is why the *Lugano Computing Education research lab* develops education-focused projects such as the Expression Tutor, a notional machine in which students can learn and verify their understanding of expressions.

In this online learning platform, it is possible to create exercises for students in which they are required to draw expression trees of a given expression string (or vice-versa, write the expression that corresponds to a given expression tree).

Writing expressions can be a very tedious and error-prone process: a minimal change or typographical error in the expression string can lead to a completely different expression tree, making the expression impossible to parse due to ambiguity or even render it impossible to parse in case any of the syntactic rules are not respected. That's why we want to provide means to automatically generate expressions that can be parsed by students.

In order to make this functionality worth using, we want to give the possibility to control what's going to be generated, starting from the language in which the expression will be generated. Our tool will have to be completely language-agnostic so that it can be used for teaching expressions in any imaginable language, in fact it'll even be possible to tailor existing languages definitions o accommodate for particular use cases or create subsets of programming languages for teaching purposes. For convenience, three grammars will be provided by the tool: a simple math grammar (Listing 8) to help understand the principles of expression parsing, a JSON document grammar (Listing 9) to see how it is possible to think of structured documents such as JSONs as expressions and the Racket Beginner Student's Language (BSL) grammar (Listing 10) as an example of a real functional programming language that can be easily parsed.

Moreover, we want to be able to define certain constraints on what's going to be generated: for instance, if we want the expression to involve arrays in order to verify that students acquired the required knowledge about them, we want to ensure that the generated expression will contain operations on arrays. Finally, we want to be able to control the complexity of all the generated expressions, so that exercises could be made easier, harder or equally difficult depending on the needs of the instructor.

Structure of this report

In this report, we'll first analyze grammars (Section 2), syntax definitions that can be used to define how expressions can be created in order to be considered valid with respect to a specific language. All the information about grammars presented here will then be useful to understand and make a better use of the Expression Generator tool, for which a detailed usage guide will be provided in the "Expression Generator" (Section 3). An implementation and detail section will follow and explain the specifications of the grammar parsing, expressions generator algorithm (Section 4), project architecture (Section 4.4) and how the quality of the code was kept in check during the development (Section 4.5). Finally we will perform a validation (Section 5), by showing how all the features work as intended and explained in the previous chapters.

2 Grammars

To be able to generate an expression, the tool needs to be fed a grammar: a grammar is a set of rules that define a formal language.

In our case we will deal with Context-Free-Grammars, a kind of grammars made of rules written in the form $R \rightarrow r$ where R is a non-terminal that acts as the “name” of the rule and r is a set of symbols that identify other rules or terminals. A Context-Free-Grammar can be used to describe any (valid) string for the language it describes.

All of these rules can be used to synthesize our own expression. By generating a tree of such rules we can then convert it to a familiar expression that the student is going to draw the expression tree for.

By default, the web interface will provide 3 grammars to the user: a simple math grammar with operators precedence 8, a JSON document Listing 9 and a Racket beginner student language (BSL) Listing 10. These grammars are written in the ANTLR format, which the expression generator program will be able to understand. The language-agnostic nature of this tool allows the user to provide grammars written using the ANTLR .g4 format for the generation of expressions.

We’ll now consider the CFG made of the following rules:

```
A → 'a' A
A → B
B → 'b' B
B → '-' C
C → '1' D
C → '2' D
C → '3' D
D → C
D → ε
```

which generates strings recognized by the following regular expression $a^*b^*- [1-3]^+$. Now we’ll write an equivalent ANTLR .g4 grammar:

```
grammar ExampleGrammar;
```

```
A: 'a' A | B;
B: 'b' B | '-' C;
C: [1-3]^+
```

From this simple example it’s clear how the syntax adopted by ANTLR differs from the one used to write a CFG and takes several clues from the regular expressions syntax. In particular:

- String terminators are written between single quotes ' '
- Multiple alternatives for each rule are separated by the pipe | character rather than having a “declaration” for each alternative.
- The ? character states that the item it’s written after is optional (repeated 0 or 1 times).
- The * character states that the item it’s written after will be repeated from 0 to ∞ times.
- The + character states that the item it’s written after will be repeated from 1 to ∞ times.
- The (...) are used to create “groups” to which a repeat modifier is applied. E.g. ('b' b)+ means that the sequence / group of elements 'b' b will be repeated from 1 to ∞ times.
- The [□ – □] is used to indicate that any character in inclusive range between the value in the first and in the second □ may be used.
- The [...] are otherwise used to indicate that any character among those written in between the squared parenthesis may be used.
- There’s no equivalent of the empty string ϵ character, instead repeat operators must be used.

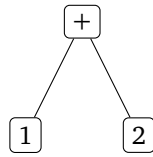


Figure 1. An AST for the 1 + 2 expression

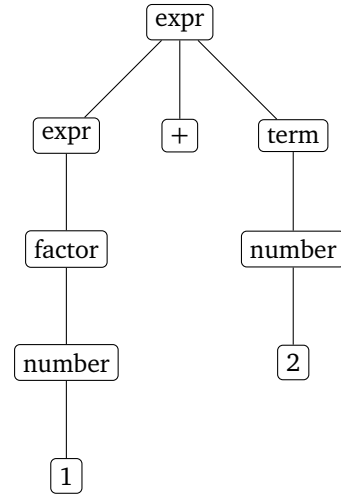


Figure 2. A CST for the 1 + 2 expression

Figure 3

Abstract and Concrete Syntax Trees

Although in order to generate an expression we need to make a tree of the rules of the Context-Free-Grammar, this tree may appear significantly different from the tree that students are going to be required to draw as the “expression”, for example when presented the simple expression $1 + 2$ the student will draw the tree on the shown on the left side of Figure 3, while the tree the expression generator builds for the same expression would look like the one on the right.

The first tree shown here, is an Abstract-Syntax-Tree (AST) (Figure 1) and differs from the second one, which is a Concrete-Syntax-Tree (CST) (Figure 2).

The Concrete-Syntax-Tree represents the code exactly how it was written and according to all the rules of the grammar: by parsing a CST one would be able to reconstruct exactly the source code snippet. On the other hand, an Abstract-Syntax-Tree only cares about representing the meaning of the expression, and so it can be thought as a simplification of the Concrete-Syntax-Tree and is more suited for the use-case of the Expression-Tutor platform, where students can learn how programs work by drawing the (Abstract-Syntax-) Trees of expressions.

3 Expression Generator

In this chapter we will discuss about how an user can interact with the Expression Generator tool through all its different supported interfaces and provide information about the knowledge that the user is expected to have in order to be able to properly use it.

3.1 Constraints

As discussed in the introduction, when the instructor wants to prepare a number of different exercises for students, it is important that the *complexity* of the expression string that is generated can be controlled in order to ensure not only that the exercises are equally difficult, but also to be able to follow the evolution of the students’ abilities through the teaching process.

A simple and quite effective way to measure the *complexity* of an exercise that consists in drawing the trees of expressions can be measured as the size of the tree one has to draw: the expression generator provides means to directly control the depth of the tree that it’s going to be generated. For *depth of the tree* we consider the number of “levels” of nodes starting from 0 for the root node.

Two parameters that act as the lower and upper bound are available to the user: the min-depth parameter defines a lower bound on the tree depth, while the max-depth will define the upper bound. The bounds are handled in a best-effort way, since it’s not possible for all grammars to allow generation of a proper Concrete-Syntax-Tree with any precise depth. It is possible that either the upper or the bound might not be respected with an error margin that depends on the grammar itself. Let’s consider this simple grammar definition:

$A \rightarrow 'a' B$
 $B \rightarrow 'b' C$

$C \rightarrow 'c' D$
 $D \rightarrow 'd'$

Listing 1. Example grammar with a fixed depth

- If the tool is instructed to generate an expression with min-depth equal to 10 starting from A, it won't be able to satisfy said requirement since the grammar has an hard upper bound for the CST depth of 3.
- Likewise, if the max-depth parameter is set to 2 starting from A, it will be impossible to satisfy as well because, again, the grammar has a lower bound for the CST depth of 3.

In a situation in which the depth constraints are not respected, the tool still provides a result, but it will also display a warning to the user about not being able to satisfy the requirements.

Let's now consider a more complex example by using the simple math grammar (Listing 8). Assume we want to generate a tree of depth exactly 4 (by setting both the lower and upper bounds to 4) that starts from the `expr` rule, a generated tree could look like the one shown in Figure 4.

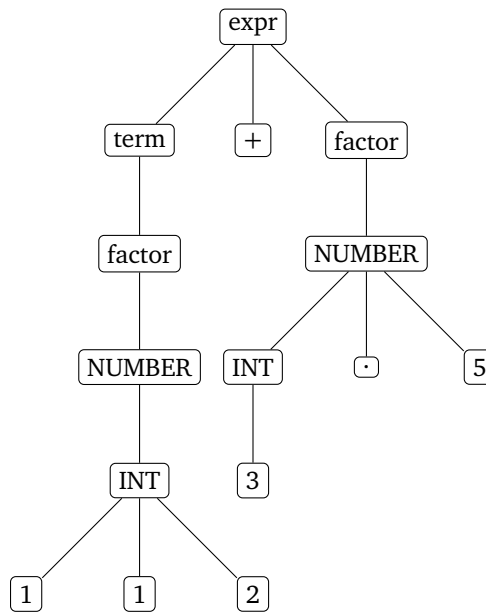


Figure 4. The smallest CST for a Math expression that starts from the `expr` rule

By looking at this tree, it's clear that any attempt at generating something with depth lower than 5 that starts from the `expr` rule would not be compliant with the grammar definitions, and so the expression generator following the grammar rules will try to terminate everything properly as soon as possible, in this case generating a tree with depth 5.

There also exists a parameter, `max-repetitions`, to limit the global number of repetitions that are possible when generating the tree to help constrain the tree width. In Figure 5 we see an example of a tree, generated from the the JSON grammar (Listing 9) that has depth 2, but can potentially *branch infinitely* due to the repetitions in the `arr` rule.

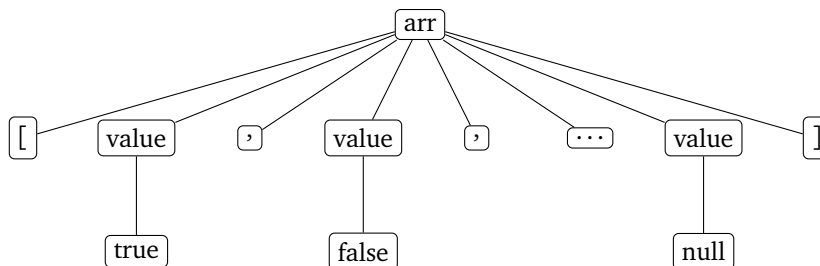


Figure 5. A CST that can branch infinitely

When we generate the Concrete-Syntax-Tree for the expression, we also need to have a starting rule. This, combined with the possibility of having a “rule inclusion” parameter allows the user to control the contents of the

generated expression, a feature that's particularly useful when we want to prepare an exercise regarding a specific topic. By specifying the start and the include parameters, the program will generate an expression that is guaranteed to make use of the specified rules.

3.2 Usage

The expression generator tool is provided in two flavours: as a command line executable and as a web service. Being built in the Java programming language, it's portable across different platforms.

3.2.1 Command line interface

The command line interface allows to generate an expression by running a single jar file with the appropriate parameters.

To execute the user must supply an ANTLR g4 grammar file and a starting rule name as the first two arguments, then the following parameters and flags are available to define the behavior of the generator:

Flag name	Short flag	Description
--include		Specify a grammar rule that must be included in the generated expression
--max-depth		Maximum generated tree depth. See Section 3.1
--max-repetitions		Maximum number of repetitions. See Section 3.1
--min-depth		Minimum generated tree depth. See Section 3.1
--out	-o	Output file (defaults to STDOUT if not specified)
--seed	-s	RNG seed for reproducible results
--debug	-d	Debug mode. Traces the algorithm execution and prints the CST
--debug-out		Debug output file (defaults to STDERR if not specified)

By using the STDOUT and STDERR for expression generation output and debug mode respectively it is possible to make use of the unix pipeline to easily customize the output of the program or integrate it in a more complex workflow. For example, if an user wanted to generate an expression and see the generated CST while not being bothered with all the algorithm execution trace logs generated with the debug flag, using the standard sed tool it's possible to filter out the unnecessary output, as seen in Figure 6.

3.2.2 Web

The web service interface allows the expression generator to be invoked by web applications while providing feature parity with the simple command line interface. The following endpoints are made available for usage to the user:

- Method: POST
 - Route: /api/expression
 - Description: Generates an expression from the given grammar and constraints set.
 - Request body:
 - grammarSource: string that contains the .g4 grammar source. **Required.**
 - startRule: string that contains the name of the starting rule. **Required.**
 - includeRule: string that contains the name of a rule that must be used while generating the expression. **Optional:** defaults to empty / null.
 - rngSeed: the random number generator seed to be used for reproducible results. **Optional:** defaults to a random value.
 - maxDepth: maximum depth of the generated expression tree. **Optional:** defaults to 20.
 - minDepth: minimum depth of the generated expression tree. **Optional:** defaults to 0.
 - maxRepetitions: maximum number of repetitions that the Concrete-Syntax-Tree generator is able to create. **Optional:** defaults to 20.
 - Accept: application/json
 - Content-Type: application/json
 - Response body:
 - expression: string containing the generated expression.


```

INT: [0-9]+;
COLOR: 'red' | 'green' | 'blue';

$ curl -X POST '${EXPR_GEN_URL}/api/expression' \
  -H 'Accept: application/json' \
  -H 'Content-Type: application/json' \
  --data-raw '{
    "startRule": "start",
    "includeRule": "COLOR",
    "grammarSource": "...
  }' # Grammar source redacted for length
I like the color red

```

Listing 2. Example usage of the micro-service Web API label

Expression Tutor integration

This web service is primarily intended for usage with the Activity Designer of the Expression Tutor platform. Expression Tutor allows users, among other features, to build “expression trees” (which are Abstract–Syntax–Trees) from a given expression code snippet in a *parse tree* activity using a custom React component specifically developed for this platform. There exists also an inverse kind of activity which consists in reconstructing the expression code snippet from a given expression tree.

The Activity Designer was extended to integrate with the Expression Generator by exposing an user interface component to create a new *parse tree* activity. If a language that is supported by the expression generator is selected by the user, an additional section will appear. This section, titled “Expression generator”, consists of an expandable layout which, once expanded, contains on the left a text area that displays the editable grammar sources (so that it can be freely customized by the user) while on the right side there are fields to specify the generation constraints. By clicking the “Generate” button, the pre-existing “Code” field below will be populated with the generated expression. The content of this field will be then shown to the students as what they will have to draw expression trees for.

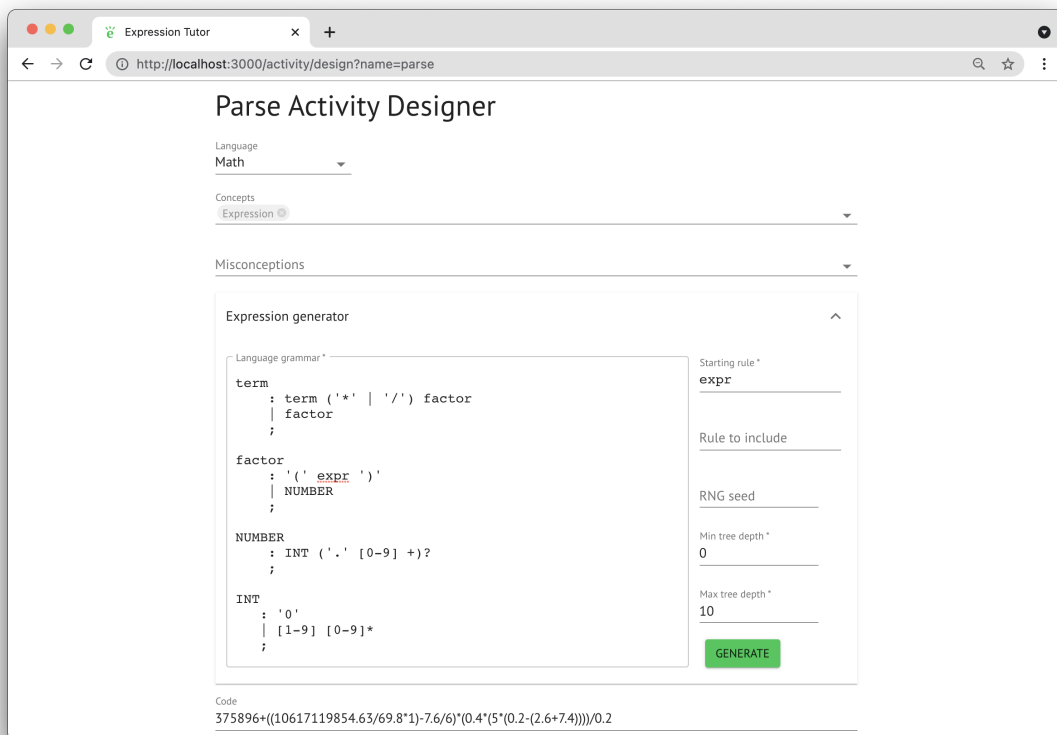


Figure 7. Generate a Math expression from Expression Tutor

4 Implementation & design

4.1 CFG

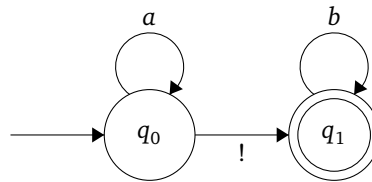
In order to generate an expression, the tool needs a set of rules to combine to create something that makes sense. These rules are defined in a *Context-Free-Grammar* (CFG): a CFG, as defined by Chomsky [2], is a formal grammar in which we have a number of production rules and terminal and non-terminal symbols that can be thought of as “an oracle that determines if a sentence complies to it” [7]. It is possible also possible to think of grammars as generators of all the possible strings that belong to the defined language.

The rules of a Context-Free-Grammar are written in the format $R \rightarrow q$ where R is a non-terminal symbol, and q is a set of terminals and non-terminals (but it can also be empty – ϵ).

Let’s consider this simple example that showcases many different “features” of the Context-Free-Grammars such as recursion and multiple alternatives for each rule:

```
A → 'a' A
A → '!' B
B → 'b' B
B →  $\epsilon$ 
```

Using this simple Context-Free-Grammar we can generate strings such as: “a!b”, “aa!b”, “a!”, “!bbb” and “!”. In particular all the strings generated by this grammar can be recognized by this non-deterministic finite automata:



and the following regular expression: $a^*!b^*$.

Let’s now consider this slightly more complicated CFG:

```
START → SUBJECT 'like' TARGETS
SUBJECT → 'you'
SUBJECT → 'we'
TARGETS → TARGETS ',' TARGET
TARGETS → TARGET 'and' TARGET
TARGET → 'the color' COLOR
TARGET → 'the number' NUM
COLOR → 'red'
COLOR → 'green'
COLOR → 'blue'
NUM → '0'
NUM → '1' DIGITS
DIGITS →  $\epsilon$ 
DIGITS → DIGITS '0'
DIGITS → DIGITS '1'
```

Listing 3. Example of a CFG

using this grammar we can then construct the following sentence: “*we like the color red and the number 10*”, which can be parsed as the CST in Figure 8.

4.2 Parser generators

Now that we have found a way to declare how expressions can be generated, the next step would be implementing a proper way to take all these rules and utilize them together to compose a new expression. The initial approach involved the usage of a parser generator to obtain information about the grammar fed into the program for generating new expressions out of it.

Parser generators, are tools capable of generating source code for parsers of grammars. They are a specific type of compiler-compilers (also called compiler generators) that focus on syntactic analysis of the grammar.

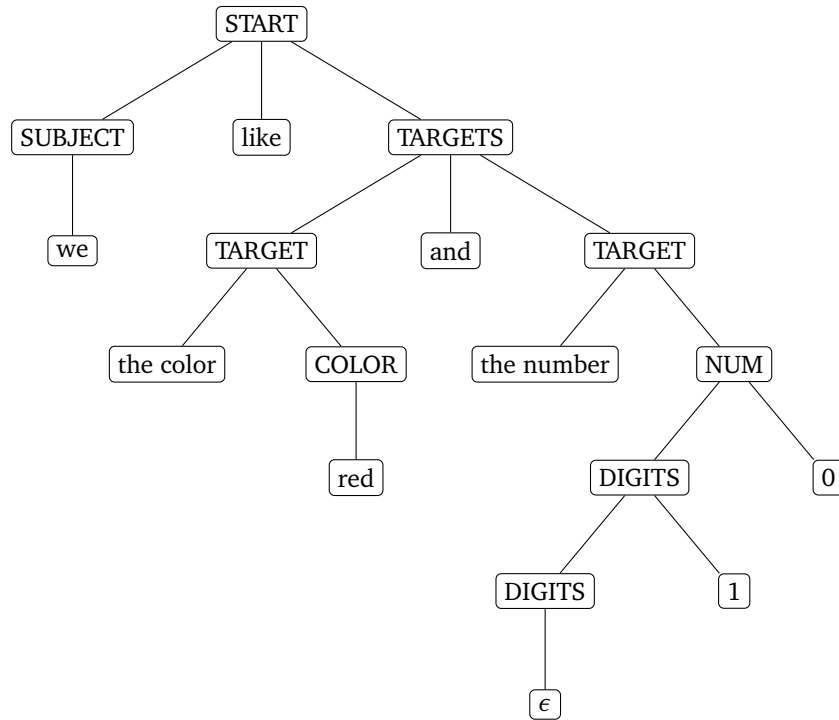


Figure 8. A CST for an expression with respect to the CFG from Listing 3

Examples of notable parser generators include YACC (Yet Another Compiler Compiler – developed for the Unix operative system at Bell Labs), GNU Bison (used in many popular tools such as Bash, CMake and PostgreSQL) and ANTLR, which is what we will be using in this project.

4.2.1 ANTLR

ANTLR is a powerful tool capable of generating parsers and tree walkers from formal language descriptions (grammars). This tool is popular thanks to its contributions to both the theory and practice of parsing, in particular with the LL(*) parsing strategy [10].

As explained in the book “The definitive ANTLR 4 Reference” [9], ANTLR grammars can be used to build parse trees and visit such nodes to execute some application specific code. If we take our simple math grammar (Listing 8), we can build a program that can interpret and execute calculations by making use of the classes generated by ANTLR: if we feed our program an expression like ‘(2.5 + 7) * 3 – 21/2’, then using the parsers and tree walkers ANTLR can generate from the grammar descriptor file, we will be able to output the computed result. To do this, first, we need to define a data structure to build an Abstract-Syntax-Tree of the given expression, so that by walking through it, the calculation result can be computed. This data structure can be defined as:

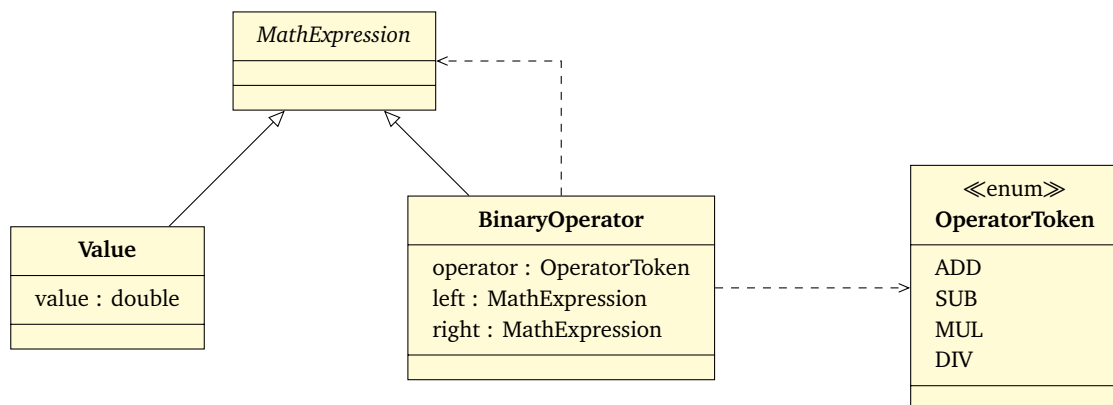


Figure 9. UML diagram of the MathExpression class

By implementing a visitor class, we can map each rule to an “expression node” of our data structure:

```

expr    → BinaryOperator: left (+ | -) right
        → $term
term     → BinaryOperator: left (* | /) right
        → $factor
factor   → $expr
        → Value: number

```

Then, by using our visitor in combination with the parser class generated by ANTLR, we can obtain, for the $(2.5 + 7) * 3 - 21 / 2$ expression, the following data structure:

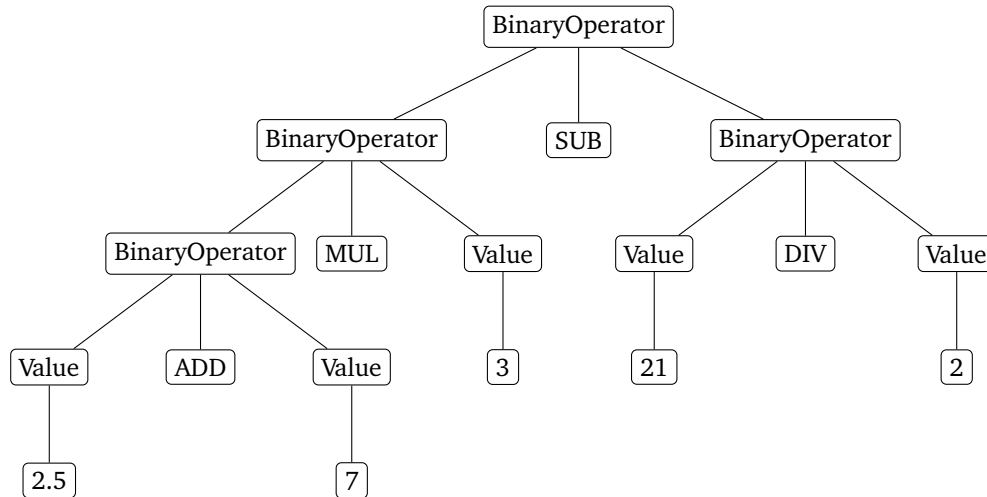


Figure 10. A tree-like representation of the data structure generated by using the parser generated by ANTLR on the example expression

Finally, using a simple recursive tree walker we can go through this data structure and compute the result “bottom-up” by performing the appropriate operation in each *BinaryOperator* node.

By putting all these components together we have a fully working interpreter of the simple math grammar:

```

final String source = "(2.5 + 7) * 3 - 21 / 2";

final MathLexer lexer = new MathLexer(CharStreams.fromString(source));
final MathParser parser = new MathParser(new CommonTokenStream(lexer));

final ParseTreeVisitor<MathExpression> visitor = new MyMathVisitorImpl();
final MathExpression ast = visitor.visit(parser.expr());
final MyMathInterpreter interpreter = new MyMathInterpreter()

final double result = interpreter.interpret(ast);
Assert.assertEquals(18.0, result, 0.0);

```

Listing 4. Java code snippet for a simple calculator app made with the help of the code generated by ANTLR

4.3 Generating expressions

The first step that needs to be taken to generate an expression is getting to know the language which we want the tool to *speak*. Initial approaches focused on the the analysis of the parsers and tree walkers generated by ANTLR to understand the grammar rules, but proved over-complicated. So, the approach to the problem of parsing grammars was changed, while still retaining the use of ANTLR: instead of using the parser generator as intended to, it'd be easier if we could invoke only the code involved in the generation of the parsers up until the parser classes generation. This way it'd be possible to have access to the very same data structure that ANTLR uses internally to generate the walkers and parser classes and use it to generate our own data structure that suits our needs.

For each rule declared in the original ANTLR .g4 file, we have a number of alternatives identified by a common name. Each of these rules is described by a *RuleDescriptor* instance, which provides all the information that are needed to know the syntax of a rule and its relationships with other rules.

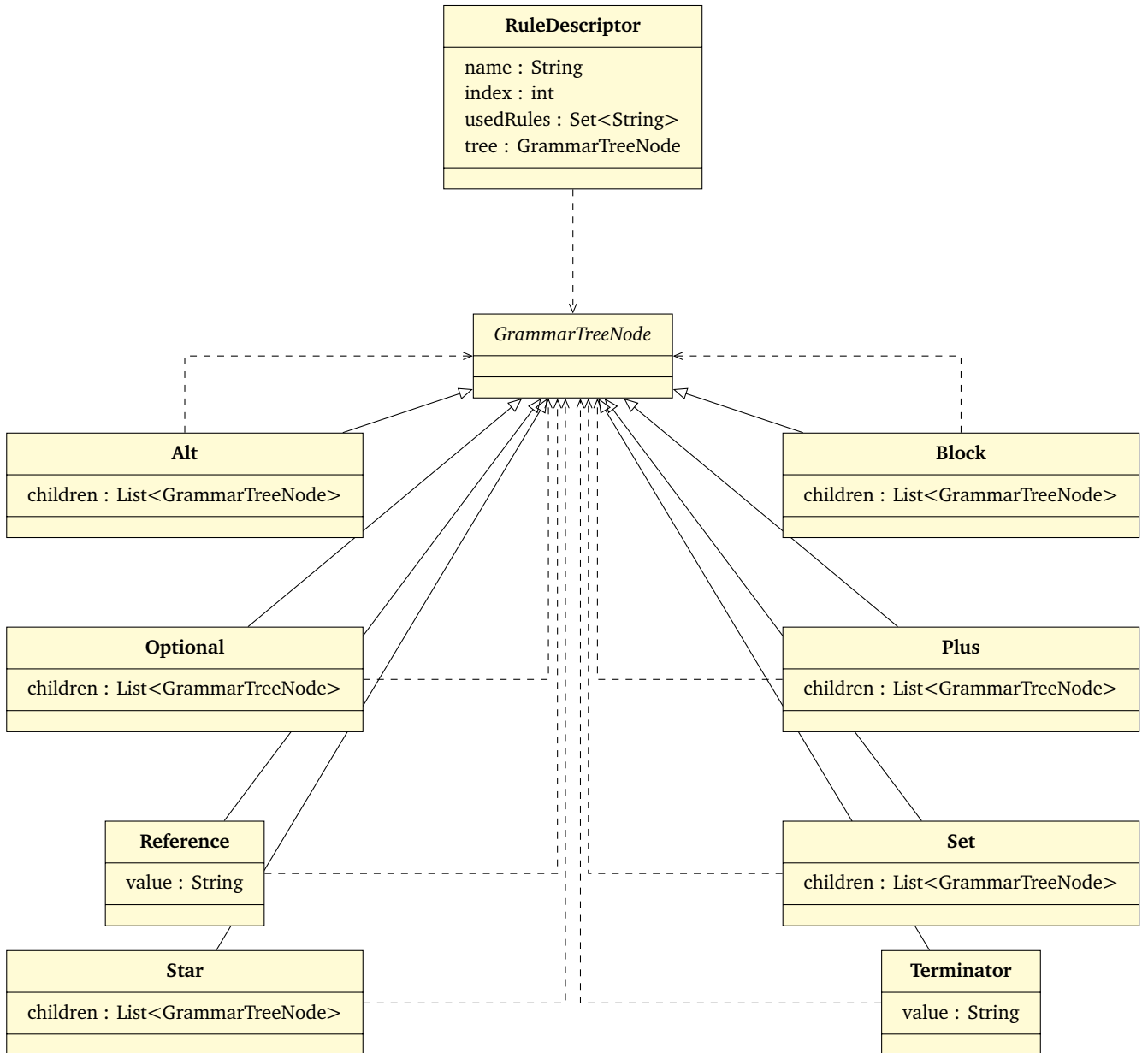


Figure 11. UML diagram of the RuleDescriptor class

By wrapping the ANTLR grammar parser, we obtain a set of **RuleDescriptor**, one for each rule implementation and we group them by name so it's possible to easily retrieve all the possible alternatives that match a given rule name.

Now that we have collected the information about the grammar, we are able to begin generating our expression.

The first prototype iteration of the expression generation algorithm was built without constraints in mind. It was in fact based on fuzzing tools such as those described in the *Grammar* chapter of the “The Fuzzing Book” [12]. Fuzzing is a very powerful technique that can aid uncover issues in software by executing tests with randomly generated inputs / arguments.

Since we want our generated expression to be somehow *randomized*, we will make use of a random number generator to let our algorithm make choices when presented with multiple viable alternatives. In order to ensure consistency of results across different machines and different JVM implementations, we will be using a built-in random number generator (RNG), an implementation of the PCG 32 algorithm. PCG32 is part of the PCG family of random number generators algorithms, which are designed to be “Simple Fast Space-Efficient Statistically Good” [8]. For our use case, simplicity of implementation and efficiency in terms of space and computation time are more important over the unpredictability of results since we will simply be dealing with making choices among a small number of options and being able to predict which numbers are going to be generated has zero benefits.

While the fuzzing-based approach was initially effective, as constraints were added, it became apparent how this approach was not sustainable, hence why the algorithm was re-written from scratch. Now, the expression generation

algorithm takes clues from routing algorithms, but with several changes in order to support the more complex use case of generating a Concrete–Syntax–Tree within a set of user-defined runtime constraints (see Section 3.1).

Fulfilling constraints

The basic idea of the algorithm is to start from one of alternative of the “starting rule” defined by the user, reach the “must include” rule (if defined by the user), and then conclude the creation of the nodes until we have generated all the terminators. While we have a “destination” to reach (the rule that must be included in the generated expression), our algorithm will work pretty much like a normal routing algorithm in which we try to find our way to the target, although not always in the shortest possible path, as we will see later. When we have reached our “destination” (while making random viable choices) at least once, then we don’t care about what the expression will include next and the algorithm will make only *viable random* choices until the end.

We to constrained the tree depth to be within a certain range between the lower and upper bounds. This means that reaching our “must include rule” (or terminating the tree) too early or too late is not ideal.

In order to ensure that the lower bound on the tree depth is respected, we choose to introduce purposeful *mistakes* in the routing algorithm. These mistakes are similar to when we’re driving and we take a wrong turn, the car navigator will recompute a route that still allows us reach the destination, while taking longer than previously anticipated. We make the algorithm select a non-optimal path. This non-optimal choice still guarantees that the “destination” is reachable, but it will take longer, maybe even circling back to the node where we were, so that the tree depth naturally increases. Controlling the upper bound on the other hand is done by limiting our choices to those that allow us to reach the “destination” (or the terminator child node) within a number of “steps” that are within the user-defined bound.

Another constraint is the *number of available repetitions*, which controls the number of maximum repetitions for the star ‘*’ and plus ‘+’ operators in the generated rules. As it effectively helps constrain the width of the tree to avoid *infinite branching*, this restriction is not applied to optional ‘?’ operators as they are *repeated* at most one time. Every time we encounter a star ‘*’ or plus ‘+’ operator, a random choice of how many times it is repeated is made by “taking” a number from the available repetitions left.

All these constraints have to fit together when generating an expression, and this may create some conflicts that need to be resolved.

```
A: 'a' B* | 'z';
B: 'b' C;
C: 'c';
```

Listing 5. Fulfilling rule inclusion and max repetitions constraints

Let’s consider for example the grammar from Listing 5, and assume we are generating an expression from rule A and we want to include rule C. To do so, we are required to pick the first alternative implementation of A which uses rule B, from which we can reach C, our “destination”. If we had no available repetitions (either because of the constraint was set to 0 or because we already have used them all earlier in the generation of other nodes), we’d have a *conflict of interest* between the two constraints: one wants the algorithm to visit B for reaching the rule that the user asked to include, while the other doesn’t want us to visit B because we’re out of repetitions. In this situation, the algorithm will prioritize satisfying the rule inclusion and will make an exception allowing B to be reached only once (with no additional repetitions). The same logic is also applied in the case where the “destination” is reached through an optional ‘?’ node, in which case it’ll forcefully be included.

The tree height bounds and rule inclusion constraints must also be aware of each other when they are being used to choose viable choices.

```
A: 'a' B | 'a' Z;
B: 'b' C | 'b' Z;
C: 'c' D;
D: 'd' Z;
Z: 'z';
```

Listing 6. Fulfilling tree depth and rule inclusion constraints

Let’s consider the grammar from Listing 6 and the case in which from rule A we want to include rule Z in a tree that has depth exactly 2 (lower bound = upper bound = 2). There are three ways to reach Z if we are in A: from the second alternative implementation of A, from the first implementation alternative of B or from the second implementation alternative of B. The first option would generate a tree of depth 1 which violates both the lower and upper bounds. The second option (first alternative of B) would generate a tree of depth 4, which satisfies the

lower bound but violates the upper one. The third option (second alternative of B) would generate a tree with depth 2, which is exactly what we want. If the last option had not been available, the algorithm would have adopted a best-effort approach by terminating as soon as “grammar-ly possible” after having satisfied the lower bound. In our example, it would have chosen the second option, generating a tree of depth 4. Whenever bounds are not respected, a warning message is given to the user to inform that it was not possible for the generator to fulfill the requirements.

As seen in Listing 1, grammars may not allow constraints to be respected. In this example, if we start from rule A we are forced to generate a tree of depth exactly 2, regardless of the user’s depth constraints. Similarly, if we were to generate an expression that starts from rule B we’d have no way to reach rule A and include it in our generated expression. In this case, the algorithm will simply inform the user that it is not possible to reach the desired rule from the selected starting rule.

Algorithm in details

The algorithm is implemented as a recursive function, which has a state defined by the following values:

1. Maximum (available) depth (tree height constraint).
2. Minimum (available) depth (tree height constraint).
3. Name of the rule we have to build (inclusion constraint).
4. (Reference to the) Number of available repetitions (width constraint).
5. (Reference to the) Name of the rule to be reached (once the rule has been visited at least once or if there was no specific rule to reach, the value is *nullified*).

The maximum and minimum available depth are decreased by 1 at each recursive call in order to keep track of the depth of the tree we are building.

The reason why we are using the reference to the number of available repetitions rather than the a plain value is to ensure that its changes get properly propagated both “down” and “up” in the recursion. To achieve this the value is stored outside of the recursion and retrieved when needed for computations. Similarly, the value of name of the rule we want to include is stored outside of the recursion, so that it can be *nullified* once it has been included at least once.

grammar JSON;

```
obj: '{' pair (',' pair)+ '}' | '{' '}' ;
pair: STRING ':' value;
value: STRING | NUMBER | obj | arr | 'true' | 'false' | 'null';
NUMBER: INT ('.' [0=9] +)?;
// [...]
```

Listing 7. Subset of the JSON grammar

Let’s assume references are not used. We are generating an expression with the grammar from Listing ??, starting from an obj node and we are asked to include the NUMBER rule. To do so, we’d pick the first obj alternative, which contains a pair node that can reach the desired rule. What our algorithm would do is build a pair node with the requirement to reach the NUMBER rule. From here we would have to build a STRING and a value nodes. While building the value node we’d finally have the chance to include a NUMBER rule. Once that happens, we’d mark the “to include rule” as reached and propagate it to all of its children as well (just the INT node in our case). Because of the plus ‘+’ operator in the obj definition, we’d still have to build at least another pair node. In this case all the recursive method calls would be completely unaware of the fact that the NUMBER requirement had already been satisfied, forcing the second pair node to reach the NUMBER rule as well. This would introduce a bias towards the usage of the NUMBER rule, which is not what we want. Once we reach the rule that the user wants included, the algorithm should be freed of this constraint. That’s why we want to use references for the rule to include and the number of available repetitions for the width constraint.

Now that the *state* of the recursive function has been defined, the first thing we have to do is decide which “alternative” of the rule we’re visiting. To do so we have to make use of a rule picker algorithm. This algorithm is responsible for picking the best rule alternative given the depth constraints and a possible destination. Initially we filter among all the rule alternatives defined by the grammar those that have as name, the name of the rule we’re visiting. Then, if there is a destination to be reached, we additionally filter those that can reach it. Now these rules will be divided in three different lists: the first list will contain all the elements that will reach the destination (or

terminate) too early with respect to the depth bounds, the second other will contain those that terminate within bounds and finally the last one will contain all the alternatives that terminate (or reach the destination) too late. If the list of elements *within bounds* is not empty, then a random item of it will be selected. Otherwise if the list of elements terminating too early has at least one element and we still haven't satisfied the minimum depth requirement, then we pick a random element among those terminating too early that also happen to be able to reach the node we're visiting right now, so that we can guarantee we'll still be able to reach our destination if we have it, otherwise we've still made a good choice that increased our tree depth. If we couldn't satisfy the previous cases, we simply make a random choice. Finally the last case is the one where all the rules we have at our disposal don't terminate within the tree depth upper bound, in which case we pick a shortest reaching rule.

So far we have figured out which rule alternative to use, the next step is actually building the corresponding nodes. To do so, the `tree` field contained inside the `RuleDescriptor` (see Figure 11) instance of our rule, provides clues regarding how it is possible to build the Concrete-Syntax-Tree (CST) for a given rule alternative. The `tree` field is an instance of `GrammarTreeNode`. A `GrammarTreeNode` is one of the following:

1. `Alt`: represents an alternative grammar node.
2. `Block`: a node that contains other `GrammarTreeNode`s child nodes grouped together.
3. `Optional`: a whose `GrammarTreeNode`s child nodes may or may not be included in the generated CST.
4. `Plus`: a node whose `GrammarTreeNode`s child nodes may be repeated one or more times.
5. `Reference`: a node that references another grammar rule.
6. `Star`: a node whose `GrammarTreeNode`s child nodes may be repeated zero or more times in the generated CST.
7. `Set`: a node that contains a number of alternative `GrammarTreeNode`s child nodes. Only one of them shall be built and included in the generated CST.
8. `Terminator`: a node that contains a terminator value.

We can build our CST, by walking through the `tree` field of the `RuleDescriptor` instance we have chosen. If we run into an `Optional` the random number generator (RNG) is used to decide whether the node will be included or not. For the case of the `Plus` and `Star` nodes, we use the RNG with the highest possible generated value set to the number of available repetitions (which is then decreased once the value is chosen) to decide how many times to repeat the child nodes. When there's a `Set` node the node to be built is selected by the RNG. For the case of the `Reference` node, we invoke the original function to build a new node recursively, changing the state to have decreased maximum and minimum tree depth, and set visiting rule to the name of the referenced rule. Additionally if the name of the referenced rule is equal to the name of the rule we want to include, we set the "to visit" referenced value to null so that it's marked as *visited* which means we no longer have to reach it. The last case is the `Terminator` node, in which we simply build a *terminator* node that holds a string value that will be used in the final generated expression.

Now that we have generated the Concrete-Syntax-Tree of our expression, we can finally convert it to a string expression that users will read. To do so, a simple tree-walker algorithm that concatenates the string values of the terminator nodes together in the right order has been implemented.

To improve the "randomness factor" of the generated string, support for regular expressions-like character ranges has been added. If a terminator matches the familiar regular expression syntax of character ranges a random character from said ranges will be generated. This allows for the generation of random numbers and strings in the final output.

For example, the range `[A-Z]` includes all the characters from the uppercase A to the uppercase Z. Similarly, the range `[0-9]` has all the ten digits. The ranges can be combined like in `[A-Za-z]`, which includes any char from uppercase A to uppercase Z and from lowercase A to lowercase Z.

4.4 Architecture

The Expression Generator is built as a Java project. It provides two user interfaces (web micro-service and command-line based), the grammar parser and the expression generation algorithm.

The web micro-service was developed using the Dropwizard, a Java framework that *"pulls together stable, mature libraries from the Java ecosystem into a simple, light-weight package that lets you focus on getting things done"* [4]. A Dropwizard micro-service provides some resources, which are building blocks associated with an URI template that can handle REST requests. For the Expression Generator web micro-service there are two resources. The first is responsible for generating expressions and the other one can provide some "pre-built" grammars that can be used with the other expressions generator resource or as a starting point for customized grammars.

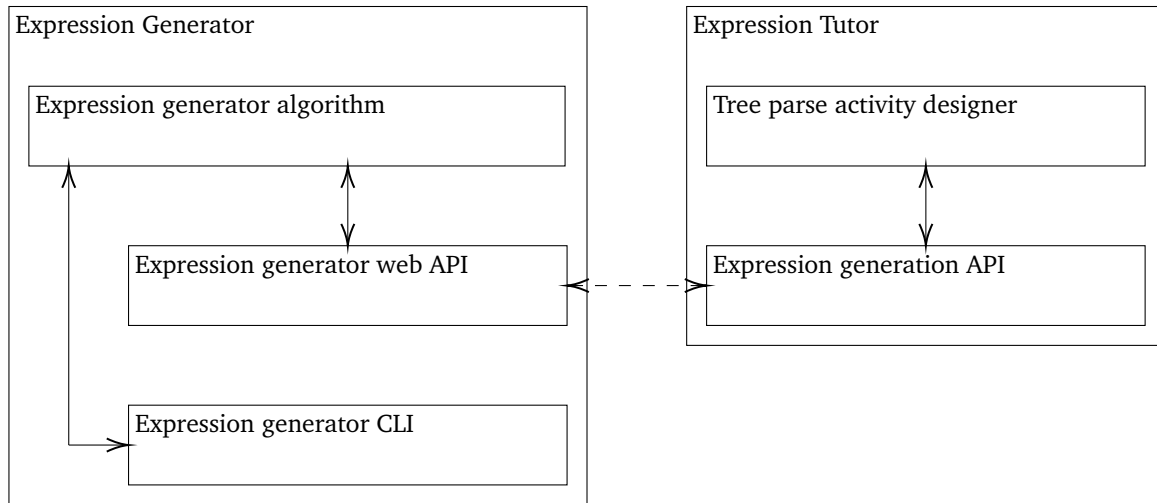


Figure 12. The overall architecture of the Expression Generator and its integration with Expression Tutor

The Expression Generator is made of the a number of modules:

1. `cli`: The `cli` module provides the command line interface to generate expressions from the command line.
2. `core`: The `core` module is responsible for working with the grammars and generating CST and expression strings from them. It contains code for parsing ANTLR grammar files as well.
3. `util`: The `util` module holds constants and utilities used across multiple other modules.
4. `web`: The `web` module holds the code for providing a simple REST API for the generation of expressions.

While this structure could be even further modularized, it already allows, for instance, to generate *minimal* versions of the program that do not contain features that are not required, in case we would want to deploy the program on a device with low storage availability. For example, by not including the web user interface module, not only the code and resources contained within the web module do not get included, but also all the dependencies such as the web server framework.

When deploying the Expression Generator, it is possible to choose which user interface to use: the web micro-service or the command-line interface.

As a client for the web APIs of Expression Generator, an integration for the Expression Tutor was developed. Expression Tutor is developed in JavaScript using the Next.js React framework and Material UI components for the user interface. The Expression Tutor was modified to include an additional user interface component in the *Parse Activity designer* to aid in the creation of exercises by generating expressions that students will have to draw expression trees for. Some grammars (the Math grammar from Listing 8, the JSON grammar from Listing 9 and the Racket BSL grammar from 10) are provided in the Expression Tutor website so that they can be used as they are or customized right from the user interface. The integration with the Expression Generator web micro-service is made through Next.js APIs added to Expression Tutor that wraps the Expression Generator micro-service.

The Expression Generator project is built using Bazel [5], the open source multi-platform build system designed by Google for internal usage but later released to the public due to its widely appreciated feature set. It aims to be fast, support multiple languages and be completely reproducible. Bazel encourages the division of the code base in smaller reusable units so that compilation can be parallelized and smaller chunks of code have to be recompiled when a file changes. Using Bazel it is possible to generate *deployable Jars*, which are compiled “fat” jar files that contain all the dependencies within them so they can be easily executed on any JVM (that supports the JDK version against which the Jar was compiled — in this case Java 11) without having to load external class-paths or other dependencies.

4.5 Code quality

Verifying the quality of the code is fundamental to keep a project maintainable and catch issues earlier. To do so, a number of code quality tools have been integrated in the development toolchain:

- `CheckStyle`: a development tool that ensures that Java code adheres to a specified coding style. The coding style adopted for the Expression Generator project is the Google Style [1].

- PMD: Programming–Mistakes–Detector (PMD) is a static source code analyzer that can find many common mistakes [11].
- ErrorProne: augments the Java compiler’s type analysis to catch more errors at compile time [6].

These three tools are executed at build time: while ErrorProne is embedded in the default Bazel Java toolchain, custom Bazel plugins were developed for CheckStyle and PMD, so that code refuses to compile in case any violation to the PMD and CheckStyle rules is found. Upon premature build termination because of a code quality issue, it will be printed in the build failure output message. Bazel plugins are scripts written in the Bazel StarLark language, which resembles Python syntactically. These plugins define a number of input files and output files. If any of the input file changes between the previous and the “current” build, a set of commands that is expected to produce the files declared as output using the files declared as input, is executed. This simple mechanism is how Bazel ensures that only the needed files are recompiled or set of commands executed, but unlike other build systems such as the popular GNU Make, the check for file changes is more powerful and is not tricked by the touch command or change of ownership. Moreover, since the outputs are assumed to be deterministic (even if they are not they are treated as so), it is possible for Bazel to have “shared compilation cache” that allows different machines to share build artifacts to cut down compilation time by simply copying from the cache rather than building the exact same target with the exact same inputs. Thanks to this powerful mechanism, it is easy to integrate such powerful code quality tools without having significant negative impact on build time: only files that change between builds are checked.

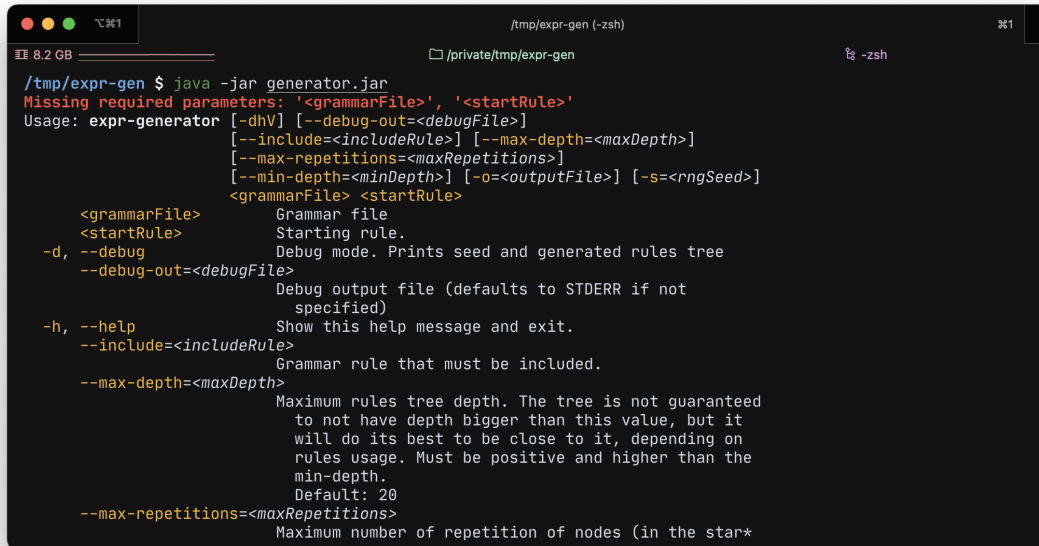
Other than checking that the code respects a common set of good practices through the usage of these three tools, it is also important to ensure that the code actually works as intended. To do so *testing* is required. While most of the code is tested with classical JUnit4 tests, some *property-based* testing was introduced. Property-based testing is a practice that originally gained popularity in the Haskell programming language community thanks to the QuickCheck library [3]. It works by having properties that a function should fulfill. The testing executor software will ensure that the method under test by giving generated random inputs. If a single test case falsifies the property, the property no longer holds and the test is considered failed. The values that falsified the property are then exposed in an error message for the developer to reproduce, investigate and solve the issue. In the Expression Generator case, property-based testing is used to ensure constraints are always respected. Each constraint can in fact be considered as a property that must hold true in order to satisfy the user request, the generated Concrete–Syntax–Tree depth must be constrained and the requested rules included. Unlike fuzzing techniques, which attempt to break programs with certain input combinations, property-based testing wants to verify the logical value of a certain proposition. This technique proved very useful indeed because it helped expose some corner cases that would otherwise have been difficult to find. Finally code coverage is measured using the Bazel built-in tools which output coverage reports in the LCOV format, which can then be exported to HTML, XML or plain text reports.

5 Validation

5.1 Command line interface

In order to execute the Expression Generator tool from the command line, we use the *deployable JAR* generated by Bazel (see Section 4.4). This JAR file includes all its runtime dependencies, so there's no need to specify any Java class-path and it can be simply executed.

If no arguments (or the `--help` / `-h` argument) are supplied, the program will display this help menu that details all the information about the program usage and the supported parameters.



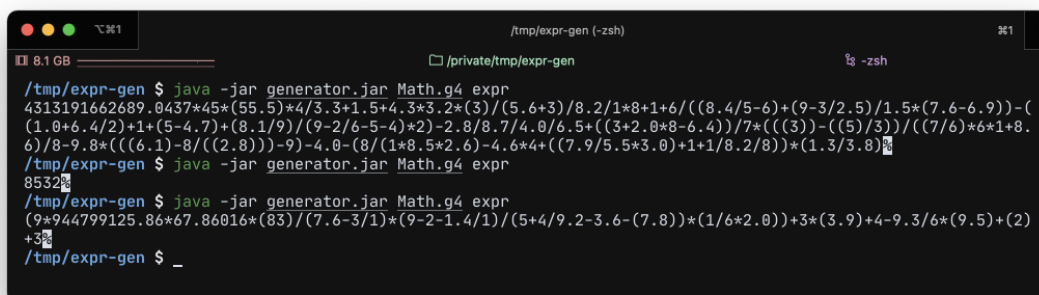
```
/tmp/expr-gen (-zsh)
8.2 GB
/private/tmp/expr-gen
-zsh

/tmp/expr-gen $ java -jar generator.jar
Missing required parameters: '<grammarFile>', '<startRule>'
Usage: expr-generator [-dhV] [--debug-out=<debugFile>]
                    [--include=<includeRule>] [--max-depth=<maxDepth>]
                    [--max-repetitions=<maxRepetitions>]
                    [--min-depth=<minDepth>] [-o=<outputFile>] [-s=<rngSeed>]
                    <grammarFile> <startRule>

    <grammarFile>      Grammar file
    <startRule>         Starting rule.
    -d, --debug         Debug mode. Prints seed and generated rules tree
    --debug-out=<debugFile>
                        Debug output file (defaults to STDERR if not
                        specified)
    -h, --help         Show this help message and exit.
    --include=<includeRule>
                        Grammar rule that must be included.
    --max-depth=<maxDepth>
                        Maximum rules tree depth. The tree is not guaranteed
                        to not have depth bigger than this value, but it
                        will do its best to be close to it, depending on
                        rules usage. Must be positive and higher than the
                        min-depth.
                        Default: 20
    --max-repetitions=<maxRepetitions>
                        Maximum number of repetition of nodes (in the star*
```

Figure 13. Help message of the CLI jar

By specifying as the first two parameters the grammar file path and the starting rule name, the program will generate a random expression using the default constraints (minimum depth = 0, maximum depth = 20, maximum repetitions = 20, no specific rule to include and casual random number generator seed).



```
/tmp/expr-gen (-zsh)
8.1 GB
/private/tmp/expr-gen
-zsh

/tmp/expr-gen $ java -jar generator.jar Math.g4 expr
4313191662689.0437+45*(55.5)+4/3.3+1.5+4.3*3.2+(3)/(5.6+3)/8.2/1*8+1+6/((8.4/5-6)+(9-3/2.5)/1.5*(7.6-6.9))-((1.0+6.4/2)+1+(5-4.7)+(8.1/9)/(9-2/6-5-4)*2)-2.8/8.7/4.0/6.5+((3+2.0*8-6.4))/7*((3)-(5)/3))/((7/6)*6*1+8.6)/8-9.8*(((6.1)-8/((2.8)))-9)-4.0-(8/(1*8.5*2.6))-4.6*4+((7.9/5.5*3.0)+1+1/8.2/8))*(1.3/3.8)
/tmp/expr-gen $ java -jar generator.jar Math.g4 expr
8532
/tmp/expr-gen $ java -jar generator.jar Math.g4 expr
(9*944799125.86*67.86016*(83)/(7.6-3/1)*(9-2-1.4/1)/(5+4/9.2-3.6-(7.8))*(1/6*2.0))+3*(3.9)+4-9.3/6*(9.5)+(2)+3
/tmp/expr-gen $ _
```

Figure 14. Running the program with no specified constraints to generate random math expressions

By using the appropriate argument flags, it's possible to apply constraints to the expression generator, for example by bounding the generated Concrete-Syntax-Tree depth or rule inclusion:

- `--max-depth` and `--min-depth` are used to control the generated CST depth.
- `--max-repetitions` is used to control how many repetitions through `+` and `*` operators are possible for the generated expression.
- `--seed` can be used to define the random number generator seed.

- `--include` allows the user to define a rule that must be used for the generation of the expression.

```

/tmp/expr-gen (~zsh)
8.1 GB
/private/tmp/expr-gen
~ zsh

/tmp/expr-gen $ java -jar generator.jar JSON.g4 json --min-depth=10 --max-depth=20 --include=NUMBER
[[[[[0],null,false,false,[7,"",0],false,[0]],null,false,false,[0],"",[[true]],[],true,{},[[0.9]],{"":["tru
e]]]]]]]
/tmp/expr-gen $ java -jar generator.jar JSON.g4 json --min-depth=10 --max-depth=20 --include=NUMBER
{"6R3":{"9iieN4Sis70vn":{"Y":{"1":"83,9"}}}}
/tmp/expr-gen $ java -jar generator.jar JSON.g4 json --min-depth=10 --max-depth=20 --include=arr
[{"3C8IFo8j74":{},{,"Lkvk":0,"":true,"":false},{":{"":[]},[]]}]
/tmp/expr-gen $ java -jar generator.jar JSON.g4 json --min-depth=10 --max-depth=20 --include=arr
[[[0.5757466967,"NFT6","",3.3,""]]]
/tmp/expr-gen $ java -jar generator.jar JSON.g4 json --min-depth=10 --max-depth=20 --include=pair
{"Q1Q86cjinMr389DAl":{"099":["null"]}}
/tmp/expr-gen $ _

```

Figure 15. Generating random JSON documents with a set of constraints

We see how, in Figure 16, by taking advantage of the unix pipeline it is possible to easily integrate this command line interface of the Expression Generator tool into a workflow, as highlighted by this simple example of a random Math expression being generated, printed to the shell and then its result is computed with the `bc` utility.

```

/tmp/expr-gen (-zsh)
8.2 GB /private/tmp/expr-gen -zsh

/tmp/expr-gen $ echo "$(java -jar generator.jar Math.g4 expr)" | tee $(tty) | bc
5980469766320867614+((4.2*6.9*(2.4)*(2.4+2.5*8)+3.6/(3)*(3.4+1/5)/(4.9*4+4*2.2/1.8))/(9)/4*(8.2-4.6*4.3*(9.2
)*3.3/(7/3)/8+5.9-(2.9))/(8)*((8.5)/(4.6)+6.1/(7)/(5)*(7-1*3-4.1)))*3.5)
5980469766320864167.6

/tmp/expr-gen $ echo "$(java -jar generator.jar Math.g4 expr)" | tee $(tty) | bc
((24060548508.8848)*(4.6+43.8/4.6)*9.9)*((3/7)*(3-1.9*2.6/5-8.6)+8))+8.5*6-9.3/6*(6*1.4*9.6+(1.3*4/9))*5
25916098009537.9888

/tmp/expr-gen $ echo "$(java -jar generator.jar Math.g4 expr)" | tee $(tty) | bc
3193522.62529179*((60-280/3.1*(6/(6))))/4.5+(3+6)
-21290141

/tmp/expr-gen $ _

```

Figure 16. Using the command line interface of the Expression Generator tool in a pipeline

An exclusive feature of the command-line interface is the `--debug` flag, shown in Figure 17: by setting this flag, the program will print an “algorithm trace” to the file specified by the `--debug-out` parameter (it is set by default to the `STDERR` of the shell session). This “algorithm trace” will show to the user how every single node of the generated CST was built and under which constraints (see Section 4.3 for details on how the algorithm builds the CST for the expression) and finally it’ll print a visual representation of the Concrete-Syntax-Tree from which the expression is generated. This is an useful feature that can help figure out why the generated expression *looks* the way it is.

will be generated and set to the *Code* field below. The constraint fields are validated to prevent issues such as the maximum depth being lower than the minimum depth or being negative.

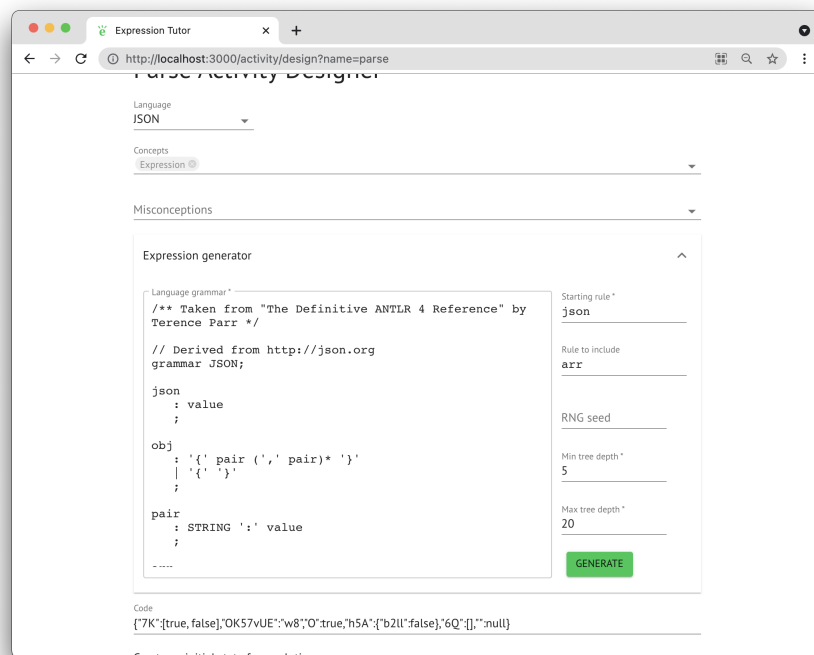


Figure 19. The *parsing activities* designer page on Expression Tutor

Finally, by selecting the “Custom” entry in the dropdown selector *Language* field, it is possible to use a completely customized grammar that can be used for generating an expression. It is also possible to edit pre-built grammars to remove unwanted language features or specify some custom terminals values (e.g. in order to have string values that are not random strings or to bound number sizes), as it is displayed in Figure 20.

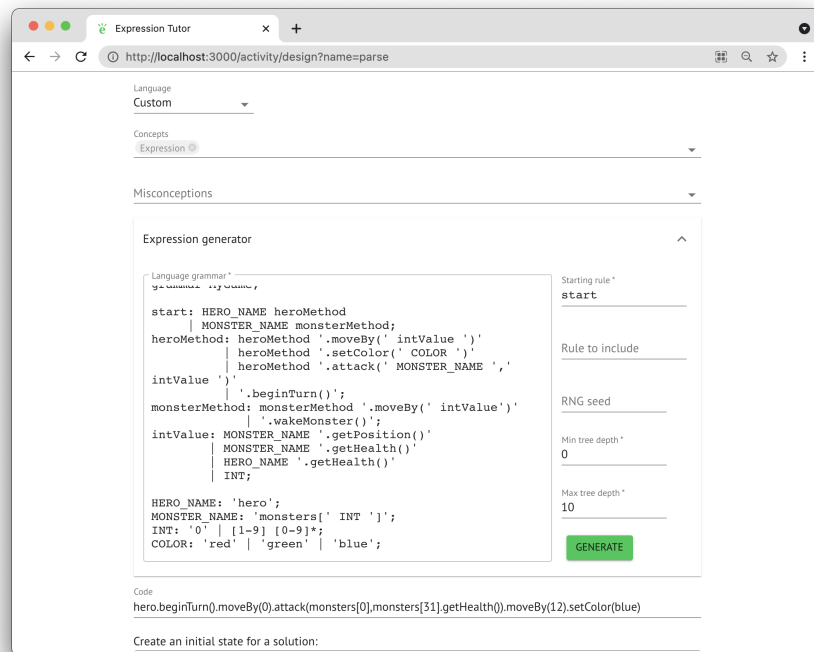


Figure 20. A custom grammar written directly in the Expression Tutor website can be used to generate expressions

6 Conclusion

We have presented a tool that is able to generate expressions from syntax definitions. By leveraging the ANTLR parser of grammars we were able to extract and store the grammar rules in an appropriate data structure. These are then used to compose expressions, while fulfilling a number of constraints defined by the user. The user can apply three constraints: one bounding the generated Concrete–Syntax–Trees depth (both from above and below), one for limiting the number of possible repetitions of the star ‘*’ and plus ‘+’ operators in the grammar definitions to avoid infinite branching and finally one to define a certain rule that must be included in the generated expression. These constraints define the *contents* and *complexity* of the expression string that’s going to be generated. The Concrete–Syntax–Tree of the expression is generated with a routing–like algorithm that makes routing choices that may not always be optimal in order to satisfy all the constraints. If some constraint can’t be satisfied, then the algorithm adopts a best–effort approach and tries to satisfy rule inclusion first, then the number of node repetitions, the lower tree depth bound and finally the tree depth upper bound.

The Expression Generator can be used with multiple interfaces, allowing it to be a versatile utility that can be easily integrated in web services, as seen in the Expression Tutor website for which an integration was developed, or other more complex workflows thanks to the command–line interface. The code quality of the project is ensured thanks to a number of tools integrated in the toolchain to avoid common mistakes and catch issues as soon as possible while also making use of powerful testing techniques such as property–based testing.

While this program was designed with the generation of programming expressions in mind, the effort put in making this tool language agnostic pays off by allowing it to be easily repurposed to generate any kind of string–based content: be it documents, plain text or program code.

Future work

While the Expression Generator tool is now in a state where it can fulfill the initial requirements that sparked its development, it has potential for improvement through the addition of new features and improvements of existing ones:

1. One of the most interesting features that would benefit the Expression Tutor platform would be the possibility of being able to provide alongside the generated expression string, the “expression tree” / Abstract Syntax Tree. By having this data structure it’d be possible to automate even more tasks, such as providing not only the question formulation but also its answer as a reference solution. Moreover, it could also assist in the creation of “worked examples” by providing the tree nodes to the Expression Tutor activity designer. The challenge

in providing this functionality lies in being able to efficiently providing a way to convert the Concrete Syntax Tree that is being generated to an Abstract Syntax Tree while preserving the language agnosticism. Some suggest that a viable approach to achieve this would require the use of “*pure and declarative syntax definitions*” [7]: grammars for which the Concrete Syntax Tree already resembles the Abstract Syntax Tree – this is made possible by re-thinking how rules precedence, disambiguation and lexical syntax are handled through the usage of “purer” syntax definitions.

2. Another potential addition to the feature set of the Expression Generator could be the support for generating expression that must include more than one rule. While this seems like a trivial task, it is actually more complex than it seems because we would have to figure out the optimal order those rules should be included within the given user constraints.
3. While this is not particularly concerning for the time being, the performance of the Concrete Syntax Tree generation can be improved by making the algorithm implementation multi-threaded so that multiple child nodes can be built asynchronously.
4. More languages should be provided automatically from the web user interface. Right now only 3 languages are provided to the user (with the addition of the “Custom language” which is an example that helps users of the web interface write their own language grammar). These new languages shall be decided depending on the user demands once the Expression Tutor integration is deployed.
5. An interesting feature that could improve usability is the possibility for the user to provide “pools” of values that the generator can use instead of generating completely random number and strings, making the generated expressions easier for students to approach. While this is technically already doable by editing the grammar sources by hand, it might be worth investigating means to ease this procedure.

7 Appendix A: example grammars

Simple math grammar

```
grammar Math;

expr: expr ( '+' | '-' ) term
    | term
    ;

term: term ( '*' | '/' ) factor
    | factor
    ;

factor: '(' expr ')'
    | NUMBER
    ;

NUMBER: INT ( '.' [0-9] + )?;

INT: '0'
    | [1-9] [0-9]*
    ;
```

Listing 8. Math.g4

JSON document grammar

```
// Taken from "The Definitive ANTLR 4 Reference" by Terence Parr
// Derived from http://json.org
grammar JSON;

json: value;

obj: '{' pair (',' pair)* '}'
    | '{' '}'
    ;

pair: STRING ':' value;

arr: '[' value (',' value)* ']'
    | '[' ']'
    ;

value: STRING
      | NUMBER
      | obj
      | arr
      | 'true'
      | 'false'
      | 'null'
      ;

STRING: '"' [0-9A-Za-z]* '"';

NUMBER: INT ('.' [0-9] +)?;

INT: '0'
    | [1-9] [0-9]*
    ;
```

Listing 9. JSON.g4

Racket BSL grammar

```
// Derived from https://docs.racket-lang.org/htdp-langs/beginner.html
grammar BSL;

program: (defOrExpr '\n')+;

defOrExpr: definition
  | expr
  ;

definition: '(' 'define' ' ' '(' NAME (' ' VARIABLE)+ ')' expr ')'
  | '(' 'define' ' ' NAME ' ' expr ')'
  | '(' 'define' ' ' NAME '(' 'lambda' '(' VARIABLE (' ' VARIABLE)+ ')' expr ')' ')'
  | '(' 'define-struct' ' ' NAME '(' (' ' NAME)+ ')'
  ;

expr: '(' NAME ' ' expr (' ' expr)+ ')'
  | '(' 'cond' (' ' '[' expr ' ' expr ']' )+ ')'
  | '(' 'cond' (' ' '[' expr ' ' expr ']' )+ ' ' '[' 'else' ' expr ']' ')'
  | '(' 'if' ' ' expr ' ' expr ' ' expr ')'
  | '(' 'and' ' ' expr ' ' expr (' ' expr)+ ')'
  | '(' 'or' ' ' expr ' ' expr (' ' expr)+ ')'
  | NAME
  | NUMBER
  | BOOLEAN
  | STRING
  | CHARACTER
  ;

NAME: [a-z] ([A-Za-z0-9] | '_' | '-')*;

VARIABLE: [A-Z] ([A-Za-z0-9] | '_' | '-')*;

NUMBER: INT
  | INT '.' [0-9]* [1-9]
  | INT '/' INT
  ;

INT: '0'
  | [1-9] [0-9]*
  ;

BOOLEAN: '#true'
  | '#T'
  | '#t'
  | '#false'
  | '#F'
  | '#f'
  ;

STRING: '"' ([ -!#-~])* '"';

CHARACTER: '#' [A-Za-z0-9]
  | '#space'
  ;
```

Listing 10. BSL.g4

8 Bibliography

References

- [1] Checkstyle contributors. Checkstyle. <https://checkstyle.sourceforge.io/>, 2021. [Online; accessed 6-June-2021].
- [2] N. Chomsky and D. Lightfoot. *Syntactic Structures*. De Gruyter Reference Global. Mouton de Gruyter, 2002.
- [3] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Notices*, 35(9):268–279, Sept. 2000.
- [4] Coda Hale, Yammer Inc., Dropwizard Team. Dropwizard. <https://www.dropwizard.io/>, 2021. [Online; accessed 6-June-2021].
- [5] Google Inc. Bazel. <https://bazel.build/>, 2021. [Online; accessed 6-June-2021].
- [6] Google Inc. Error-prone. <https://errorprone.info/>, 2021. [Online; accessed 6-June-2021].
- [7] L. Kats, E. Visser, and G. Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *Pure and Declarative Syntax Definition: Paradise Lost and Regained*, pages 918–932, 01 2010.
- [8] M. E. O’Neill. PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, Sept. 2014.
- [9] T. Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, LLC, 2012. Retrieved 2021-06-22.
- [10] T. Parr and K. Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *LL(*): The Foundation of the ANTLR Parser Generator*, 06 2011. Retrieved 2021-06-22.
- [11] PMD contributors. PMD: An Extensible Cross-Language Static Code Analyzer. <https://pmd.github.io/>, 2021. [Online; accessed 6-June-2021].
- [12] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. Fuzzing with Grammars. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-12-21.