POLITECNICO DI MILANO

Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

A framework for generating random Java programs with Prolog knowledge base validation

Supervisor: Prof. Marco Domenico Santambrogio

M.Sc. Thesis by:
Giovanni Agugini Bassi
id number 877101

Academic year 2017-2018

A framework for generating random Java programs with Prolog knowledge base validation

Giovanni Agugini Bassi

Abstract

Compilers and *pRogram Analysis and Testing tool (RAT)* are software programs whose correctness affects in a significant way the quality and the performance of the software implemented in any production sector. Compilers are primarily essential for code design and implementation in terms of run time performance, reliability and even security of an application[41]. Thus, the process of testing compilers and *RAT* tools for discovering vulnerabilities and anomalies is in the best interest for any organization whose software is a core asset. Various software and academic organizations have approached with different methodologies the problem of testing compilers. A possible technique consists of applying formal methods, which are good for expressing constraints but do not *a priori* guarantee correctness [14]. Moreover, it is difficult to formalize a model and avoid ambiguity when using formal methods. Another possible approach is random testing, or fuzzing, which refers to the process of generating random programs as benchmarks ([43], [36], [33]). Random testing has emerged as an important tool for finding bugs in compilers[12]. In this work, we propose ProGen, a framework for generating random Java programs with Prolog knowledge base validation. ProGen aims at combining both previously mentioned methods; a Prolog formal specification of the Java Language Specification [26] validates the work of an automated program generator. The approach of the framework is to construct an abstract syntax tree from the BNF specification of a predefined subset of the Java grammar and to validate the nodes of the AST to be consistent with the Java Language Specification. Thus, the Prolog knowledge base models some rules of the language specification, enabling the random generator to prune incorrect sub-trees of the AST and to output a compilable program. The results from the application of this framework are syntactically correct but semantically meaningless programs in a subset of Java. We have used Scala as our programming language for allowing greater testability of the application and avoiding side effects in the application. Finally, we let the user specify parameters related to the generation properties, for example, the number of classes and the primitive types, to let the application be highly customizable and configurable.

A framework for generating random Java programs with Prolog knowledge base validation

Giovanni Agugini Bassi

Sommario

I compilatori e i software di analisi (*pRogram Analysis and Testing (RAT)*) sono programmi software la cui correttezza influenza in modo significativo la qualità e le prestazioni del software implementato in qualsiasi settore di produzione. I compilatori sono primariamente essenziali per la progettazione e l'implementazione del codice in termini di prestazioni, affidabilità e sicurezza di un'applicazione [41]. Pertanto, testare i compilatori e i software *RAT* per scoprire vulnerabilità e anomalie è nell'interesse di tutte le organizzazioni il cui software è una risorsa fondamentale. Svariate organizzazioni commerciali e accademiche hanno affrontato con diverse metodologie il problema di testare i compilatori. Un approccio consiste nell'applicazione dei metodi formali, i quali si rivelano buoni per esprimere vincoli su modelli ma non garantiscono *a priori* la correttezza [14]. Inoltre, è difficile formalizzare un modello o una specifica ed evitare ambiguità nella descrizione del modello. Un altro approccio possibile è il random testing, o fuzzing, che si riferisce al processo di generazione di programmi casuali come benchmark ([43], [36], [33]). I fuzzers sono emersi come uno strumento importante per trovare bug nei compilatori [12]. In questo lavoro, viene proposto ProGen, un framework per generare programmi casuali in Java attraverso la convalida con una knowledge base in Prolog. ProGen mira a combinare entrambi i metodi precedentemente menzionati; una specifica formale Prolog della Java Language Specification [26] convalida il lavoro di un generatore di programmi automatici. L'approccio del framework è di costruire un albero astratto di sintassi a partire dalla specifica BNF di un sottoinsieme predefinito della grammatica Java e di convalidare i nodi dell'AST per essere coerenti con la specifica del linguaggio Java. Pertanto, la knowledge base in Prolog elabora alcune regole della specifica del linguaggio, consentendo al generatore casuale di eliminare i sotto-alberi non corretti dell'AST e di generare un programma compilabile. Il risultato dall'applicazione di questo framework è una serie di programmi in un sottoinsieme di Java sintatticamente corretti e semanticamente privi di significato. Abbiamo usato Scala come linguaggio di programmazione per consentire una maggiore testabilità dell'applicazione ed evitare il più possibile *side-effects*. Infine, viene offerta all'utilizzatore del framework la possibilità di specificare alcuni parametri relativi alle proprietà di generazione, ad esempio il numero di classi e i tipi primitivi, per consentire all'applicazione di essere altamente personalizzabile e configurabile.

Dedication

To my parents Paolo e Valeria, who are no longer of this world but live in my daily present. Your love and example guided me throughout all of this journey, and I hope I made you proud of me.
To my sister Giulia and my brothers Carlo and Pietro. Whatever will be our paths in the future, we will always share the strength that made us fight and go forward.
To my friends Eddi, Giuse, Deggi, Shubi. You are part of my family.
To my grandmothers nonna Dona and nonna Titti, my cousin Paolo, my aunt Stefania, my cousin Matteo. Every one of you has a different and compelling reason to be strongly loved.
Last but not least, to all the people who live in my heart.

Acknowledgements

I would like to express my sincere gratitude to my advisor professor Marco Domenico Santambrogio for the continuous support of my MSc study and related research, for his patience, motivation, and immense knowledge. A special thanks to my family, without whom I would not have had the opportunity to study abroad, whose support and encouragement have always pushed me to strive for greatness. Last but not least, I would like to thank all of my friends, with whom I have had the immense pleasure to share this wonderful experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied."

Noam Chomsky

The generation of large random programs has always been significant in the software industry for a primary reason: they provide a useful tool for testing compilers and software analysis tools[24]. Automated testing can dramatically curtail the cost of software maintenance and development[8]. This work aims to present a framework for the generation of large random programs whose objectives are to employ validation against the language specification (notably, the Java Language Specification) and to be as general as possible so that new languages can be easily plugged into the framework. The application resulting from this framework takes as input a Java BNF grammar and a set of configuration parameters. The output is a syntactically correct but semantically meaningless Java program. It is important to remark that the application is implemented in Scala following as much as possible a functional approach. The result of this work is a framework that provides random Java programs with the possibility for the user to configure entry parameters and specify the number of the entities generated. This chapter contains a section that provides the rationale for this work. In chapter 2 we present similar works that provide random program generators and frameworks for automated testing, and we compare those works to ProGen. Chapter 3 is devoted to explaining the scope and the context of this work from a theoretical perspective. In chapter 4 we present our solution, ProGen. In chapter 5 we summarize the concept of validating the generation through the interaction with the Prolog system, and we give a translation table between Prolog rules and The Java Language Specification. Finally, in chapter 6 we offer the results in term of generated programs, we comment and evaluate them, and ultimately we propose a discussion about future work.

1.1 Motivation

Compilers are fundamental software blocks which translate high-level code to low-level or machine-level code. Issues or deviations from normal execution in compilers may lead to high costs in bug finding and software redesign. Moreover, compiler performance is a vital parameter for evaluating the quality and overall performance of a given programming language. For this reason, compilers represent one of the most delicate pieces of software to produce. They are big, some of them consisting of several million lines of code. For example in 2015 GCC, a multi-compiler system mainly used for the C language consisted of approximately 15 million of lines of codes [2]. LLVM, a collection of compilers targeting several conventional programming languages, consisted of nearly 2.5 million of lines of code in 2013 [4]. Due to their size and complexity, it is very likely to produce bugs, even if they are reviewed and tested by highly skilled developers. It is also possible to take advantage of compilers bugs to elevate user privileges, bypass authentication mechanisms or steal information[30]. On the other hand, *pRogram Analysis and Testing* (RAT) tools are essential applications for program analysis tasks such as profiling, evaluating performance, and detecting bugs[31].

1.1.1 Availability of large-size benchmarks

Large-size benchmarks for evaluating compilers and RAT tools are difficult to obtain for mainly two reasons: first, the necessity to collect long sized benchmarks that are different in terms of usage of language structures collides with the reduced availability of them in the software industry. Indeed, software companies often treat their software as a precious asset without sharing its content externally. Second, creating benchmark applications from scratch requires much effort, and the risk of human errors has to be seriously taken[24]. Therefore, the lack of high reproducibility of results and the different biases introduced by programmers leads to the need for a large program generator that can be useful to avoid errors or side-effects in software[24].

1.1.2 Combination of language constructs and atypical code

Moreover, it is of particular interest exploring a wide range of combinations in the language to detect pitfalls in the compiling phase or to measure the compiler performance against a mix of language constructs. In fact, with this framework, it is possible to generate an extensive variety of benchmarks, as a result of using configuration parameters that can be tuned by the user. This variety could sometimes lead to the generation of atypical code, which may be seen useless as a test case. However, this represents a fallacy, as generating unusual benchmarks may detect pitfalls that can be dangerous for complex systems[43].
In addition to that, there is also the need to provide a tool that uses the grammar without relying too much on the idiosyncrasies of the language that this grammar represents.

1.1.3 Towards a language-independent generation

The direction of this approach is to show how language idiosyncrasies can be minimized to test different compilers of different programming languages. Our framework addresses this issue by avoiding as much as possible language-specific encodings in the generation of method and constructor bodies. The second phase of the generation with the construction of the abstract syntax tree is evidence of this. Moreover, the usage of the Prolog knowledge base for validating the correctness represents an innovative and unique approach that has never used before.

Chapter 2

Related Work

The main source of our investigation has been the article describing RUGRAT, a *Random Utility Program Generator for pRogram Analysis and Testing* for the generation of large benchmarks under a predefined set of properties and a specified set of constraints. [24]. The goal of our proposed work is identical to RUGRAT, while the proposed approach deviates from it. As RUGRAT, we aim at offering the possibility to configure the program generation to have both synthetic and real-world application benchmarks [24]. Our work is an attempt to improve some of the features of RUGRAT, although the workflow is very similar and the goal of testing RAT tools and compilers is the same. RUGRAT is grammar-based, but the grammar is encoded into the application. Our work instead takes as input a BNF grammar and parses it to create an internal structure. This approach leads to a greater generality, as every programming language's BNF grammar can be parsed and transformed into a grammar graph by our component. RUGRAT uses symbol tables and sets for verifying the correctness of the generated programs. This framework uses both this approach and a validation approach by sending queries to a Prolog Knowledge Base. Moreover, this formal method gives us more flexibility in the application and enforces the proof of correctness in the generated programs. Our work also benefits from the functional design which leads to greater reusability and testability. To summarize, the usage of the Prolog knowledge base brings our work to a better separation of concerns, and the parsing of the grammar adds more generalization to the approach. However, in terms of generalization, the entire Prolog KB, the interface of the application to the KB and the first generation phase are still language-specific. As regards the Prolog validation approach, [19] uses the same method for verifying source code in a file system. The method proposed validates code with Prolog rules for C/C++ software. Moreover, it models in Prolog an abstract syntax tree as we do in our work. Our AST, however, is translated into Prolog facts while in the case of [19] the nodes of the AST are directly translated from the source code to be verified. The work aims at verifying a specific system as proof of the validity of the method. Our method instead is meant to offer a more general framework to be used for generating programming languages instances. Regarding the usage of Prolog for examining Abstract Syntax Tree a work from R. Crew [15] defines ASTLOG, a domain specific language for exploring AST in Prolog. While our work takes in input a BNF grammar for generating programs, there are other techniques used for the same purpose. [40] produces very realistic workloads for hard-drives with a technique of composing different individual workloads to match probability distributions. In the following section we focus our search on the related field of grammar-based test input generation, pioneered by Hanford[22] and Purdom[35] in the 1970s. It is possible to find additional resources in a survey article on program generation techniques for testing compilers [9].

2.1 Probabilistic grammar-based generation

Most of earlier works using probabilistic grammar-based generation [34, 32, 38, 44, 10, 37, 43, 16] are mainly related to testing and debugging. We may describe these works as the generation of a collection of micro test cases, while our approach provides end-to-end benchmarking and flexibility on program configuration parameters. Murali and Shyamasundar present a generator based on the programming language grammar which targets the compiler of a subset of Pascal[34]. Maurer [32] presents the Data-Generation Language which supports different actions for functional testing of VLSI circuits. A probabilistic language-based generator which generates Fortran language is presented by Burges[10], in which the user can specify the Fortran syntactic rules in an attribute grammar. There is also the possibility to assign weights and probabilities to the different productions to control how many times the language features appear in the generated program. This work produces programs up to 4KLOC, compared to up to 2MLOC of our work. A production grammar is used by Sirer and Bershad[38] for enhancing the context-free grammar with probabilities and actions. The generated programs are in bytecode for testing the Java Virtual Machine, and they can produce up to 60kLOC, while our work produces Java source code programs with up to 2MLOC. Another work that takes our attention is CSmith[43], a generation tool for the C language. CSmith is a C programs generator for finding anomalies and bugs in C-based complex systems. Based on the hypothesis that greater expressiveness brings broader bug discovery, the authors implemented a tool which is rich in language features. Grammar-based, it generates random programs exploring atypical code combinations. In the same sense, our work explores different code combinations thanks to the configuration parameter setting and the randomly distributed generation. As RUGRAT, CSmith encodes grammar rules into the application instead of taking a grammar as an input. Global and local environments are used by CSmith to keep track of the productions used and the defined types and methods. In our work, we make use of a global table with information on classes and their relations, interfaces, method, and constructor signatures. However, in our case the construction of an abstract syntax tree represents our execution trace, as every added node to the tree represents a symbol of the grammar. Others have used CSmith, for example, used it for generating static analyzers and found that CSmith contains some bugs, for instance, generation of dangling pointers and assignments to the same memory location. CSmith has also been used by Cuoq et al.[16] for testing static analyzers. They found 50 bugs by testing C-Frama, a framework for C programs transformation and analysis. In the domain of Java, Yoshikawa et al. implemented a random program generator [44] for testing JIT (just-in-time) compilers. This tool, as [24] states, does not generate large programs.

2.2 Combinatorial grammar-based generation

Other than probabilistic grammar coverage, there is another method to produce programs for testing compilers, which is the combinatorial grammar-based generation. Purdom's algorithm[35] is the first well-known example of this approach: Purdom generates sentences which cover all the production rules of a context-free grammar. The problem of this algorithm is that it generates small test cases; in fact, it aims to generate the shortest sentence covering the productions. Celentano et al. adopted usage of multi-level grammars for supporting correctness rules in modern programming languages[11]. Purdom algorithm has been implemented by Boujarwah et al. for a subset of Java, but no experimental results have been published. A test data generator called *Geno* for general purpose languages is proposed by Lämmel and Schulte[27]. *Geno* lets the user define combinatorial coverage for the language rules, and it provides correctness although not at the level required for Java-like languages. Harm and Lämmel provides an extension of the test case generation from the systematic coverage of context-free grammars to the systematic coverage of attribute grammars[23]. However, it is not clear how to scale the technique, and this approach does not show a test on Java-like programming languages. Hoffman al. developed *YouGen* a generator tool based on context-free grammars which employ tags for restricting the generation.

2.3 Exhaustive program generation

The technique of exhaustive program generation consists of exhaustively computing all possible programs up to a certain size. Coppit and Lian present *yagg*, a generator for test case generators which exhaustively computes all the possible test cases up to a certain size. *Yagg* works with context-free grammars and gives the possibility to express semantic rules. Daniel et al. describe *ASTGen* [17] which provides a systematic generation of small Java programs. A requirement for ASTGen is to combine multiple generators, and more importantly, some generated programs produced have compile-time errors and present a simple structure. Majumdar and Xu present a directed test data generation technique. The technique grounds on the exhaustive derivation of all possible programs up to a given size and the symbolic or concolic execution of these programs as inputs to the programs under test. The related tool, *CESE*, produces small size test programs while our work produces large programs which not depend on a particular program under test.

2.4 Model-based generation

Grammars are not the only means by which a language can be defined. The model-based approach exploits the semantic richness of language models to generate programs. These generators are usually slower and produce small size programs targeted to specific features. Zhao et al.[45] describe a temporal logic model which seizes the behavior of compiler optimization applications. It is still not clear how this approach scales and how it can be used for application benchmarks generation. Soeken et al.[39] describe the problem of generating test programs in terms of a model finding application therefore demonstrating the generality of these methods.

This approach indeed generates programs by finding models that fit into a meta model. However, the technique limits the possibility to configure the generation in a detailed fashion as in our work.

Chapter 3

Theoretical Background

The theoretical background of this work lies upon at the intersection of various computer science areas such as compiler theory, the Java language with its syntactic and semantic rules, logic programming in Prolog and functional design. Knowing the structure of a compiler is relevant for this work for two main reasons: first of all, because compilers are the software products to be tested and understanding it can give the reader a broader view of the complexity and the structure of this software component. Secondly, the way that our framework generates a program has a strict relation to the implementation of some compiler components. In the next sections, we address differences and similarities between compiler components and the framework approach as we cover the theoretical aspects. The Java Language is the language we generate with the framework, and its semantics and syntax are described in the next sections, as long as its multithreading feature that allows maximizing the use of CPU when some of the computational logic can be parallelized. As regards logic programming, we want to describe the paradigm which constitutes the Prolog language and permits to build a knowledge base to verify a model formally.

3.1 Compilers and BNF grammars

In this section, we describe the most relevant concepts of Compiler Theory, and we highlight the similarities with related software components in our application. We also explain Context-Free Grammars and the Backus Naur/Normal Form notation which we employ as our grammar notation.

3.1.1 Compiler definition

A compiler is a program which takes as input a program, often called source code, and translates it into an equivalent program, often called target code[6]. Usually, the target program is an executable machine-language program. The interaction among different software components constitutes a compiler.

3.1.2 Lexical analyzer

The *lexical analysis* or *scanning* is the first phase of the compilation process. The work of a *lexical analyzer* is to ingest the source code, to transform the characters encountered into tokens, which are defined by regular expressions, and often to delete special characters like for example white spaces and comments.

3.1.3 Syntax analyzer: Parser

The second phase of the compiler is the *syntax analysis*, often called *parsing*. The parser takes the tokens one-by-one and tries to construct the parse tree following the rules of the grammar of the source language. If some sequences of tokens are not following the productions of the grammar, the parser signals a compiler error. Our generator works similarly to a parser: it employs a context-free grammar and constructs an abstract syntax tree. The difference stays in the fact that our generator constructs the tree from the grammar choosing every time a production from the grammar, while the parser builds the tree from a sequence of tokens given in input.

3.1.4 Semantic analyzer

The third phase is what is called *semantic analysis*, which uses the parse tree and the symbol table. The aim of this phase is to test whether the program complies with the semantic constraints of the language. *Type checking* is a primary task for this phase; by using the information stored in the symbol table, the compiler checks whether each node in the parse tree has the correct type. In our work, a similarity with this phase is encountered in the Prolog engine which our application calls every time a production is selected. The Prolog rules in the KB returns either true or false, accordingly to the Java Language Specification.

3.1.5 Intermediate code generation

When translating the source code into the target code, the compiler creates from the parse tree one or more intermediate representations of the program. It is important that the intermediate language is easy to convert from the source language and to the target langage. This phase is called *intermediate code generation*.

3.1.6 Code optimization

Then we have the *code optimization phase*, where the given component optimizes its input and output an intermediate code representation to produce a better target code and to make the computer consume fewer resources in terms of CPU and memory.

3.1.7 Code generation

Code generation is the last phase of the compiler. This component takes as input the program in the form of the intermediate language and generates the final result in the target language.

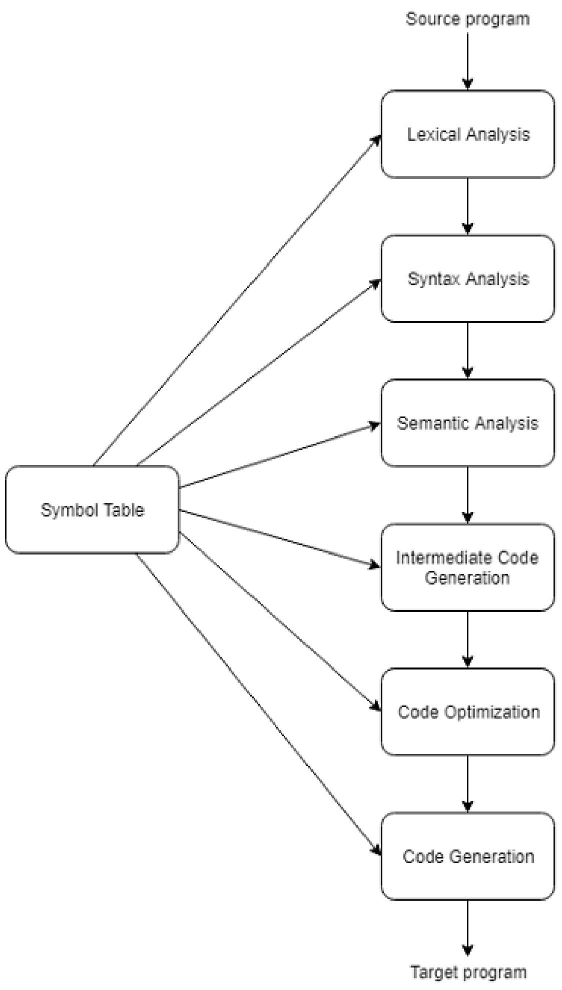


Figure 3.1: The phases of a compiler.

3.1.8 Symbol table

The *symbol table* represents a data structure managed by the compiler and consists of all the identifiers name along with their types. It helps the compiler to function smoothly by finding the identifiers quickly. Figure 3.1 shows the structure of the compiler, in which the symbol table has a link to every other component. During lexical analysis, the symbol table creates entries about tokens. In syntax analysis, the compiler adds all the information in the symbol table, and during the other phases, the symbol table is referred to get the information. In our application, each generated class has a table which contains all the class members, i.e., fields, constructor signatures, interfaces implemented, method signatures, superclass if applicable. The table provides information to the software components that generate the program, mainly for constructors, method invocations and field accesses.

3.1.9 Context-free grammars

A context-free grammar is a particular type of formal grammar in which a derivation expresses every rule from a left symbol to one or more right symbols. A context-free grammar presents four main elements:[6]:

1. *Production rules*, where each rule contains a left part called *head* consisting of a non-terminal symbol, a separator symbol, and a right part or *body* of the rule consisting of a sequence of terminal or non-terminals or both.
2. *Terminal* symbols, often named “tokens.” They are the primary elements of the grammar.
3. *Non-terminal* symbols, often named “syntactic variables”. Each non-terminal is described by a derivation to a group of terminal symbols.
4. A *start* symbol from which the derivation can start, consistin of a non-terminal symbol chosen among the available ones.

3.1.10 BNF notation

The BNF notation (Backus-Naur Form or Backus Normal Form) is a notation system for context-free grammars. Computer scientists often use it as a syntax description of programming languages. Every rule in Backus-Naur form has the following structure[3]:

$$\textit{non-terminal} ::= \textit{expansion}$$

The symbol *::=* means *is defined as.* Angle brackets ⟨ ⟩ surround every non-terminal in BNF, no matter if the non-terminal is on the left or right part of the production. An *expansion* refers to the right part of a rule and it can be of two different types: *sequence* or *choice.* If the symbols are connected without a vertical bar, the expansion is of the *sequence* type. If the symbols are instead separated by a vertical bar, the expansion is of the *choice* type, thus meaning that the expansion must follow one among the possible alternatives of symbol sequences[3]. In Figure 3.2, it is shown an example of a BNF grammar for the standard arithmetic expression grammar. When generating a program by just following the grammar rules, we are not sure that the program passes the compilation phase, as the choice of a production also depends on semantic constraints. Indeed, many programming languages have semantic rules which limit the possibility to explore a production given the previously selected productions. In the case of the Java language, these semantic constraints are listed and deeply analyzed in the Java Language Specification[26]. The idea of a Prolog knowledge base to validates the generation comes from here. A knowledge base is a type of database that stores structured and unstructured data which models the knowledge of a given world. The technology consists of facts and rules about the world and an engine that makes logical inferences from those facts and rules to deduce new facts or highlight inconsistencies. We want that our knowledge base models the Java Language Specification with its constraints to validate our production rules in the generation.

```
<expr> ::= <term> "+" <expr>
         |  <term>

<term> ::= <factor> "*" <term>
         |  <factor>

<factor> ::= "(" <expr> ")"
           |  <const>

<const> ::= integer
```

Figure 3.2: Simple Arithmetic BNF Grammar

3.2 Prolog and logic programming

Prolog (PROgramming in LOGic) is a declarative logic programming language invented by Alain Colmerauer and Robert Kowalski in 1972. The authors aimed to make a programming language that permits explicit logic expression instead of specifying procedures on the computer. Artificial Intelligence applications heavily employ Prolog for creating Knowledge Bases.

3.2.1 Data types

Prolog's single data types are *terms*, which can be *atoms, variables, numbers* and *compound terms.* An atom is a simple name without connotation, for example *y, purple, 'Bread'*, and *'any atom'.* A number can be float or integer. A variable consists of letters, numbers and underscores characters; its first letter is capitalized or is an underscore, for example *A and _hello.* A compound term consists of an atom called a *functor* (or predicate) and some terms, which are called arguments. Compound terms are written as a predicate followed by a list of arguments separated by a comma and contained in parentheses. An example of compound term is *father(bob,alice).* The number of arguments is called the term's arity. It is common to express the predicate with a / followed by its arity, e.g. *father/2* meaning that the compound term has arity 2.

3.2.2 Facts, rules and queries

Prolog is made up of facts and rules[1]. Facts describe the nature of the object. Rules are properties that can be derived from the relations between objects. Prolog expresses the relations between objects with clauses, that are of the form: *Head :- Body.* This reads as *Head is true if Body is true.* Clauses with empty bodies are facts, while clauses with bodies are rules. For writing rich logic expressions, two important built-in predicates are the conjunction (logic AND) predicate which is expressed by a comma and the disjunction (logic OR) predicate which translates into a semicolon. Table 3.1 and Table 3.2 show the truth tables for the logic AND and logic OR as Prolog predicates. A collection of facts and rules consists of a knowledge base. Due to the conversational nature of Prolog, the user can perform queries to the knowledge base and obtain either a true or false response.

A	B	A , B
false	false	false
false	true	false
true	false	false
true	true	true

Table 3.1: And Prolog predicate

A	B	A ; B
false	false	false
false	true	true
true	false	true
true	true	true

Table 3.2: Or Prolog Predicate

3.2.3 How Prolog works

A Prolog program is executed by searching the tree of choices and determining the instances which satisfy the provided rules. When the engine does not find an answer in a subtree, Prolog backtracks to go to another branch. The program execution does not terminate once an answer is found but keeps on looking for other evaluations until the tree is wholly visited. For this work, it seemed very reasonable to us exploiting all the properties of Prolog for modeling the Java Language Specification. We defined as terms the entities of the language, and we expressed the rules that describe the relations between those entities. For example, the inheritance between classes is validated by Prolog with the rule *canExtend(Class1,Class2)* that checks if there is not a class circularity error, i.e., there is not a cycle of inheritances, and that *Class1* is not the same class as *Class2*. If the rule is validated, a new fact *extends(Class1,Class2)* is added to the KB . Noticeably, the predicates used by the rules represent the relations between the entities, which are represented by the terms.

3.3 The Java language

The Java language, created in the mid-'90s and distributed by Oracle Corporation, is a general-purpose, object-oriented programming language.[21]. It is a general-purpose language because it has a wide variety of application domains. Java is said to be concurrent because the Java platform offers basic support for concurrency and usage of threads. Finally, the language is class-based because it grounds its implementation on classes of objects and the relations between them. Similarly, it is object-oriented because of the importance of the concept of objects, which are dynamically saved on the heap and may contain data (i.e., fields) and code (i.e., methods).

3.3.1 Influence and characteristics

According to the TIOBE index [5], Java is the most popular programming language in the world. The reasons why Java is so popular are because it is portable, thanks to the platform-agnostic Java Virtual Machine, scalable and benefits from a large community of users. The object-oriented paradigm that Java possesses is another reason for the popularity of the language; indeed developers may take advantage of the concepts of encapsulation, inheritance, composition, and polymorphism for describing the computation. The Java language is a strongly and statically typed programming language. Thus, the majority of typing constraints are forced during the compilation phase.

3.3.2 The Java Language Specification

The Java language is meticulously described by the Java Language Specification[26], which is a programmatic document that provides developers a technical reference for understanding the semantics of the Java environment. The specification makes a distinction between *compile-time* errors that are detected during compilation and *run-time* errors that occur at run-time. When writing the Prolog rules contained in the knowledge base we targeted the rules whose violation cause compile-time errors. However, run-time errors sometimes are causing the program not to be executed. Thus, we put attention to some run-time errors, for instance, the *ClassCircularity-Error* caused by a cycle of inheritances between classes. Since one of our targets is the Java compiler, we want to highlight some characteristic traits of the Java language, especially in the process of compilation, interpretation, and execution.

3.3.3 Compilation and interpretation

In the Java environment, compilation and interpretation are combined. The compiler takes a Java source program and converts it into an intermediate form named bytecodes. After the compilation, the *bytecodes* are interpreted by the Java Virtual Machine. This arrangement leads to the portability of *bytecodes* on every system with a JVM, even though interpretation adds a performance overhead. To overcome the performance overhead of interpretation, *JIT* (just-in-time) compilers that compile *bytecodes* at run-time have been created[6]. However, the purpose of this work is to create a benchmark generator for Java-to-bytecodes compilers. Thus *JIT* compilers are out of our scope. After the interpretation, the machine-level instructions are executed by the processor of the computer containing the JVM. Interpreters are another type of language translator which appear to directly execute the instructions defined in the source code instead of translating a source program and outputting a target program.

3.3.4 Concurrency and multithreading

In this work, we exploit the advantages of concurrency for making the computation faster. Concurrency is the capability of different sections of a program to be run out-of-order or in a partial order, without altering the final result. More technically, concurrency is related to the decomposition of an algorithm, a problem or a program into units or components that are order-independent or partially ordered [28]. The generation of an object-oriented Java program is a process in which software modules like classes have to share their information with the other classes, given that their accessibility is public. For example, a class A with an instance b of a class B may need to know what methods are defined in class B in order to call methods on object b. This need makes the idea of implementing a multithreading approach difficult at first glance, as assigning a thread for each class would be challenging to handle because of the dependencies between threads. The idea of this approach is to generate at first all the information that requires the shared knowledge between classes, i.e., reference types (class names and interfaces), class fields, constructor and method signatures for every class. Then, with such data at disposal, a multithreading approach can be implemented. In the Java execution environment, upon which the Scala language lies, there exist two primary units of execution: *processes* and *threads*.

Defining and starting threads with executors

Creating an instance of a thread implies providing the code for running that thread. There exist two manners of doing that: creating a subclass of the Java class *Thread* or providing a *Runnable* object, where the interface *Runnable* defines a single method *run*, where there are the instructions executed by the thread. In our application the second way to define a thread has been approached, thus creating a *Runnable* object for each task we want to be performed by a different thread. Managing threads in applications can lead to a worse design and poor readability. Thus, *executors* are used for solving the problem, because they are objects that manage the creation and management of threads.

3.4 Scala and functional programming

In the proposed work, we had to choose the language to use for implementing the application and the design paradigm to adopt. We ended up using Scala as our working language and the functional paradigm as our design approach.

3.4.1 Scala origins

Martin Odersky has invented the Scala language at EPFL (École Polytechnique Fédérale de Lausanne) in the first years of the 21st century. Scala runs on the Java Virtual Machine, and it is designed to address some issues of Java. The main features of Scala are its advanced type system (which supports algebraic data types, covariance, higher-order types, and anonymous types), the interoperability with the Java system, a higher level of concurrency and distribution on collections and with asynchronous programming and finally the most important one: functional programming.

3.4.2 Functional programming

Functional programming naturally derives from the lambda calculus, a formal system introduced by Alonzo Church in the 1930s. In pure functional programming, functions are first-class objects: they can be used and passed everywhere, and they don't have side effects. Pure functions take input data, apply a computation on it and return an output data, without modifying the internal state of the data or accessing external resources on the computer. Therefore, data in functional programming are immutable. Using pure functional programming is hugely beneficial because of the increase of modularity that we gain. Because of this, functional programming becomes easy to test, reuse, parallelize and generalize. Moreover, pure functions are much less prone to bugs[13].

3.4.3 Higher-order functions

Another essential concept of functional programming deriving from the fact that functions are first-class objects is the concept of higher-order function. Simply put, a higher-order function is a function that takes as argument a function and returns a function. Examples of higher-order functions are *map, flatMap, foldLeft*. It is of utmost importance the strict relation of pure functional programming and algebra: the operations on the types and the transformations of functions can be seen as algebraic operations between groups and sets and transformations of mathematical functions. Indeed there is a one-to-one correspondence between pure functions in functional programming and mathematical functions.

3.4.4 Considerations

The Scala language is not a pure functional language: it allows side-effects and mutable states. Therefore Scala embraces both object-oriented and functional paradigms, and this could be a reason for confusion because there is not a single computational paradigm and thus the design is not coherent. Our opinion is that this aspect of Scala makes it a versatile language employable in application domains where we can employ both design paradigms. In our work, we use an object-oriented approach when there are data dependencies and when we need mutable values, for example when we want to change weights for the rules. We approach a functional design when passing a state or using high-order functions seems more straightforward, more concise and more prone to be thoroughly tested.

Chapter 4

Solution

In the third chapter, we show our solution for generating large random Java programs with the use of a Prolog knowledge base which models the Java Language Specification. The goal of this work is to design an approach whose inputs are the BNF grammar of a language and a set of configuration parameters representing some constraints, for example the number of code lines, the number of classes, the list of allowed types. The output is a syntactically correct and semantically meaningless program that satisfies the constraints and the grammar rules. In the section 4.1 we describe the architecture of the framework by showing the interaction between the software components. Afterwards, section 4.2 provides an high level description of the application to give an idea of the entire process of generation by considering the components previously presented. Section 4.3 gives the motivation of such solution. The next sections are organized as follows : section 4.4 covers the phase of the generation of program entities (e.g. classes, interfaces, method signatures) and the creation of relationships between them. Section 4.5 is dedicated to grammar parsing, graph creation and filling of the program entities inside the abstract syntax tree. In section 4.6 we explain how a multithreading approach has been employed in this phase for improving performance. In section 4.7 we show how the GT combines the intermediate outputs of ProGen (see 4.2) for filling the AST of the program. In section 4.8 we show how constructor and method bodies are generated through a depth-first traversal algorithm that produces a completed AST and thus a correct program. Finally, in section 5, a deeper explanation of the Prolog rules and the testing phase is described along with details about the implementation of the communication between ProGen and Prolog KB through the gRPC protobuf protocol.

4.1 Architecture and components

To accomplish this task, our approach needs two main components: the application ProGen that generates the programs and the Prolog KB engine that validates the generation.

4.1.1 ProGen component

The Progen component, as showed in Figure 4.1, contains in turn three components that we need for our purpose: a program entities generator (PEG) that produces program entities, a BNF grammar parser (GP) and a grammar traversal engine (GT). For the sake of abbreviation we will use the terms:

- PEG for Program Entities Generator.
- KB for Prolog Knowledge Base.
- GP for Grammar Parser.
- GT for Grammar Traversal Engine.
- AST for Abstract Syntax Tree.

4.1.2 Prolog KB component

The Prolog KB is instead implemented as a server which contains the Prolog rules, adds Prolog facts when requested by ProGen, solves the queries asked by ProGen and sends the responses to ProGen. As shown in the component diagram of Figure 4.1, the ProGen component interacts with the Prolog KB with a message exchanging protocol, that is RPC (Remote Procedure Call). The Prolog KB components contains three components: a gRPC Server that offers the procedures to the client at the ProGen side, a TuPrologHandler which dispatches and translates the Prolog queries and facts to the TuProlog Service, and finally the PrologRules which reads from a file the Prolog rules and provides to the TuPrologHandler an interface to read them.

4.1.3 TuProlog service component

Finally, a third main component is shown in Figure 4.1, the TuProlog Service. It is a third party API used for running the Prolog engine and based on SWI-Prolog, a standard software for Prolog solvers. TuProlog Service receives the information from the Prolog KB about the generated entities and solves the queries performed by the TuPrologHandler.

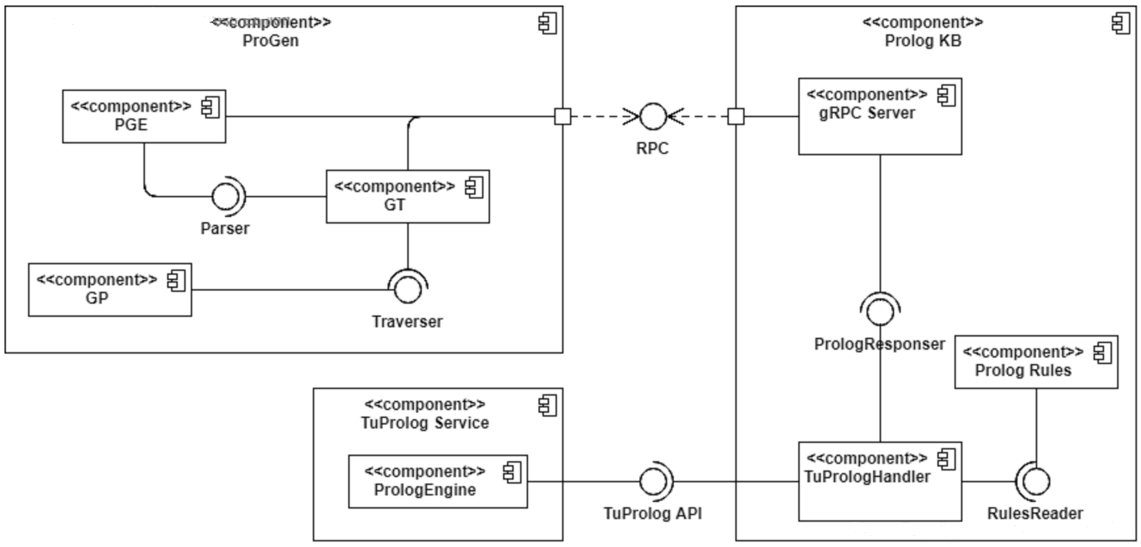


Figure 4.1: Component Diagram of the solution

4.2 Workflow of the framework

Given the definition of the main components in the previous section, we want to present in this section the execution flow of the application. In the next subsections, we will refer to the nodes of Figure 4.2, which captures the execution flow of the application.

4.2.1 Entity generation

The application starts its execution with the Program Entity Generator (node 1 in Figure 4.2) which takes in input configuration parameters (Input 1 in Figure 4.2 about number of classes, interfaces, methods and other information related to the modalities the generation is carried on. The PEG generates partial classes with interfaces, fields, constructor signatures and method signatures along with a global table containing generated interfaces, class names and primitive types. The result of this step is showed in node 2 of Figure 4.2.

Use of Prolog

The PEG queries the KB when certain entities are generated. In Figure 4.2 there is a bidirectional arrow because the PEG adds Prolog facts to the KB when generating new entities. For example, when a method signature is generated, the information is added to the Prolog KB.

4.2.2 Grammar parsing

After the Program Entities Generation, the application passes the control to the GP (node 3 in Figure 4.2). This component takes in input the BNF representation of the grammar (Input 2 in Figure 4.2, parses it and creates an internal representation of the grammar, i.e. a weighted directed graph (node 4 in Figure 4.2).

4.2.3 Filling of partial classes and interfaces

Afterwards, the Grammar Traverser (node 5 in Figure 4.2) creates the abstract syntax tree of the future program by adding the root taken from the graph, i.e. the $< compilationunit >$ node. Node 5 traverses the graph and fills the newly created AST with the nodes corresponding to the results contained in node 2 in Figure 4.2, i.e. the partial classes and the interfaces contained in the global table. At this moment, we have an incomplete abstract syntax tree because the methods and constructors bodies of the generated classes have not been generated yet.

4.2.4 Method and constructor body generation

For each class, node 5 in Figure 4.2 traverses the grammar graph and generates the missing parts of the program, i.e. the bodies of methods and constructors.

Use of Prolog

The Grammar Traverser queries the KB when generating statements and expressions inside the bodies of constructors and methods. The GT component interacts with the Prolog KB also to add facts related to the nodes added to the AST.

4.2.5 Completed AST

When the GT finishes to generate methods and constructors for each class, all the class sub-trees are merged together into the AST of the program, created at a previous stage. The result, node 6 in Figure 4.2, represents a completed AST ready to be translated into source code.

4.2.6 BackTrack and final result

However, even if node 6 represents a correct program, we have to check against one of the previous configuration parameters, e.g. the number of lines of code. This constraint is satisfied either if the AST has a number of LOC greater to the constraint or if the number of LOC is close to the configuration parameter. If node 6 satisfies this constraint, the application translates the AST into source code and terminates. Otherwise, the application backtracks and goes back to node 1 for generating more classes and satisfying the constraint.

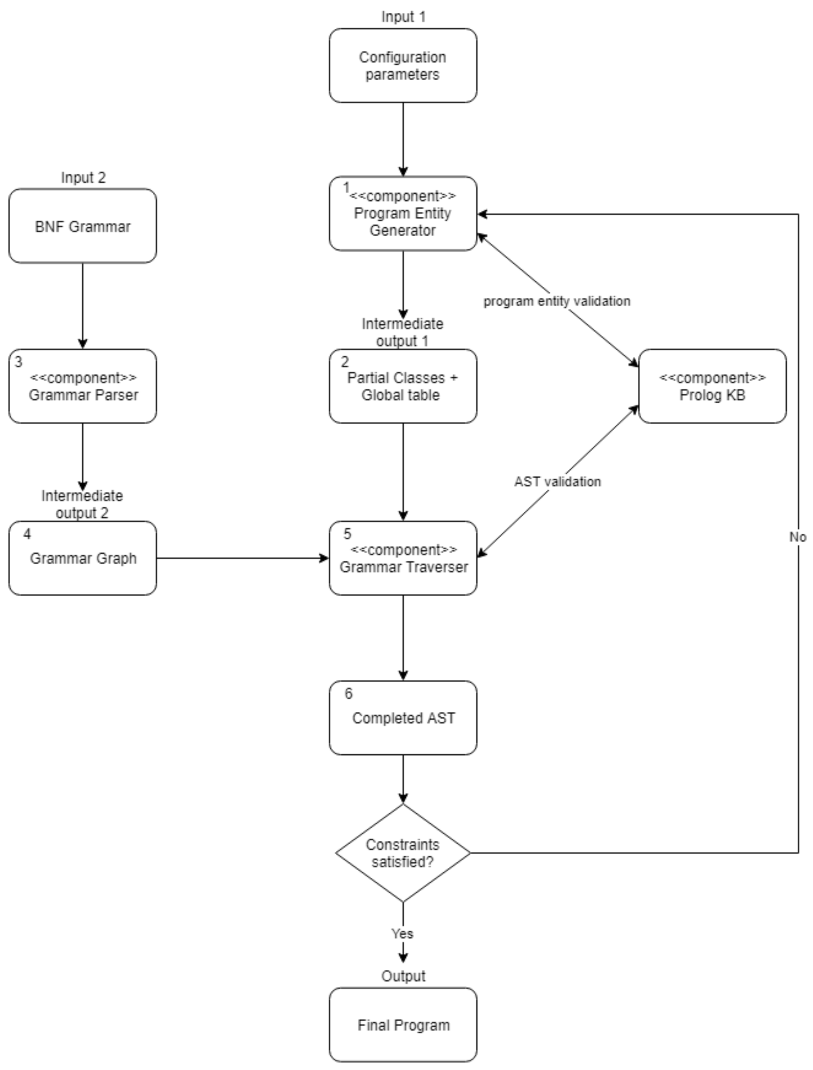


Figure 4.2: Overview of ProGen execution flow

4.3 Rationale

We now want to explain our design choices. We have taken some decisions with the aim to offer a compromise between generality and simplicity. At the beginning of the design process, we have considered two options for generating random programs.

4.3.1 Option A: hard-coding rules

The first option we have considered is to hard-code the grammar rules into the application. This solution consists of creating classes that correspond to the grammar symbols. Option A creates objects based on methods and interfaces that implicitly implement the grammar rules. A good point of this option is that the design is intuitive and simple to reason about, because it is sufficient to define classes based on program entities such as classes, interfaces, methods, statements and then define methods that relate together these classes implementing grammar rules. However, option A does not offer a clear design solution and makes the code poor and difficult to read, because all the grammar rules are spread out in the code without a clear view. Also, it does not seem to be easy to scale the application, as the rules are interconnected and the classes are coupled together.

4.3.2 Option B: traversing the whole BNF grammar

The second option consists of first parsing the BNF grammar into a graph. The graph represents the grammar because every node corresponds to a grammar symbol or to a group of grammar symbols, and node can be either terminal or non-terminal. Then Option B considers traversing the graph from the root, i.e. from the non-terminal node $< compilationunit >$, in order to create an abstract syntax tree. When all the reachable terminal nodes are reached, the traversal ends and the AST is considered completed. Option B offers a high grade of generality because it can be used with a BNF grammar of any programming language. In addition it makes the grammar rules explicit. In this option, the nodes are added without knowing the context of the grammar rule inside the application. However, this lack of context makes very challenging to guarantee the correctness of the generation, because we just know the name of the symbols of the grammar added to the AST in order to ask the Prolog KB if the generation can be validated. Moreover, there are issues related to traversing the entire grammar graph covered in the next paragraphs.

Issues of depth-first traversal of entire grammar graph

In Option B, it is difficult to conceive an algorithm for traversing the grammar graph: if we used a depth-first traversal of the grammar graph, we would incrementally generate classes and thus we would not have many information for generating the first classes. For example, if the traversal starts constructing a class A, the algorithm would first try to generate the methods: inside the methods, we could not use any class type and any class instance creation except for A, because no other classes have been created.

Issues of breadth-first traversal of entire grammar graph

Otherwise, if we used a breadth-first traversal of the grammar graph, we could overcome the problem mentioned above about the lack of information, but we would not be able to check if the symbols of the grammar can be added. Without a depth first algorithm it is not possible to extrapolate all the information for sending queries to Prolog, as we could not consider a completed sub-tree. As an example, let consider the case when we want to generate a method invocation (Figure 4.3): we would start adding to the AST the symbol $< methodname >$ and then the other symbols. Since the symbol $< argumentlist >$ is optional, we should make a choice whether to include it or not in the AST. But the presence of arguments in a method invocation depends on the method chosen, so we would not be able to decide if the $< argumentlist >$ node has to be chosen or not, because the breadth-first traversal does not allow to do so. With depth-first instead, we would be able to make this choice because we would have visited all the sub-tree related to the symbol $< methodname >$ and therefore we would know whether the method invocation requires arguments or not.

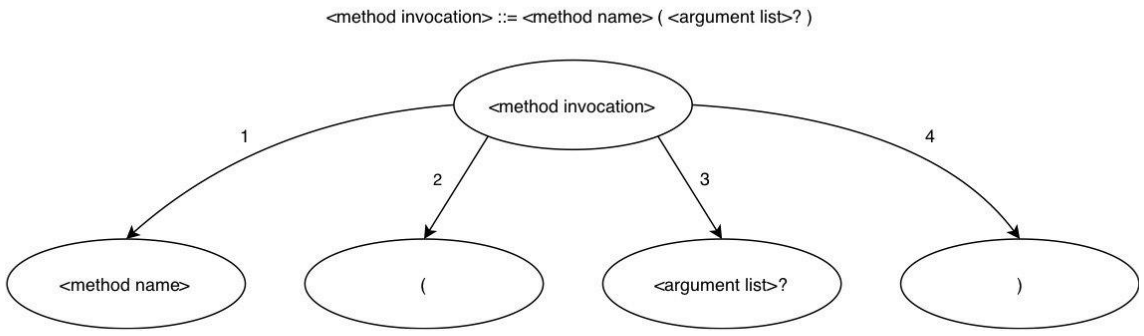


Figure 4.3: The method invocation rule with its graph representation

4.3.3 Combination of Option A and Option B

Our design solution represents the union of the two previous options; we use Option A for generating certain program entities and then we start using the grammar with Option B for generating the remaining parts of the program. Therefore, the PEG is delegated to realize Option A while the GP and the GT are delegated to realize Option B. Option B covers only a part of the grammar, particularly from the $< methodbody >$ and the $< constructorbody >$ nodes/symbols, so that the problem of the lack of information for the classes is avoided. In fact, Option B starts from a partially built AST that has already all the nodes related to the program entities generated by PEG.

Filling and traversal

In Figure 4.4 you can see how this is achieved by ProGen: after the PEG generates the program entities, the AST is constructed from its root, i.e. $< compilationunit >$, and then the information from the PEG is filled in the tree while the GT traverses the grammar graph: as an example, the names of each class are filled with the information from the PEG. Then, the GT stops the traversal when no other information are to be filled, i.e. at the rules $< methodbody >$ and $< constructorbody >$. From this point, Option B is adopted and the GT begins a depth-first traversal of the graph by adding nodes and validating then. Later in this document the traversal algorithm is formalized.

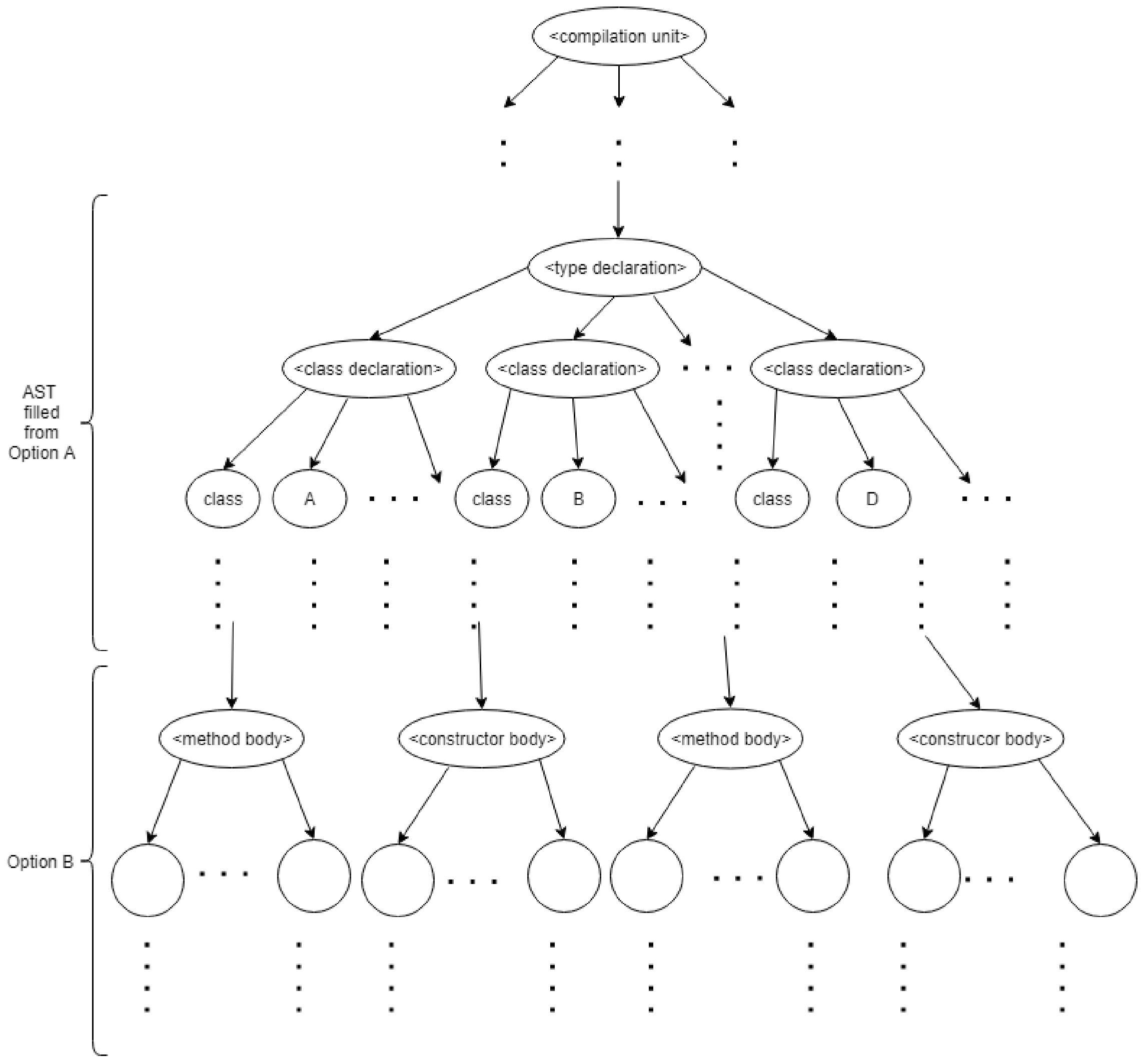


Figure 4.4: AST construction with the use of the two options

4.4 Program entities generation

In this section the first phase of program entities generation is explained. The first step for ProGen is to start generating interfaces, information about classes, i.e. fields, method and constructor signatures, relations between classes. This part is done by creating in the application the instances for every entity generated. Thus, in this phase we create objects of the following classes:

- **MethodSignature**: this class represents a method signature, with attributes *return type*, *method name*, *formal parameters* and *class name.*
- **ConstructorSignature**: this class represents a constructor signature, with attributes *class name* and *formal parameters.*
- **Interface**: this class represents an interface, with attributes *interface name* and *method signatures.*
- **Field**: this class represents a field, with attributes *type* and *name.*

- **Class**: this class represents a Java class, with attributes *name*, *constructor signatures method signatures*, *fields*, and *superclass name.*
- **SymTab**: this class represents the symbol table that keeps all information from a partially generated class.
- **GlobalTable**: this class represent the global table which contains all the generated interfaces and all the possible types to be used by the application.

4.4.1 Structure of the configuration file

The configuration file is what the PEG needs for generating programs with different characteristics. The standard chosen for the configuration is the lightbend typesafe config, a configuration library for JVM languages.

- noOfInterfaces: number of interfaces to be generated, with attribute ‘max’ as upper bound.
- nooOfMethodsPerInterface: number of methods per interface, with attributes ‘max’ as upper bound.
- noOfClasses: lower bound on number of classes in the generation, with attribute ‘max’ as upper bound.
- noOfPackages: number of packages in the generation, with attribute ‘mas’ as upper bound.
- allowedMethodCalls: lower bound on number of allowed method calls, with attribute ‘max’ as upper bound.
- allowMethodChain: ‘yes’ if method invocation chains allowed, ‘no’ otherwise.
- noOfMethodChains: if allowMethod chain is ‘yes’, lower bound on number of method chains with attribute ‘mas’ as upper bound.
- noOfMethodsInTheChain: if allowMethodChain is ‘yes’, lower bound on number of method invocations in a chain with attribute ‘mas’ as upper bound.
- allowMethodComposition: ‘yes’ if method composition is allowed, ‘no’ otherwise.
- noOfMethodCompositions: if allowMethodComposition is ‘yes’, lower bound on number of method compositions with attribute ‘mas’ as upper bound.
- noOfMethodComposedInComposition: if allowMethodComposition is ’yes’, lower bound on number of methods composed in a method composition with attribute ‘max’ as upper bound.
- noOfMethodsComposed: if allowMethodComposition is ‘yes’, lower bound on number of methods composed with attribute ‘max’ as upper bound.
- maxValueForLoop: maximum value of iteration in a loop.
- allowIndirectRecursion: yes if indirect recursion is allowed, no otherwise.

- maxArraySize: maximum number of elements in arrays.
- maxListSize: maximum number of elements in lists.
- noOfInheritanceChains: number of the inheritance chains of classes or interfaces in the generation.
- maxNestedIfs: maximum number of nested if statements in the generation.
- allowArray: yes if array are allowed in the generation, no otherwise.
- noOfParametersPerMethod: lower bound on number of parameters in a method signature with attribute 'max' as upper bound.
- inheritanceDepth: lower bound on number of classes or interfaces. in an inheritance chain with attribute 'max' as upper bound.
- totalLOC: number of lines of code of the generation.
- maxInterfacesToImplement: maximum number of interfaces to implement.
- maxRecursionDepth: maximum recursion depth.
- classNamePrefix: prefix to add to each identifier for a class declaration.
- noOfClassFields: lower bound on number of fields in each class with attribute 'max' as upper bound.
- noOfMethodsPerClass: lower bound on number of methods in each class with attribute 'max' as upper bound.
- distribution: the probability distribution to use for the generation.

4.4.2 Interface and class names generation

The starting point of the program generation is the generation of interface names: we take from the configuration file the number of interfaces and the prefix for the name of each interface. Then all the names for the classes to generate are created, with the same parameters than the interface names. The generation of class names is required before the generation of the interfaces because we can define method signatures with using all the reference and primitive types. Each interface is generated assigning it a certain number of methods based on the parameter on the number of methods per interface. The method signatures for each interface are generated, using the class names and the parameter of the primitive types, taken from the configuration file, for the return type and the formal parameter of each method.

4.4.3 Inheritance relationships generation

The next step of the application is to generate relationships between classes, i.e., randomly assigning superclasses to certain other classes. In this phase, the PEG takes the information from the configuration file and interacts with the Prolog engine. The configuration retriever of the application takes the number of inheritances and the depth level of each inheritance chain from the configuration file. This phase is one of the first in the process because the generation of fields and methods also depends on the relationships between classes.

4.4.4 Interface assignment

After the inheritance chains generation, the control passes to the interface assigner component, which assigns the interfaces to the classes based on the number of interfaces to be implemented by classes, the number of implementations of a specific interface, and the number of classes that implement at least an interface. In this part, it is crucial to have coherent configuration parameters for correctly assigning interfaces to classes. For example, the application contains a constraint on the number of interfaces that has to be higher or equal than the number of classes divided by two, because it is difficult to satisfy the other configuration constraints if the number of interfaces is too small for the problem of assigning them to classes.

4.4.5 Class members generation

After the assignment of interfaces, the application starts generating fields, constructor signatures, and method signatures. The relevant information here is the relationship between classes: if a class A extends a class B then A inherits fields and methods from B. The process of generating the members of each class is thus executed in the following way: for each inheritance chains the generation of fields and method signatures starts from the superclass, so that the subclasses inherit the fields and the signatures from the superclasses, while the configuration constraints about the number of fields and methods are satisfied. The PEG component generates the constructor signatures with the attributes from the generated fields. The maximum number of constructors that a class can have depends on the number of class fields. The PEG component generates the method signatures by taking into account the methods derived by the interfaces implemented by the class if any.

4.4.6 Use of Prolog KB

We employ the Prolog KB here to create inheritance relations between classes, method and constructor signatures. To guarantee the correctness of the signatures and to avoid duplication, the application queries the KB and add facts consequently.

Inheritance relationship

When a class is assigned to another class as its superclass, the Prolog KB is queried for validating the inheritance. A specific Prolog rule determines if a class can extend another class and if no class circularity cycles are detected, e.g., if there is not a class A which extends a class B which in turn extends A. If the Prolog engine returns true, a fact about the inheritance just created is written on the Prolog KB. Indeed for detecting cycles, the KB has to know what classes are extending what other classes.

Method signatures

For validating a method signature the following query is sent to the Prolog KB: $useMethod(Name, Class, RetType, FormParams)$. The predicate $useMethod/4$ has four terms: the first represents the name of the method, the second the name of the class that owns the method, the third is the information about the result type of the method and the fourth term represents the type list of the formal parameters or an empty list if the method doesn't have parameters. The Prolog KB checks if the corresponding rule returns true. If that is the case, a Prolog fact of the following description is added: $method(Name, Id, Class, RetType, FormParams)$.
All the terms are the same as the query except for the term *Id* which is the unique identifier for the method.

Constructor signatures

Similarly, the PEG performs a query to the KB whenever a constructor signature needs to be validated.
The query is of the form: *canUseConstructor(Name,FormParams).* The first term is the name of the constructor, i.e., the class name, and the second term represents the list of types of the formal parameters of the constructor. In the same way than with methods, if the query returns true the following Prolog fact is added: *constructor(Name,Id,FormParams).* The second term is the unique identifier of the constructor.

4.4.7 Context creation

The generation of the program mentioned above is then stored in newly created objects. For every class, a symbol table object is created and filled with all the related class information, i.e., with the superclass name (if any), the interfaces implemented (if any), the method signatures, the constructor signatures, and the fields. To store type information, a global table object is instantiated and filled with class names, interfaces and primitive types.

4.5 Grammar parsing

This section covers in detail the work of the GP component: parsing the Java BNF grammar to represent it in a graph. Therefore, we explain the reason why the GP component parses a BNF grammar and then we show an example of a production rule translated into a graph. Afterward, we explain the task of parsing the grammar and creating the graph which represents the grammar, while showing the graph structure used for the traversal algorithm in the GT component.

4.5.1 Program as the set of terminal symbols of the AST

We can see a program as the set of leaf nodes of an *abstract syntax tree* (AST), where the non-leaf nodes of the AST represent the non-terminal symbols of the grammar producing the language and the leaf nodes represent the terminal symbols.

4.5.2 Grammar as a graph

Similarly, we can see the BNF grammar of a programming language as a graph containing all the possible abstract syntax trees that can be generated following the production rules. If the grammar contains recursion, the set of the possible ASTs is infinite.

4.5.3 Meaning of the graph

Leveraging these observations, we conceived the approach of parsing the grammar from the BNF notation and putting it into a directed graph, whose each node represents a symbol of the grammar and the directed arcs between two nodes represent the derivation of a rule given by the first symbol-node to the second symbol-node. The order of the arcs is of utmost importance for each node, as the order represents the correct sequence of symbols in each production rules.

Example of a production translated into a graph

An example of parsing a grammar rule is shown in Figure 4.5: each grammar symbol is converted into a node, and an arc from the left-hand side symbol to each of the symbols in the right-hand side of the rule is created. With this data structure as our internal representation of the grammar, it is possible to construct an abstract syntax tree starting from the root of the grammar and following a graph traversal algorithm.

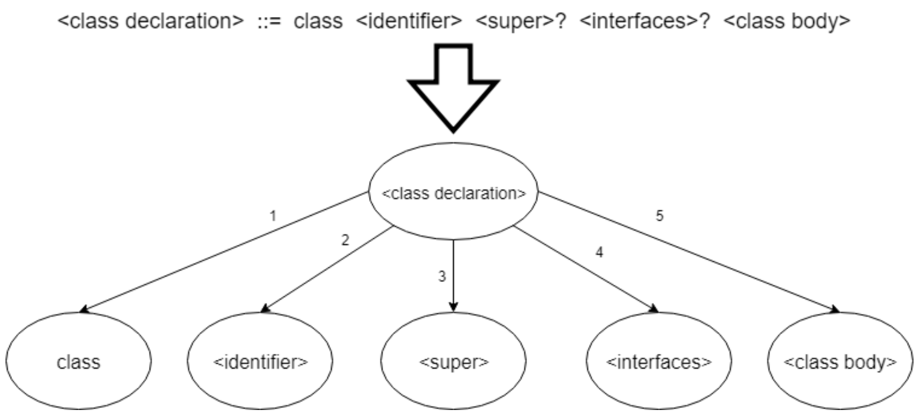

<class declaration> ::= class <identifier> <super>? <interfaces>? <class body>
<class declaration>
1
2
3
4
5
class
<identifier>
<super>
<interfaces>
<class body>

Figure 4.5: Example of a grammar rule converted into a directed graph

4.5.4 BNF Grammar

In this subsection we present the grammar used in the framework and we analyze the structure of the employed BNF grammar.

Featherweight Java

The BNF grammar used in the application generates a subset of Java very similar to Featherweight Java [25]. FeatherWeight Java is a lightweight and subset of the Java language that ensures type soundness, which means that the language guarantees that at run-time the prediction on the types is accurate. In our work, we used this subset as our reference, and we added the rules for interfaces and explicit constructor invocations using *this*.

Structure of the grammar

The grammar is in BNF form, with some symbols added to facilitate the parsing of it. For distinguishing between terminal and non-terminal symbols, two enclosing characters, < and >, are added to each non-terminal symbol. In Figure 4.6 we show the first rules of the grammar. The separator *::=* divides the left-hand side of each rule from the right-hand side of the rule. The other characters used in our BNG grammar are the following:

- The vertical bar | means that the right-hand side of the rule can have two distinct possibilities. In the $< typedeclarations >$ rule for example, the right-hand side can be either $< typedeclaration >$ or $< typedeclarations >< typedeclaration >$. In this case this is used to model recursion in the grammar rules.
- The symbol *?* means zero or one occurrence of the terminal. In the $< classdeclaration >$ rule for example, the $< super >$? element can have zero or one occurrence meaning that a class can avoid to extend another class or implement one or more interfaces.

```
<compilation unit> ::= <type declarations>

<type declarations> ::= <type declaration> | <type declarations> <type declaration>

<type declaration> ::= <class declaration> | <interface declaration>

<interface declaration> ::= interface <identifier> <interface body>

<interface body> ::= { <method headers>? }

<method headers> ::= <method header> ; | <method headers> <method header> ;

<class declaration> ::= class <identifier> <super>? <interfaces>? <class body>

<interfaces> ::= implements <interface list>

<interface list> ::= <interface type> | <interface list> , <interface type>

<interface type> ::= <type name>

<super> ::= extends <class type>
```

Figure 4.6: BNF Grammar rules

4.5.5 BNF Grammar parsing

As explained above, the grammar is parsed and a directed graph is created from the BNF representation. The Grammar Parser component is responsible of doing that.

Parsing the left-hand side of grammar rules

For each left-hand symbol of the grammar the GP creates a *Node* object. After this procedure, a set of node objects is available: these nodes represent all the non-terminal symbols of the grammar (because they have a derivation).

Parsing the right-hand side of grammar rules

Then, the algorithm looks at the right-hand part of each rule: for each symbol in the right-hand side the GP creates an arc object from the node of the left-hand part to the node of the right-hand part.

Types of nodes in the right-hand side

The nodes of the graph that are found in the right-hand part of each rule can be of three different types:

- **non-terminal**: it represents a symbol delimited by angular brackets. When a right-hand symbol is a non-terminal, the corresponding node is found in the set of nodes corresponding to non-terminal symbols previously created.
- **fictitious**: it represents a set of symbols representing an alternative in a rule with alternatives, in a manner we shall describe.
- **terminal**: it represents a symbol not delimited by angular brackets. When a right-hand symbol is a terminal, a new node is created.

Fictitious nodes A *fictitious* node is a node in the graph that represents an alternative of the right-hand part of a rule. We used the term *fictitious* because they are not meant to be related to specific grammar symbols.

Example of *fictitious* nodes In the rule < *typedeclarations* > ::=< *typedeclara tion* > | < *typedeclarations* > < *typedeclaration* > the GP parses the right-hand side of the rule and creates two *fictitious* nodes: one represented by the symbol < *typedeclaration* > and the other represented by the two symbols concatenated: < *typedeclarations* > < *typedeclaration* > The fictitious nodes need to be further parsed because they may present more nodes.

Further parsing of *fictitious* nodes In the same example of the previous paragraph, the *fictitious* node < *typedeclarations* > < *typedeclaration* > needs to have two arcs, one from < *typedeclarations* > < *typedeclaration* > to < *typedeclaration s* > and one from < *typedeclarations* > < *typedeclaration* > to < *typedeclaration* >. The other *fictitious* node, i.e. the one representing the symbol < *typedeclaratio n* >, simply becomes a non-terminal node after this process. Similarly, if the last *fictitious* node corresponded to a terminal symbol, it would become a terminal node.

Graph construction

At the end of the parsing process we end up having a graph object with a set of nodes and a set of arcs that link nodes together. Each node object has the following attributes:

- **description**: a String value representing the name of the node, e.g. < *classdeclaration* > or *extends*
- **salience**: a double value representing the weight associated to the rule. For terminal nodes the value is negligible
- **alternative**: a boolean value, true if the node corresponds to a symbol in the left-hand part of a rule whose right-hand part contains alternatives.
- **edges**: a list of the edges of the node
- **terminal**: a boolean value, true if the node corresponds to a terminal symbol, false otherwise.

Each edge object has the following attributes:

- **fromNode**: the node object from which the arc is directed
- **toNode**: the node object to which the arc is directed
- **optional**: a boolean value, true if the symbol corresponding to the *toNode* attribute contains the *?* symbol in the corresponding rule
- **weight**: a double value associated to the edges whose *fromNode* attribute is alternative, for directing the traversal algorithm.

Considerations about using a library

There are many interactive and useful graph libraries for creating graphs and applying traversal algorithms on them, such as the Spark GraphX library. We do not use them for two main reasons: first, the importance of the order of the arcs requires us to use an ordered data structure for keeping the edges for every node. In many libraries, the arcs or edges are stored in sets. Second, we need to model the alternative rules with an exclusive or for the arcs: in fact if a node is alternative, just one of its arcs has to be taken when traversing the graph. Therefore, the customization of an already implemented traversal algorithm is laborious. We propose our custom algorithm designed from scratch to handle the mentioned scenarios.

4.6 Concurrency in ProGen

As previously anticipated in chapter 3 of this document, a multithreading approach enables multiple tasks to be assigned to a fixed thread pool, where an executor handles memory management and switches context between threads, thus improving the overall performance of the application execution.

4.6.1 Rationale for concurrency

In our application, we employ concurrency after the generation of the program entities necessary to avoid dependency between classes. Indeed, with all class names, interfaces, declared constructors and methods, there is no need for a class to wait for the generation of another class, as the only parts not yet generated are the bodies of methods and constructors.

4.6.2 Thread implementation

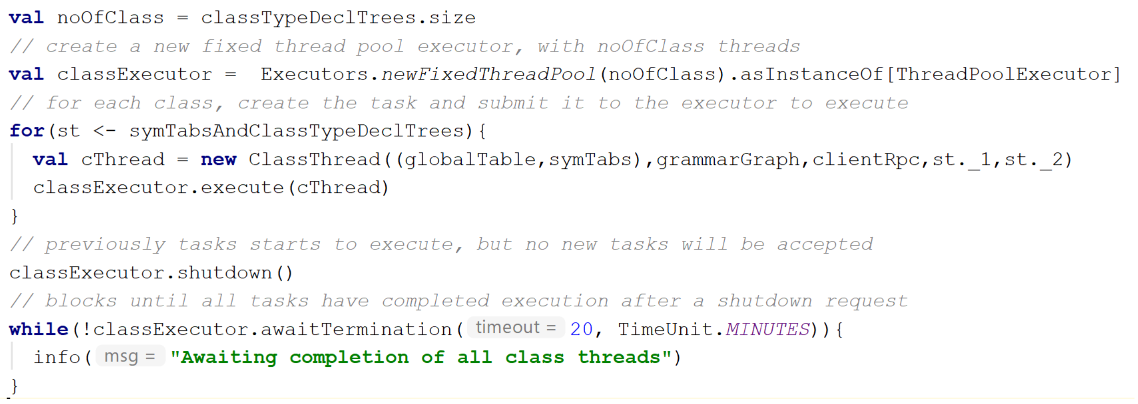
Therefore, we define a *Runnable* object called *ClassThread*. In Scala syntax this is done by declaring a class with the following piece of code: *class ClassThread extends Runnable* {} . For each class to be generated, the task *ClassThread* is created and submitted to the *ThreadPoolExecutor* as showed in Figure 4.7: the *ThreadPoolExecutor* executes the assigned tasks and then blocks until all tasks have completed execution after a shutdown request, or a timeout expires.

Characterization of ClassThread tasks

Each task represents a class to be generated and it has the following attributes:

- **context**: a shared resource by all threads, defines a tuple defined by two elements: the list of symbol tables containing class information for each class and the global table, consisting of all the available types
- **graph**: a graph to be traversed which represents the grammar. It is a shared resource
- **clientRpc**: a class which represents the interface to the Prolog Knowledge Base

- **classSymTab**: the symbol table representing the class that would be generated by this task
- **treeRoot**: the root of the AST's sub-tree from which the generation has to start



```
val noOfClass = classTypeDeclTrees.size
// create a new fixed thread pool executor, with noOfClass threads
val classExecutor =  Executors.newFixedThreadPool(noOfClass).asInstanceOf[ThreadPoolExecutor]
// for each class, create the task and submit it to the executor to execute
for(st <- symTabsAndClassTypeDeclTrees){
  val cThread = new ClassThread((globalTable,symTabs),grammarGraph,clientRpc,st._1,st._2)
  classExecutor.execute(cThread)
}
// previously tasks starts to execute, but no new tasks will be accepted
classExecutor.shutdown()
// blocks until all tasks have completed execution after a shutdown request
while(!classExecutor.awaitTermination( timeout = 20, TimeUnit.MINUTES)){
  info( msg = "Awaiting completion of all class threads")
}
```

Figure 4.7: Executing Runnable tasks with a *ThreadPoolExecutor*

4.7 Filling program entities in the AST

After having generated the program entities and having parsed the grammar into a graph, an abstract syntax tree is created with the root node of the grammar, i.e. the node corresponding to the symbol $< compilationunit >$. The goal of this phase is to fill the program entities in the AST by following the grammar rules, thus by traversing the graph.

4.7.1 Information produced by the PEG component

The information generated by the PEG component comprises a list of tables for each Java class and a global table.

Class Symbol Table

For each Java Class, a symbol table contains all the class members generated:

- **class name**: a String value denoting the name of the corresponding class
- **fields**: the list of the fields of the class, stored as tuples of type and name of field
- **methods**: the list of method signatures of a class. Each method signature is stored as an object with attribute *name*, *return type* and *formal parameters* (each one saved as a tuple of *type* and *name*)
- **constructors**: the list of constructor signatures of a class. Each constructor signature is stored as an object with attribute *class name* and *formal parameters*

- **superclass**: an optional value which contains *Some(class)* if the corresponding class has a super class, *None* otherwise
- **interfaces**: an option value which contains *Some(list)*, i.e. the list of interfaces, if a class implements one or more interfaces. None otherwise

Global table

Other than the list of class information stored in these tables, a global table is available for retrieving information for types. Indeed the global table contains the list of class names, the list of interfaces and the list of primitive allowed types retrieved from the configuration file.

4.7.2 Filling *type declaration* nodes

The first node of the abstract syntax tree is the start point of the grammar, i.e., the $< compilationunit >$ node. At the beginning, the application creates a number N of $< typedeclaration >$ nodes where $N = number\ of\ classes + number\ of\ interfaces$. Then for each symbol table and each interface the Filler Engine starts filling the information from the previous phases, traversing the graph from $< classdeclaration >$ node for the classes and $< interfacedeclaration >$ nodes for the interfaces (Figure 4.8).

Filling interface declarations

The interfaces are already entirely generated by the PEG, so the sub-trees are filled with interface names and method signatures, and the traversal of the graph comes to an end for this part.

Filling class declarations

The traversal of the graph and filling of the $< classdeclaration >$ sub-trees is instead not complete, because the information about classes lacks the implementation of method and constructor bodies. The class information filling lasts longer as the class information is more abundant than the interface information. After having filled the class information, the sub-trees starting from $< classdeclaration >$ are not complete.

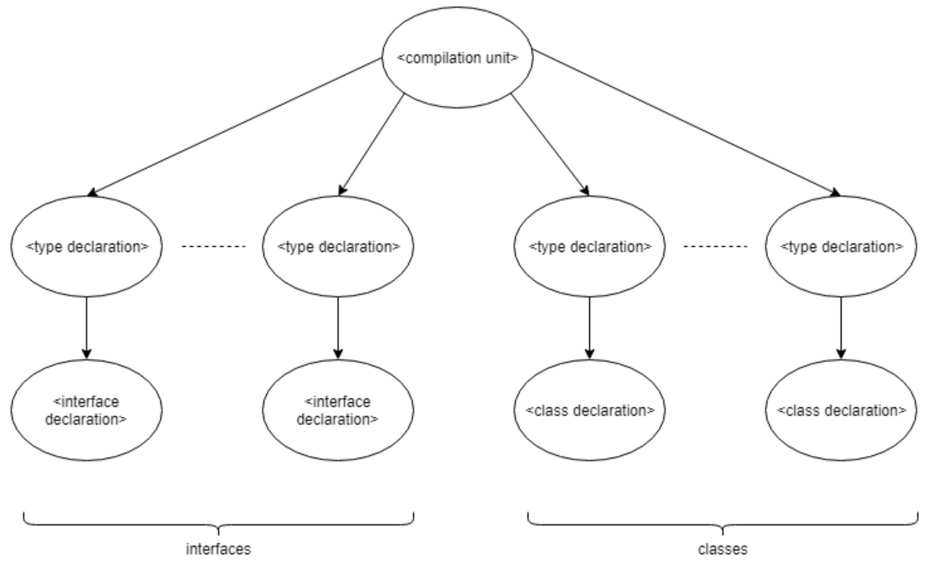


Figure 4.8: AST $< typedeclaration >$ nodes are created based on the number of interfaces and the number of classes

4.7.3 Use of Prolog in program entities filling

In this part, there is no engagement of Prolog rules: the program entities generated in the first part are ensured to be correct by the previous generation, where Prolog rules are invoked for class inheritance, use of constructors signatures and use of method signatures.

Adding nodes to the Prolog KB

However, from the moment the AST is created, the Grammar Traverser fills the Prolog Knowledge Base with facts about each node added to the AST. When the application adds a node from the graph to the AST, a fact to the Prolog Knowledge Base is added with the information of that node.

Characterization of Prolog predicate *node*

Thus the predicate *node/4* has the following terms:

- **Name**: the name of the node, e.g. $< classdeclaration >$ or *new*
- **Id**: an integer representing the unique identifier of the node in the Prolog KB
- **ParId**: an integer representing the unique identifier of the parent node in the Prolog KB
- **Depth**: an integer representing the depth of a node, i.e. the distance from the root node, which has depth equal to 0

Assigning identifiers to methods and constructors

In this phase, a critical property in the Knowledge Base must hold. As we have said in subsection 4.4.6, method and constructor signatures exist already; at the same time, Prolog facts about the signatures of methods and constructors are added to the KB with the predicates *method/3* and *constructor/3*. Associated to these facts is a unique identifier that has to be equal to the identifier of respectively the $< methoddeclaration >$ and $< constructordeclaration >$ node facts. The reason for doing so is that some Prolog rules need to find the information about the parameters of methods and constructors through the unification of identifiers.

4.8 Constructor and method bodies generation

In this phase, all the interface sub-trees have completed, and each class has its own AST sub-tree starting from the $< classdeclaration >$ node. However, the classes still lack the bodies of methods and constructors. The generation of the bodies represent the most crucial part of ProGen, as it involves many different aspects: among these aspects, the most important ones are the traversal of the BNF grammar, the interaction and the validation with the Prolog KB, and the update of the symbol tables and the scopes. In the next subsections, we analyze and describe these parts without following a temporal order coherent with the execution of the application like we have done until now. Moreover, we describe the bodies generation as it is executed by a single thread which is performing a single task because for each thread the same task is assigned for a different class to generate.

4.8.1 Properties of depth-first traversal algorithm

The generation of method and constructor bodies starts with respectively two AST nodes, $< methodbody >$ and $< constructorbody >$, that are leaf-nodes before the application starts applying the algorithm explained in this part.

Visit and backtrack

The algorithm of traversing the graph is depth-first, meaning that the algorithm visits children nodes until it reaches leaf-nodes (terminal nodes) before backtracking. In our case, this means that the algorithm explores the production rules of the grammar in a depth-first fashion until it reaches a terminal symbol, and then the algorithm backtracks.

Example of a depth-first traversal

Figure 4.9 shows a sample graph with the order of visit on each arc. First, starting from the root which is always a non-terminal node denoted by T in the figure, the adjacent nodes are visited starting from the one in the left. If the node is a non-terminal, the visit goes forward recursively on this node. Otherwise, if the node is terminal, the visit stops and backtracks to the father of the node, which repeats the visit for its second child and so on recursively. In Figure 4.9 the left part of the graph is therefore visited before the right part.

Considerations about the algorithm

We think relevant to say that for our purpose the traversal algorithm does not perfectly match a depth-first search visit of a graph. In the traditional version of the depth-first search algorithm, each node is visited exactly once. Since we are dealing with the random generation of a program and in our case the rules are the production rules of the grammar, we do not want to visit just once a production rule since that rule could be used for different situations in a program. As an example, consider the rule of the associated node $< identifier >$: this rule, or rather this node, is encountered all the times a variable is assigned, a method is invoked, or an interface is declared. We want that the visit of the graph could not stop just when a node is visited once but could stop only when all the terminal nodes are reached and the Prolog checks have passed.

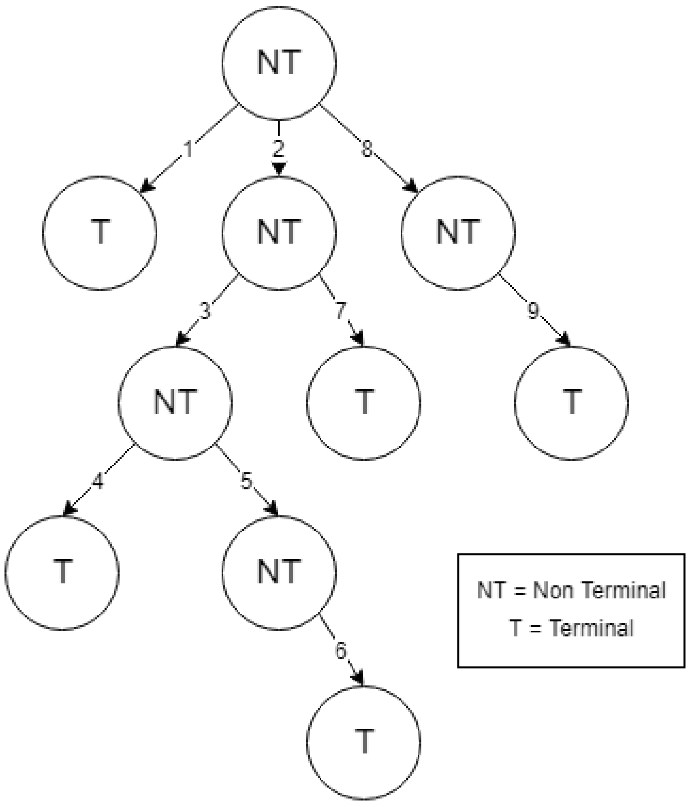


Figure 4.9: Depth First Order of Visiting Nodes

4.8.2 Body traversal algorithm

The body traversal algorithm is presented here. As a reminder, the goal of the algorithm is to add the nodes of the grammar graph to the abstract syntax tree to generate a body that is correct, thanks to the queries to the Prolog Knowledge Base.

Procedure TRAVERSE-BODY

The procedure *TRAVERSE-BODY* needs a parameter, i.e., the current abstract syntax tree. When the procedure is called for the first time, the abstract syntax tree represents the node < *methodbody* > or the node < *constructorbody* > for respectively method bodies or constructor bodies. The tree is void, meaning that the tree has no children. When the procedure *TRAVERSE-BODY* is called, the node root of the tree is taken.

Algorithm 1 Body Traversal - Part 1

```
1: function TRAVERSE-BODY($ast$)
2:     $node \leftarrow ast.node$
3:     if $node.alternative = true$ then
4:         $newNode \leftarrow select(node.arcs)$
5:         if $newNode.terminal = true$ then
6:             $ADD - TERMINAL(ast, newNode)$
7:         else
8:             $TRAVERSE - NT(ast, newNode)$
9:         end if
10:        return $ast$
```

If the node is alternative, an arc among the arcs of the node is selected and the target node *newNode* of the selected arc is taken. A node is alternative if it represents a production rule of the grammar where the right-hand part of the rule contains choices. The target node *newNode* can be either a terminal node representing a terminal symbol in the grammar or a non-terminal node. If the target node is terminal, the procedure *ADD-TERMINAL* is called. Otherwise, the procedure *TRAVERSE-NT* which traverses a non-terminal node is called.

Algorithm 2 Body Traversal - Part 2

```
11:    else
12:        $A \leftarrow node.arcs$
13:        for all $a \in A$ do
14:            $n \leftarrow a.toNode$
15:            if $n.terminal = true$ then
16:                $ADD - TERMINAL(ast, n)$
17:            else
18:                $TRAVERSE - NT(ast, n)$
19:            end if
20:        end for
21:        return $ast$
22:    end if
23: end function
```

If the node is not an alternative node, all the arcs of the node are considered. For each arc of the node, the target node is taken. As before, if the target node is terminal, the procedure *ADD-TERMINAL* is called. Otherwise, the procedure *TRAVERSE-NT* is called.

Procedure ADD-TERMINAL

Algorithm 3 Add Terminal Node

```
1: procedure ADD-TERMINAL($ast, newNode$)
2:     $newAst = ast.add(newNode)$            ▷ Adds $newAst$ to $ast$'s children
3:     return ;
4: end procedure
```

In the $ADD - TERMINAL$ procedure, the terminal node is added to a new AST, and the new AST is added to the old one.

Procedure TRAVERSE-NT

Algorithm 4 Traverse Non Terminal Node

```
1: procedure TRAVERSE-NT($ast, newNode$)
2:     $newAst \leftarrow ast.add(newNode)$                ▷ Adds $newAst$ to $ast$'s children
3:     $completedAst \leftarrow TRAVERSE - BODY(newAst)$
4:     if $completedAst.valid = false$ then                        ▷ Prolog Check
5:         $voidAst \leftarrow completedAst.removeChildren$     ▷ Empty the current AST
6:         $TRAVERSE - BODY(voidAst)$                              ▷ Recursive Call
7:     end if
8:     return ;
9: end procedure
```

The procedure *TRAVERSE-NT* continues the traversal of the graph instead of just adding a new node to the AST. First, a *newAst* is created by adding a *newNode* to *ast*. Therefore, the *newAst* is also added as a child of the *ast*. Then at line 3 the procedure $TRAVERSE - BODY$ is called on the *newAst* and the result of the procedure is assigned to *completedAst*. After line 3, the sub-tree related to *newAst* is completed, i.e., after line 3, the algorithm is backtracking. The *completedAst* is then checked through the Prolog KB. If the Prolog Knowledge Base returns *true*, then the procedure ends. Otherwise, if the KB returns $false$, all the children of *completedAst* are deleted and the procedure $TRAVERSE - BODY$ is again called on *voidAst*.

Efficiency and termination issues

The previous algorithm shows a recursive scheme that potentially has no ends. Even if the execution of the algorithm has an end, the overall execution time may be affected by the repetition of some parts not validated by Prolog. Thus, the two main drawbacks that the algorithm has are time inefficiency and infinite loop.

Time inefficiency

The rules of the grammar that are alternative are taken either with a pseudo-random distribution or with a probability assigned to the rules of the grammar by the configuration file, meaning that the context does not dictate the choice of choosing a derivation rather than another. In doing this, the application spends relatively a considerable amount of time to validate some parts of the program. For instance, let consider a method invocation to be generated in a class. The class can call ten methods with an average of three parameters each. Then, the odds that the application picks the right rules with the correct arguments for creating the method invocation of a chosen methods are very low. Moreover, the probability of doing a correct method invocation decreases as the number of available types gets greater. Thus, in the application, we limit the number of iteration of the previous algorithm. In this way, only a predefined number of times the application tries to *guess* the correct generation. In general, the lower the number of iterations is, the faster the application executes.

Infinite loop

There may be the possibility that in the generated entities there are not enough resources to even *guess* a correct generation. For example, a method may not have some available class fields or local variables to correctly instantiates a new class. In this situation, the algorithm previously presented enters in a loop. If the Prolog KB never returns true, the algorithm remains stuck seamlessly calling the $TRAVERSE - BODY$ and the $TRAVERSE - NT$ procedure.

Using ***FeasibilityEngine*** and variable initializations

The $FeasibilityEngine$ component addresses both the problems of time inefficiency and infinite loop by looking at the resources stored in the symbol table and generating new code. For a given sub-tree of the AST, the component first determines whether there are the available resources for the given task. Then, if there are the available resources for the task, the component uses those resources for generating a correct sub-tree. If there are not enough resources, the component determines what are the missing resources and generates them. In our application, this means that the $FeasiblityEngine$ initializes other local variables *ad hoc* for the objective. The variable initialized are then used by the application to generate the missing features.

4.8.3 Weighted sampling

When an alternative node is considered during the previous algorithm, only one alternative has to be taken. An example of a rule without alternatives is in Figure 4.10. In this rule, when evaluating the symbol $< statementexpression >$, only one symbol can be chosen among the symbols $< methodinvocation >$ and $< classinstancecreationexpression >$.

```
<statement expression> ::= <method invocation> | <class instance creation expression>
```

Figure 4.10: An example of an alternative rule

Handling alternative rules

Two ways of handling the alternative nodes can be taken: the first is a random draw from the list of nodes, the second is a weighted draw with probabilities associated with each node. We take a hybrid approach and use both ways: for some alternative nodes we randomly pick up a node among the possible nodes, for other alternative nodes we provide different alternatives and we adjust them to move faster to the generation.

Choosing an alternative

In the example in Figure 4.10, if we wanted to randomly pick up one of the two nodes, we would assign the same weight to the two nodes, i.e. $w(< methodinvocation >) =$

$0.5 = w(< classinstancecreationexpression >)$. Instead, if we wanted to perform a weighted sampling, we would assign different weights to the two nodes, with the constraint $w(< methodinvocation >) + w(< classinstancecreationexpression >) = 1$. Thus, when the algorithm has to choose an alternative, a function samples one of the nodes.

Sampling function This function takes the list of weights of each arc of the node and draws an arc based on the weights given in input. The weighted sampling is based on a random double *rand* between 0 and 1 which is picked up at the beginning. Then, the list of weights in the input is transformed into a list of increasing double values between 0 and 1, such that the proportion between the weights is maintained. For example, given a list of two weights $List(0.4, 0.6)$, the list transformed would be $List(0.4, 1.0)$. When compared with *rand*, if *rand* is less or equal than 0.4, the first element is selected. Otherwise, the second element is selected. It is easy to notice that with a transformed list and the comparison of the *rand* value, the first element has 0.4 probability to be chosen and the second element has 0.6 probability to be chosen.

Weight configuration file

In the application, a file with an initial configuration of the weights of certain arcs is read as input as part of the configuration parameters. In Figure 4.11 we show the weights associated with some alternative rules. The application reads the configuration and sets the weights to the arcs corresponding to the given rules. For every symbol in the file, there is a list of double values which represent the weight applied to each alternative. The sum of the weights for each symbol must be equal to 1. The other alternative rules are given a default value which is equal to $1/(numberofarcs)$.

User intervention in weights setting The choice to give weights to specific rules is to generate relevant code. However, the user can play to change the weights or can add a new symbol, but if the sum of the list of weights is not equal to 1, the application prompts an error.

```
weights{

  <methodbody> = [1.0,0.0]

  <argumentlist> = [0.4,0.6]

  <assignmentexpression> = [0.5,0.5]

  <blockstatement> = [0.6,0.4]

  <statementwithouttrailingsubstatement> = [0.5,0.0,0.5,0.0]

  <postfixexpression> = [0.3,0.7]
}
```

Figure 4.11: The weights configuration file

Setting weights to control generation

The Prolog KB and the previously presented algorithm ensure that the generation of programs does not incur in syntactic errors. Nonetheless, from an efficiency point of view, the generation of totally random code can add a significant overhead given the almost infinite combinations of rules. Thus, the application makes use of weights to guide the generation to certain specific nodes when there is the necessity.

Return statement example A clear example is for the $< returnstatement >$ rule: according to the Java Language Specification, a $< returnstatement >$ has to be the last statement in a block or a method, i.e., no other statements have to appear after a return statement. When the application is generating the body of a method, the weight of the arc from the alternative node $< statement\ without\ trailing\ substatement >$ to the node $< returnstatement >$ is set to 0. After the application has completed the generation, the weight of that arc is set to 1.0, and the other arcs from the same node are set to 0. Then, the function of body traversal and AST construction is called again, having the certainty that a return statement is going to be generated at the end of the method. After the generation of the return statement, the default configuration settings are stored for the weights. Similarly, some weights are changed to control the generation in other situation to avoid that the application remains stuck in the generation of a part of the abstract syntax tree.

4.8.4 Use of symbol table

As previously explained in the second chapter, the generation of random programs makes use of software components and data structures that are strictly related to the components a compiler. In particular, we make use of a data structure that we call *symbol table* because it is very similar to the symbol table in a compiler.

Contents of *Symbol Table*

The Symbol Table in our application is a table that contains information about scopes in the program, and it provides the following attributes:

- **symTabEntry**: an entry for the symbol table with all the specific information of the scope
- **prev**: an optional value of symbol table of the previous scope
- **next**: an optional list of symbol tables of the next scope

Contents of *SymTabEntry* In detail, the information of the **symTabEntry** are the following:

- **kind**: the enclosing scope can be $COMPILATIONUNIT$, $CLASS$, $CONSTRUCTOR$, $METHOD$, $BLOCK$.
- **name**: an optional value representing the name of the enclosing scope, e.g. the class name if **kind** is $CLASS$ or the method name if **kind** is $METHOD$.
- **superClass**: an optional value for the name of the super class present if **kind** is $CLASS$ and the class is a subclass.
- **interfaces**: an optional list of interfaces, present if **kind** is $CLASS$ and class has interfaces.
- **methods**: an optional list of method signatures, present if **kind** is $CLASS$.
- **constructors**: an optional list of constructor signatures, present if **kind** is $CLASS$ and class has defined constructors.
- **fields**: an optional list of fields, present if **kind** is $CLASS$ and class has at least one attribute.
- **locVars**: an optional list of tuples $(String, String)$ defining the local variables of the scope, where the first element of the tuple represents the type of the local variable and the second element the name. Present if **kind** is $METHOD$, $CONSTRUCTOR$ or $BLOCK$.
- **retType**: an optional *String* value representing the return type of a method, present if **kind** is $METHOD$.
- **params**: an optional list of tuples $(String, String)$ defining the formal parameters of a constructor or a method, where the first element of the tuple represents the type of the local variable and the second element the name. Present if **kind** is $CONSTRUCTOR$ or $METHOD$.

Entering and leaving scopes with Symbol Tables

During the generation, only the symbol table of the current scope is available. However, thanks to the attribute **prev** previously seen, it is possible to look up elements from enclosing scopes, i.e., the previous symbol tables. As our traversal function starts traversing the grammar from the constructor bodies and method bodies, the only symbol table that is added is of type $BLOCK$. In fact, before calling the traversal function on each body node of each class, a symbol table of kind $CONSTRUCTOR$ or $METHOD$ is created, and the related class Symbol Table is inserted in the attribute **prev** of the newly created symbol table.

Entering and leaving a block When a node $< block >$ is found, a new scope is entered and a new symbol table of kind $BLOCK$ is created and assigned to the value of the current symbol table. When the function backtracks, if a node $< block >$ is left, the current symbol table retakes the value of the previous scope's symbol table. In the application the functions *enterScope* and *leaveScope*, present in Figure 4.12 handles the symbol table value.

```
trait SymTab{
  def enterScope(tab:SymTab,node: Node) : SymTab

  def leaveScope(tab:SymTab,node: Node): SymTab

  def visitSymTab(kind: SymTabEntryKind, sm: SymTab) : Option[SymTab]

  def lookUpFields: Option[List[(String,String)]]

  def lookUpConstructors(name: String): Option[List[ConstructorSignature]]

  def lookUpMethods: Option[List[MethodSignature]]

  def getRetType(tab: SymTab): Option[String]

  def lookUpType(name:String): Option[String]

  def lookUpClassTab(name: String, tab: SymTab): Option[SymTab]

  def insertLocVar(sType:String,name:String): SymTab
}
```

Figure 4.12: The *SymTab* trait exposing functions

Retrieving and inserting information with Symbol Table

During the generation, the application needs some information about scopes and types to make a correct generation and to select the right elements. Thus, the *SymTab* Scala trait in Figure 4.12 provides look-up functions to look for values and enclosing symbol tables.

Look-up functions The look-up functions look for kinds of symbol tables, list of signatures, types and names of fields, and types of every name inserted. Their implementation includes the invocation of the *visitSymTab* function which finds the symbol table with the specified *SymTabEntryKind*.

Inserting local variables A symbol table also needs to be updated whenever a local variable is declared, and the function *insertLocVar* does this by inserting into the current symbol table the type and names of the variable declared.

Getting the return type of a method Finally, a function *getRetType* is defined for getting the return type of the method of the current symbol table.

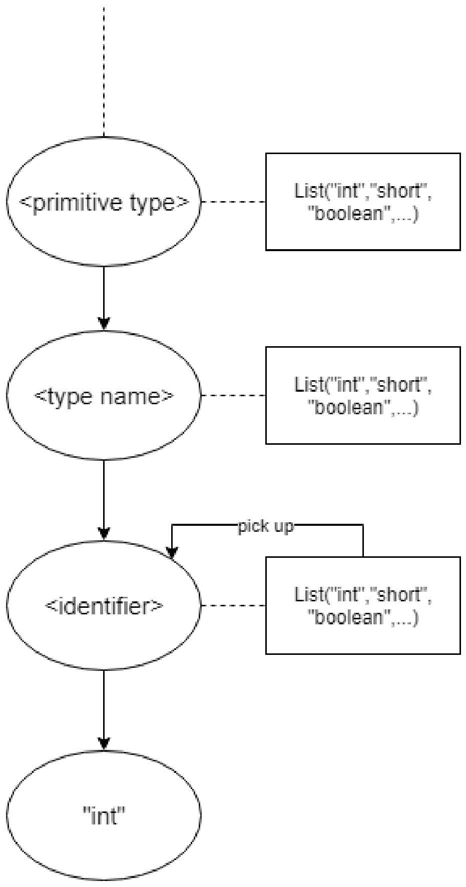

<primitive type>
List("int","short", "boolean",...)
<type name>
List("int","short", "boolean",...)
pick up
<identifier>
List("int","short", "boolean",...)
"int"

Figure 4.13: How the list of possible names is used to generate a name

4.8.5 Use of list of possible names

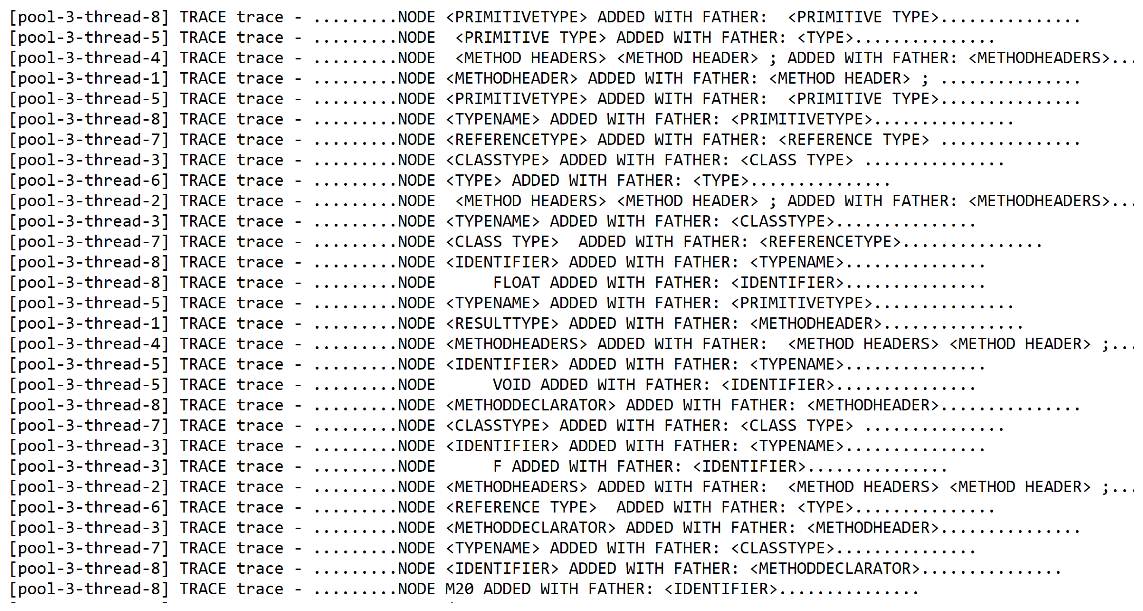
During the generation of method and constructor bodies, one of the most critical and challenging parts is to generate the right name for the right context related to a given derivation of a rule. Considering all the possibilities of using a name would be time-consuming.

Handling identifiers

The node $< identifier >$, which is the node related to a name in the program, is found in many different contexts. An identifier can be a name of a class, a name of an interface, a primitive type, a local variable name, a field name, a name of a method, a name of a parameter. For the sake of making the process of generation faster and for avoiding too many queries to the Prolog KB, a software component is devoted to generating the list of possible names given a node. As an example, when a $< primitivetype >$ node is traversed, the $IdentifierHandler$ component generates a list of all the possible primitive types so that when the $< identifier >$ node is encountered, a right name is assigned to the next node (Figure 4.13).

4.8.6 Execution trace

During the traversal algorithm, an execution trace log is maintained at run-time to keep track of the added nodes and the AST construction. The execution trace allows verifying whether the Grammar Traverser properly constructs the AST. When generating many lines of code, the execution trace file increases its size and can arrive at a significant size of 60-70 Megabytes. In Figure 4.14 an extract of the execution trace is presented.



```
[pool-3-thread-8] TRACE trace - .........NODE <PRIMITIVETYPE> ADDED WITH FATHER:  <PRIMITIVE TYPE>..............
[pool-3-thread-5] TRACE trace - .........NODE  <PRIMITIVE TYPE> ADDED WITH FATHER: <TYPE>..............
[pool-3-thread-4] TRACE trace - .........NODE  <METHOD HEADERS> <METHOD HEADER> ; ADDED WITH FATHER: <METHODHEADERS>...
[pool-3-thread-1] TRACE trace - .........NODE <METHODHEADER> ADDED WITH FATHER: <METHOD HEADER> ; ..............
[pool-3-thread-5] TRACE trace - .........NODE <PRIMITIVETYPE> ADDED WITH FATHER:  <PRIMITIVE TYPE>..............
[pool-3-thread-8] TRACE trace - .........NODE <TYPENAME> ADDED WITH FATHER: <PRIMITIVETYPE>..............
[pool-3-thread-7] TRACE trace - .........NODE <REFERENCETYPE> ADDED WITH FATHER: <REFERENCE TYPE> ..............
[pool-3-thread-3] TRACE trace - .........NODE <CLASSTYPE> ADDED WITH FATHER: <CLASS TYPE> ..............
[pool-3-thread-6] TRACE trace - .........NODE <TYPE> ADDED WITH FATHER: <TYPE>..............
[pool-3-thread-2] TRACE trace - .........NODE  <METHOD HEADERS> <METHOD HEADER> ; ADDED WITH FATHER: <METHODHEADERS>...
[pool-3-thread-3] TRACE trace - .........NODE <TYPENAME> ADDED WITH FATHER: <CLASSTYPE>..............
[pool-3-thread-7] TRACE trace - .........NODE <CLASS TYPE>  ADDED WITH FATHER: <REFERENCETYPE>..............
[pool-3-thread-8] TRACE trace - .........NODE <IDENTIFIER> ADDED WITH FATHER: <TYPENAME>..............
[pool-3-thread-8] TRACE trace - .........NODE      FLOAT ADDED WITH FATHER: <IDENTIFIER>..............
[pool-3-thread-5] TRACE trace - .........NODE <TYPENAME> ADDED WITH FATHER: <PRIMITIVETYPE>..............
[pool-3-thread-1] TRACE trace - .........NODE <RESULTTYPE> ADDED WITH FATHER: <METHODHEADER>..............
[pool-3-thread-4] TRACE trace - .........NODE <METHODHEADERS> ADDED WITH FATHER:  <METHOD HEADERS> <METHOD HEADER> ;...
[pool-3-thread-5] TRACE trace - .........NODE <IDENTIFIER> ADDED WITH FATHER: <TYPENAME>..............
[pool-3-thread-5] TRACE trace - .........NODE      VOID ADDED WITH FATHER: <IDENTIFIER>..............
[pool-3-thread-8] TRACE trace - .........NODE <METHODDECLARATOR> ADDED WITH FATHER: <METHODHEADER>..............
[pool-3-thread-7] TRACE trace - .........NODE <CLASSTYPE> ADDED WITH FATHER: <CLASS TYPE> ..............
[pool-3-thread-3] TRACE trace - .........NODE <IDENTIFIER> ADDED WITH FATHER: <TYPENAME>..............
[pool-3-thread-3] TRACE trace - .........NODE      F ADDED WITH FATHER: <IDENTIFIER>..............
[pool-3-thread-2] TRACE trace - .........NODE <METHODHEADERS> ADDED WITH FATHER:  <METHOD HEADERS> <METHOD HEADER> ;...
[pool-3-thread-6] TRACE trace - .........NODE <REFERENCE TYPE>  ADDED WITH FATHER: <TYPE>..............
[pool-3-thread-3] TRACE trace - .........NODE <METHODDECLARATOR> ADDED WITH FATHER: <METHODHEADER>..............
[pool-3-thread-7] TRACE trace - .........NODE <TYPENAME> ADDED WITH FATHER: <CLASSTYPE>..............
[pool-3-thread-8] TRACE trace - .........NODE <IDENTIFIER> ADDED WITH FATHER: <METHODDECLARATOR>..............
[pool-3-thread-8] TRACE trace - .........NODE M20 ADDED WITH FATHER: <IDENTIFIER>..............
```

Figure 4.14: Execution Trace

Chapter 5

Validation with Prolog

The validation of the nodes of the AST is a central part of this work. Thus, we dedicate a chapter for the explanation of the rules and how the application interacts with the Knowledge Base. In section 5.1, we summarize the most relevant part of the interaction between the application and the Prolog KB. In this section, we also show an example of a Prolog rule in detail. Section 5.2 contains the correspondence of some Prolog rules with their JLS counterpart. Then, in section 5.3 we show the process of testing the Prolog rules with the *Eclipse ASTParser* library, according to the Java Language Specification rules. Finally, in section 5.4.2 we explain how we implement the communication between the Prolog Knowledge Base and the application through the usage of gRPC protobuf communication protocol.

5.1 Recap of the use of Prolog KB by Progen

The goal of the Prolog Knowledge Base is to validate the nodes of the AST following the Java Language Specification rule.

5.1.1 Rationale of using Prolog

The reasons for modeling the *JLS* for generating random programs are different. Here we want to give three main reasons that characterized our choice.

Separation of concerns

First, a reason is the separation of concerns: the syntactic rules are respected and guaranteed through the traversal of the grammar in the application, while the semantic rules are checked in the Knowledge Base through appropriate queries from the application. In this way, the software components that handle the different parts are loosely coupled.

Scalability

Second, the specification of the Java Language through the Prolog Knowledge Base and consequently the description of a system are easy to evaluate and maintain, and new rules can be added with a minimum effort [42].

Formal specification

Third, even though a formal specification cannot prove the correctness of the model *apriori*, it can drastically increase the possibility to detect errors and anomalies in the modeled system [14].

5.1.2 Interaction of Prolog KB with ProGen

In ProGen the interaction with the Prolog KB takes place in three main situations:

- When a node is added to the AST, a fact about the node is inserted into the KB with a unique identifier, where *node*/4 has a *Name*, a unique *Id*, the unique id of its father *ParId* and the level of *Depth* in the AST. For the formalization of the AST in Prolog, we took inspiration from [19].
- When the body traversal algorithm backtracks, the sub-tree constructed is passed to a function which constructs a query and sends it to the Prolog KB for validating the sub-tree.
- When the body traversal algorithm backtracks, the type of the node is determined by a function and a Prolog fact is added to the KB with predicate *type*/2 with a *Type* value and the unique *Id* of the corresponding node.

5.1.3 Example of a Prolog rule

Filling the Prolog with facts

Before showing the example rule, it is important to notice that the application fills the Prolog KB with all the information about the class relations, i.e., the inheritance between classes, the method signatures, and the constructor signatures. When the application starts traversing the graph and adds the nodes to the AST, a Prolog fact is added to the Prolog KB for each node added to the AST. The reason is for keeping a model description of the AST in the Knowledge Base and for making possible the application of the Prolog rules.

Unification through ID in methods and constructors As already remarked in 4.7.3, when adding two particular nodes, i.e. $< methoddeclaration >$ and $< constructordeclaration >$, the same unique ID of their corresponding *method*/5 and *constructor*/3 facts is given to the nodes, thus allowing the Prolog engine to find useful information thanks to unification of the unique identifier.

Return statement rule

JLS Rule 14.17 According to the Java Language Specification [26] at paragraph 14.17: *"When a return statement with an Expression appears in a method declaration, the Expression must be assignable to the declared return type of the method, or a compile-time error occurs"*. So, when a method declares a return type, the type of the *Expression* in the *return* statement must coincide with the return type. The corresponding Prolog rule *useRetStmt*/3 is in Figure 5.1.

Constructing the query for return statement When the traversal function of the graph backtracks and the current generated sub-tree has its root at the $< returnstatement >$ node, a query for Prolog is constructed using a recursive function on the sub-tree that finds the type of the $Expression$ in the $return$ statement. The type represents the $RetType$ in the Prolog rule. Then, the application looks for the parent of the $< returnstatement >$ node and takes its unique identifier for the $ParId$ term in the rule. The depth level of the node $< returnstatement >$ is directly taken from the attributes of the AST, because every time a node n is added to the AST, the corresponding sub-tree with root n has the depth level incremented by 1. The depth goes for the term $Depth$, and finally, a query is constructed.

Performing a query A query could be for example $useRetStmt('int', 3521, 23)$. and the Prolog engine looks at the corresponding rule to determine whether it is true or false. As Figure 5.1 shows, the Prolog engine looks for a $node/4$ fact that has the first term equal to $' < methoddeclaration >'$ and is an ancestor of the $< returnstatement >$ node. If the node is found, the Prolog engine looks if there is a $method/5$ fact that has the same unique id of the $' < methoddeclaration >'$ node and the same $RetType$ of the query. If this is true, the Prolog engine returns true, otherwise is false.

```
useRetStmt(RetType,ParId,Depth) :-
                        lookForNode('<methoddeclaration>',X,_,_, ParId, Depth),
                        method(_,X,_,RetType,_).

lookForNode(N,X,Y,Z,ParId,Depth) :-
                        Z is Depth-1,
                        X is ParId,
                        node(N,X,Y,Z).

lookForNode(N,X,Y,Z,ParId,Depth) :-
                        T is Depth-1,
                        U is ParId,
                        node(_,U,S,T),
                        lookForNode(N,X,Y,Z,S,T).
```

Figure 5.1: Prolog Rules checking JLS Rule 14.17

Unification and recursion

In the previous rule, we show how unification and recursion are two powerful concepts for the implementation of a Knowledge Base. Thanks to the unification of the identifiers and the depths, we can go upwards in the tree representation in Prolog to find the enclosing method declaration that in turn unifies its id with the information about the return type of the method. For looking at the nodes, the Prolog rules exploit the advantage of recursion by giving two definitions of the same rule $lookForNode/6$: the former definition represents the base case, the latter represents the recursive call.

5.2 Prolog-JLS correspondence

In this section, we present the correspondence between Prolog rules and JLS rules. For every considered rule of the Java Language Specification [26], we propose the corresponding Prolog rule. In this work, only a part of the following rules has been used, because the considered subset of Java that we used for the generation does not have all the language features included in some rules.

5.2.1 Legend of the tables

JLS Rules In the JLS Rules, the number of the rule reflects the location of the rule in the Java Language Specification Java SE8 Edition that we have chosen [26]. The abbreviation of CTE stands for Compiler Time Error.

Prolog Rules In the Prolog rules, the predicate \+ stands for the logical *NOT*.

5.2.2 Rules about modifiers

JLS 8.1.1 - two modifiers

In Table 5.1 we show a rule about the use of two modifiers in the same class declaration, that is not allowed by the corresponding JLS rule. In Prolog two predicates are used: $useModifiers/4$ and $two/1$. The Prolog rule states that in the same class declaration node it is not possible to have more than one class modifier.

JLS Rule	Prolog Rules
§8.1.1: "CTE if more than two class modifiers for a class."	useModifiers('<classmodifiers>',_,X,Y) :- Z is Y-1, node('<classdeclaration>',X,_,Z), \+ two('<classmodifiers>'). two(Name) :- node(Name,_,Y,_), node(Name,Y,_,_).

Table 5.1: Two modifiers rules

JLS 8.1.1 - same keyword

In Table 5.2 the JLS rule states that the same keyword cannot be used more than once in a class modifier. The Prolog rule shows an example of the rule, where the keyword *public* appears more than once.

JLS Rules	Prolog Rules
§8.1.1: "CTE if the same keyword appears more than once in a class modifier."	useModifier('public',_,X,Y) :- Z is Y-1, \+ node('public',_,X,Y).

Table 5.2: Same keyword rules

JLS 8.1.1 - collision between abstract and final keywords

In Table 5.3 there is a rule that prohibits the use of the keyword *final* when there is already a keyword *abstract* or the other way around.

JLS Rules	Prolog Rules
”§8.1.1: cannot use abstract when there is final and the other way around.”	useModifier('abstract',_,X,Y) :- Z is Y-1, node('classmodifier',X,_,Z), \+ node('final',_,X,Y), \+ node('abstract',_,X,Y).

Table 5.3: Collision between a*abstract* and *final* rules

JLS 8.1.1.1 - abstract class

In Table 5.4 we show that the Prolog rule uses unification with the ID of the added nodes to find the class modifier of a class. If the keyword *abstract* is present, the Prolog rule returns false.

JLS Rules	Prolog Rules
”§8.1.1.1: no abstract classes can be instantiated.”	useIdentifier(Name,_,X,Y) :- P is Y-1, node('<typename>',X,Z,P), node('<classtype>',Z,W,_), node('<classinstancecreationexpression>',W,_,_), type(Name,S), node('<classdeclaration>',S,_,_), node('<classmodifiers>',U,S,_), node('<classmodifiers>',R,U,_), node('<classmodifier>',T,R,_), \+ node('abstract',_,T,_).

Table 5.4: Abstract class rules

JLS 8.1.1.1 - final class

In Table 5.5 the JLS rule states that a CTE occurs if a final class has a subclass. Then in Prolog, when adding a superclass, the class modifier of the class is found. If the keyword *final* is found, then Prolog returns false.

JLS Rules	Prolog Rules
”§8.1.1.1: if a class is declared final, it cannot have subclasses.”	useIdentifier(Name,X,Y) :- P is Y-1, node('<typename>',X,Z,P), node('<classtype>',Z,W,_), node('<super>',W,S,_), node('<classdeclaration>',S,_,_), node('<classmodifiers>',U,S,_), node('<classmodifiers>',R,U,_), node('<classmodifier>',T,R,_), \+ node('final',_,T,_).

Table 5.5: Final class rules

JLS 8.3 - field modifier

In Table 5.6, a CTE happens if the same keyword is employed more than once in a field declaration, i.e., for the fields of a class. In Prolog, we have the example of the keyword *public*, but for the other modifiers, the rule has been added. Prolog looks for other modifiers of the field. If they are found it returns false, otherwise it returns true.

JLS Rules	Prolog Rules
”§8.3: compile error if the same keyword appears more than once as a modifier for a field declaration.”	useModifier('public',_,X,Y) :- Z is Y-1, node('<fieldmodifier>',X,_,Z), \+ node('public',_,X,Y), \+ node('private',_,X,Y), \+ node('protected',_,X,Y)

Table 5.6: Field modifier rules

5.2.3 Rules about inheritance between classes

JLS 8.1.4.1 - class inheritance direct cycle

In Table5.7, a CTE occurs if a class depends on itself. We model the rule in Prolog in two different situations: when a class directly extends itself and when a class indirectly extends itself and raises a class circularity error (next rule).

JLS Rules	Prolog Rules
”§8.1.4.1: CTE if a class depends on itself.”	canExtend(A,B) :- \+(A == B).

Table 5.7: Inheritance rules

JLS 8.1.4.2 - class circularity error

In Table 5.8 the class circularity error is handled. Here we show the first example of recursion: the Prolog rules are two because one represents the base case and the other represents the recursive call.

JLS Rule	Prolog Rules
”§8.1.4.2: if circularly declared classes are detected at run-time, as classes are loaded, then a ClassCircularityError is thrown.”	classCircularity(B,A) :- extends(B,A). classCircularity(B,A) :- extends(B,C), classCircularity(C,A).

Table 5.8: Class circularity error rules

5.2.4 Rules about formal parameters

JLS 8.4.1 - formal parameters

In Table 5.9 the JLS rule states that in a method or constructor a compile-time error occurs if the same name appears more than once in the formal parameters.

JLS Rule	Prolog Rules
”§8.4.1: CTE for a method or constructor to declare two formal parameters with the same name.”	formalParams(NameLst) :- is_set(NameLst)

Table 5.9: Formal parameters rules

JLS 8.4 - method signature

As JLS 8.4 says, it is not possible to have two methods with override-equivalent signatures in the same class. Thus, in Table 5.10 Prolog checks if there are methods with the same override-equivalent signature inside the same class.

JLS Rule	Prolog Rules
”§8.4: CTE for the body of a class to declare as members two methods with override-equivalent signatures.”	useMethod(Name,ClassName,RetType, F_Lst) :- \+method(Name,_,ClassName, RetType,F_Lst).

Table 5.10: Method signature rules

JLS 8.8.2 - constructor signature

In Table 5.11 the corresponding JLS rule states that it is a CTE if two constructors with override-equivalent signatures are declared in the same class. In the Prolog rule, there is a check whether in the KB there is another constructor with the same F_Lst, i.e., the same types, in the same class, i.e., with the same *Name*.

JLS Rule	Prolog Rules
”§8.8.2: CTE to declare two constructors with override-equivalent signatures.”	useConstructor(Name,F_Lst) :- \+constructor(Name,_,F_Lst).

Table 5.11: Constructor signature rules

5.2.5 Rules about explicit constructor invocations

JLS 8.8.7 - explicit constructor invocation using *this*

Inside constructor bodies, the Java language allows to explicitly invoke other constructors at the beginning of the body implementation. This rule states that it is not possible for a constructor to directly or indirectly call itself using the keyword *this*. In Table 5.12 the Prolog rule checks with the predicate *sameArgs* if the types of the argument in the constructor invocation are the same as the constructor signature in which the invocation is done.

Circularity of invocation Prolog also checks if there is not a circularity invocation, i.e., if the constructor that the invocation is calling has, in turn, the invocation of this constructor. In Prolog, we model this using the predicate *thisInvocation*, which is added as a fact every time this rule validates an explicit constructor invocation involving *this*.

JLS Rule	Prolog Rules
"§8.8.7: CTE for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving *this*."	useThisInvocation(Args,ParId,Depth) :- \+sameArgs(Args,ParId,Depth), lookForNode(' <constructor declaration>' ,X,_,_,ParId,Depth), constructor(Class,X,ArgLst), \+thisInvocation(Class,ArgLst), constructor(Class,_,Args).

Table 5.12: Explicit constructor invocation using *this* rules

JLS 8.8 - Explicit constructor invocation using *super*

In section 8.8 of the JLS, there are some considerations about the use of *super* constructor invocations. One of the most straightforward rules is presented in Table 5.13, where the JLS asserts that it is not possible to call a super constructor invocation if the superclass does not have constructors with the same formal parameter types of the argument types of the constructor invocation. Prolog checks this by looking at the superclass of the class in which the invocation is called.

JLS Rule	Prolog Rules
"8.8: CTE for a constructor to make an explicit constructor invocation using the keyword *super* whose arguments are not of the same type of one of the superclass constructor formal parameters."	useSuperInvocation(Args,ParId,Depth):- lookForNode(' $< constructordeclaration >$' ,X,_,_,ParId,Depth), constructor(Class,X,_), extends(Class,Y), constructor(Y,_,Args)

Table 5.13: Explicit constructor invocation with *super* rules

5.2.6 JLS 14.7 - Return statement expression

Return statement result type

The *Expression* type in the return statement inside a method *m* must match the return type of *m*. Prolog checks if the type *RetType* of the query matches with the result type of the method enclosing the return statement. See Table 5.14.

JLS Rule	Prolog Rules
”14.7: CTE occurs when a *return* statement with an *Expression* appears in a method declaration, the *Expression* is not assignable to the declared return type of the method.”	useRetStmt(RetType,ParId,Depth) :- lookForNode('<method declaration>' ,X,_,_,ParId,Depth), method(_,X,_,RetType,_).

Table 5.14: Return statement expression rules

5.2.7 JLS 15.9 - Class instance creation expression

Correct argument types for class instance creation expression

In Table 5.15 we enforce the rule about the correspondence between argument types of the class instance creation expression and the argument types of one of the constructors of the class instantiated. Prolog checks if there is a constructor with the same *Class* name and with the same *Args*, i.e., the same argument types.

JLS Rule	Prolog Rules
”15.9: CTE if a class instance creation expression contains argument types that are not attributable to a constructor of the instantiated class.”	useClassInstanceCreation(Class,Args) :- constructor(Class,_,Args).

Table 5.15: Class instance creation expression rules

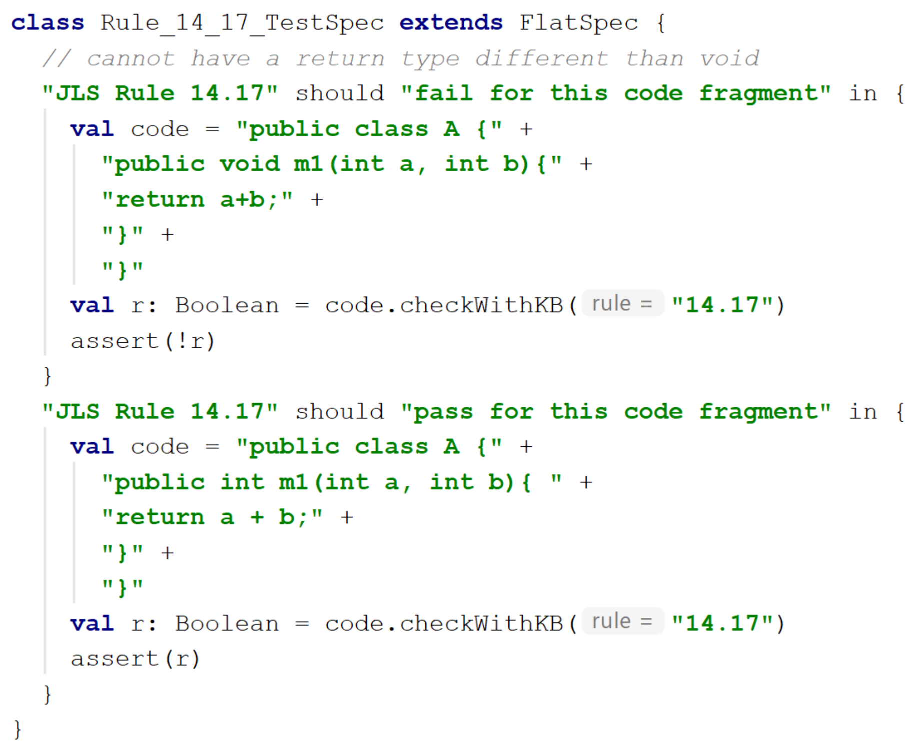
5.3 Testing Prolog rules

For testing Prolog rules and making sure that they work as expected, we set up a testing process using FlatSpec, a Scala testing framework and employed a code parser given by the *org.eclipse.jdt.core.dom* library distributed by Eclipse [18].

5.3.1 Testing process

The testing process is done in the following way:

- For every JLS rule, two code fragments are given: one fragment is syntactically correct, the other is incorrect.
- Each code fragment is parsed by the Eclipse *ASTParser* to create an AST representation of the code fragment.
- A visitor pattern is then applied to the AST through the class *ASTVisitor*. The visitor visits the nodes of interest and constructs the query for the Prolog KB.
- When the visitor visits all the nodes and the query is complete, the Prolog KB is queried and the result is stored.
- The FlatSpec test framework asserts for the correct code fragment that the Prolog response is true and for the incorrect code fragment that the response is false.



```
class Rule_14_17_TestSpec extends FlatSpec {
  // cannot have a return type different than void
  "JLS Rule 14.17" should "fail for this code fragment" in {
    val code = "public class A {" +
      "public void m1(int a, int b){" +
      "return a+b;" +
      "}" +
      "}"
    val r: Boolean = code.checkWithKB( rule = "14.17")
    assert(!r)
  }
  "JLS Rule 14.17" should "pass for this code fragment" in {
    val code = "public class A {" +
      "public int m1(int a, int b){ " +
      "return a + b;" +
      "}" +
      "}"
    val r: Boolean = code.checkWithKB( rule = "14.17")
    assert(r)
  }
}
```

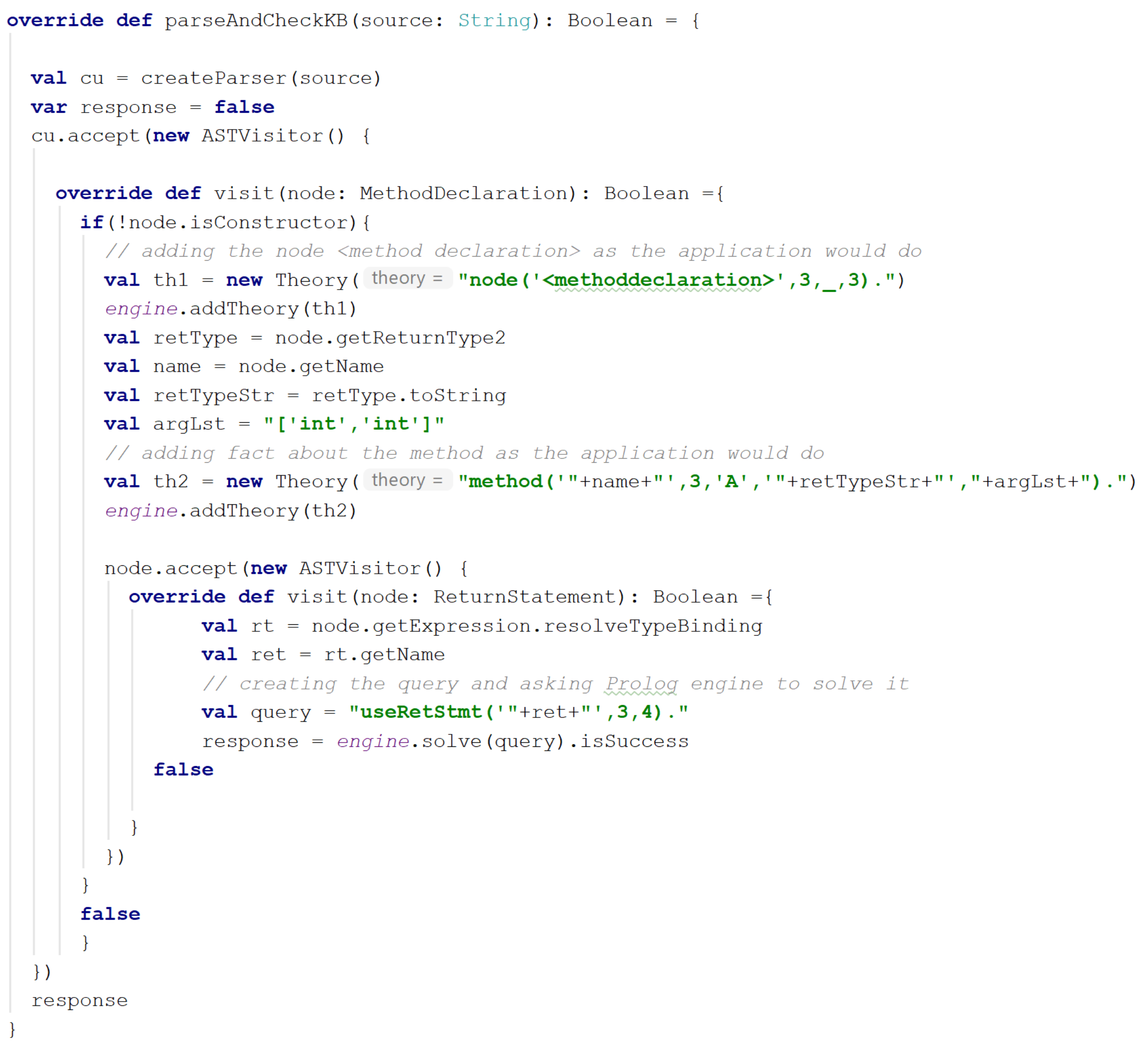
Figure 5.2: Testing the Prolog rule on *return* statement on two code fragments

5.3.2 Example of test

We show in Figure 5.2 the test on the JLS rule 14.17 for the *return* statement with two different code fragments: the first fragment should fail because the method $m1$ has a *void* declared type but the *Expression* of the *return* statement returns an integer (the sum of the two integer parameters of the method). The result from the second code fragment should instead return true, as the method return type is the same as the *Expression* of the *return* statement. The test process, given the rule number 14.17, instantiates the correct function that looks for the information to construct the *useRetStmt* Prolog query.

ASTParser

In Figure 5.3 we show how the code is parsed by the Eclipse *ASTParser* to look for the information needed by the Prolog KB. To recreate the same working environment of the real application, the node *MethodDeclaration* is visited through the parser. With the information of the AST, we can create and add the Prolog facts of *node*/4 and *method*/5 for the related method. Then with a new visitor, the *ReturnStatement* node is visited, and the query is created. Finally, the Prolog engine solves the query and returns the response.



```
override def parseAndCheckKB(source: String): Boolean = {

  val cu = createParser(source)
  var response = false
  cu.accept(new ASTVisitor() {

    override def visit(node: MethodDeclaration): Boolean ={
      if(!node.isConstructor){
        // adding the node <method declaration> as the application would do
        val th1 = new Theory( theory = "node('<methoddeclaration>',3,_,3).")
        engine.addTheory(th1)
        val retType = node.getReturnType2
        val name = node.getName
        val retTypeStr = retType.toString
        val argLst = "['int','int']"
        // adding fact about the method as the application would do
        val th2 = new Theory( theory = "method('"+name+"',3,'A','"+retTypeStr+"',"+argLst+").")
        engine.addTheory(th2)

        node.accept(new ASTVisitor() {
          override def visit(node: ReturnStatement): Boolean ={
                val rt = node.getExpression.resolveTypeBinding
                val ret = rt.getName
                // creating the query and asking Prolog engine to solve it
                val query = "useRetStmt('"+ret+"',3,4)."
                response = engine.solve(query).isSuccess
            false

          }
        })
      }
      false
      }
  })
  response
}
```

Figure 5.3: Parsing the code fragment with Ecplipse library and checking with Prolog KB

5.4 gRPC: communication between the application and the Prolog KB

As already said in the previous part, one reason for giving a formal description of the Java Language Specification through Prolog rules is the separation of concerns. Thus, in our work, we want that the application and the Prolog KB communicates through a client-server interaction. The application accesses and queries the KB as if it were a server. The framework we decided to use is gRPC, a universal open-source Remote Procedure Call framework for exchanging messages in microservice architectures.

5.4.1 RPC

RPC stands for Remote Procedure call: when a computer program causes a subroutine or procedure to run in a different address space, this call is implemented as if it were a normal (local) procedure call, thus hiding the interaction implementation details. RPC's are a form of IPC, i.e., inter-process communication, in that different processes have different address spaces. If the processes are on the same machine, they have different virtual address spaces. When the processes are on different machines, their physical addresses are different. Whenever our application wants to make an interaction with the Prolog KB (e.g., adding a fact or asking a query to be solved), a remote procedure call is done to one of the Prolog Server's methods that computes the result and returns it to the application.

5.4.2 Properties of gRPC

gRPC is a high-performance framework developed by Google that has many advantages: it can run in any environment thanks to protocol buffers, a way for serializing structured data without depending on language and platforms. Protocol buffers transport messages bidirectionally between client and server for implementing the remote procedure calls.

Defining a gRPC service

For defining a gRPC service, a *.proto* file has to be created for specifying the definition of the remote procedures and the messages that you want to pass through protocol buffers. In Figure 5.4 we show an extract of the file defining the service and the messages: the remote procedures are enclosed by the keyword *service* and then the messages are defined. The messages are in correspondence with the Prolog predicates that the messages represent: for example the message *Node* has four attributes as well as its Prolog counterpart, the predicate *node*/4. For the purpose of passing the messages from client to server and vice-versa, a protobuf compiler has been used for generating the Scala classes that represent messages. We defined our PrologValidation gRPC service along with the messages in a different project, assembled it into a Jar container and included as a dependency in the main project. Thus, the client in the application is free to instantiate as many messages it wants with the imported Scala classes and call the remote procedures.

```
syntax = "proto3";

service PrologValidation {
    rpc addFactNode (Node) returns (Response) {}
    rpc validateExtension (Extend) returns (Response) {}
    rpc validateMethodSign (MethodSign) returns (Response) {}
    rpc validateConstructorSign (ConstrSign) returns (Response) {}
    rpc validateThisInvocation (ThisInvocation) returns (Response) {}
    rpc validateUseRetStmt (UseRetStmt) returns (Response) {}
    rpc validateRetStmt (RetStmt) returns (Response) {}
    rpc validateSuperInvocation (SuperInvocation) returns (Response) {}
    rpc validateMethodInvocation (MethodInvocation) returns (Response) {}
    rpc validateClassInstanceCreation (ClassCreation) returns (Response) {}
    rpc addType (Type) returns (Response) {}
    rpc getType (Type) returns (TypeResponse) {}
}

message Node{
    string description =1;
    int32 id = 2;
    int32 par_id = 3;
    int32 depth = 4;
}

message Extend{
    string typeName = 1;
    string superName = 2;
}

message MethodSign{
    string name = 1;
    string className = 2;
    string returnType = 3;
    string paramLst = 4;
    int32 id = 5;
}
```

Figure 5.4: An extract of the *validation.proto* file for defining the gRPC service *PrologValidation*

Chapter 6

Conclusion and future works

In this work, we presented ProGen, a functional programming framework for generating random Java programs with Prolog-Based Grammar Generation. The goal of this work is to offer a new approach for automated testing towards compilers verification. In section 6.1 we present our results by showing an example of a generated class.

6.1 Results and Evaluations

The framework generates syntactically correct but semantically meaningless programs with the following properties:

- Classes with inheritances.
- Classes implementing interfaces.
- Constructors with explicit constructors invocations.
- Method invocations.
- Explicit constructor invocations using *this* and *super* keywords.
- Classes instance creation expressions using the keyword *new*.
- Interfaces with method signatures.
- Local variable declarations.
- Local variable initializations.

6.1.1 Example of a generated class

In this subsection, we show how a generated class looks after the successful execution of ProGen. The example is the result of the generation of class *A*. This class implements two interfaces, *Inter2* and *Inter3*.

Constructors

The following first part shows the implementation of the constructors. We can see that in the first and the second constructor there is an explicit constructor invocation using *this* calling the fourth construction. The Prolog rules have validated these invocations. Since class A does not inherit from another class, it is forbidden to use an explicit constructor invocation using *super*. In the body of the constructors, there are also local variable declarations, local variable initializations, and class instance creation expressions.

```
class A implements Inter2 , Inter3  {
         A(M param0,L param1,I param2){
             this(param2);
             byte b;
         }
     A(L param0,I param1,N param2,I param3){
             this ( param1 ) ;
             L t = new L(param2,param1);
             char u,f = 'l';
             {
             }
         Inter4 w,g,q,b,k,i,l,j;
             Inter6 p;
         }
     A(I param0,M param1,I param2,L param3,N param4)
         {
          Inter1 i,h,w;
         }
     A(I param0){
          short i = 2;
         {
             {
                  O s = new O(field4);
             }
         byte m,q;
             int l,g = 48,v = 35;
             {
                  {
                       Inter6 n,o,b,e,z,h;
                  }
             new F(g);
                  long o,k,e = 99,j,t,h = 9008,d,r;
             }
     }
 }
```

Fields and methods

The second part of class A shows the generation of fields and the generation of methods. As regards methods generation, we see that method m8, and m9 are overridden even if the framework does not generate Java annotations. The annotation is there for showing that the methods are defined in the interfaces *Inter*1 and *Inter*2 that the class implements. For every method signature containing a return type, there is a return statement. Finally, method invocations are used for statement expressions. In other classes, method invocations are also used for return statements.

```
M field1;
I field0,field2;
L field4;
N field3;

byte m31(A param1,double param2){
    new I(field1);
        Inter3 j,q,g;
        return 0;
}

boolean m32(char param1,float param2){
    M i;
        double h;
        A t;
        C d;
        byte g;
        new A(field0,field1,field0,field4,field3);
        return false;
}

G m33(O param1){
    return new G();
}

L m34(F param1,O param2,double param3,G param4){
    {
        }
        return new L(field4,field1,field0,field0);
}

boolean m35(F param1,G param2){
    new O(field4);
    return false;
}
@Override
public A m7(M param1,byte param2,N param3){
    return new A(field0,param1,field0,field4,param3);
}
```

```
@Override
public boolean m8(short param1,O param2){
          Inter2 l;
          new L(field3 ,field0 );
          m19(param2,param1, field4 );
          new H ();
          {
              {
                   {
                   }
              Inter4 i,j;
              }
         double k = 0.21195391444483602;
              {
                   m7 (field1 ,(byte)0,field3 );
                   {
                        A u,r,p;
                   }
              Inter4 w,s,c,z;
              }
     }
     return false;
}
```

6.1.2 Example of a generated interface

The example below shows a generated interfaces with the defined method signatures.

```
 interface Inter6 {

     float m14(float param1,short param2,H param3);

          boolean m15(long param1,C param2,A param3);

          L m16(short param1,float param2,H param3);

          O m17(G param0, boolean param1);
}
```

6.1.3 Application profiling

To get an idea of the characteristics of the application, we monitored the JVM activity during its execution with the usage of different tools available in the Java Development Kit, and we found the following results.

Execution time

We collected execution times of the application with different size of the generated programs, from a size of 100 lines of code to a size of 1000 lines of code. We witnessed that there is a non-negligible variance of execution time due to the non-deterministic nature of the random generator. Thus, this result may be less indicative than other parameters of the application. At the same time, the execution times we collected were higher than the expectations, with an average of 7-8 seconds per line of code generated.

Processor Usage

With the use of VisualVM it is possible to monitor the CPU usage of the application dynamically. Figure 6.1 shows a caption of the CPU monitoring when the application started the execution. The processor has its peak at the beginning of the application (50%), but then it starts to decrease until reaching values between 10% and 20% and remains constant. We can conclude that our application is not computationally expensive in terms of processor usage.

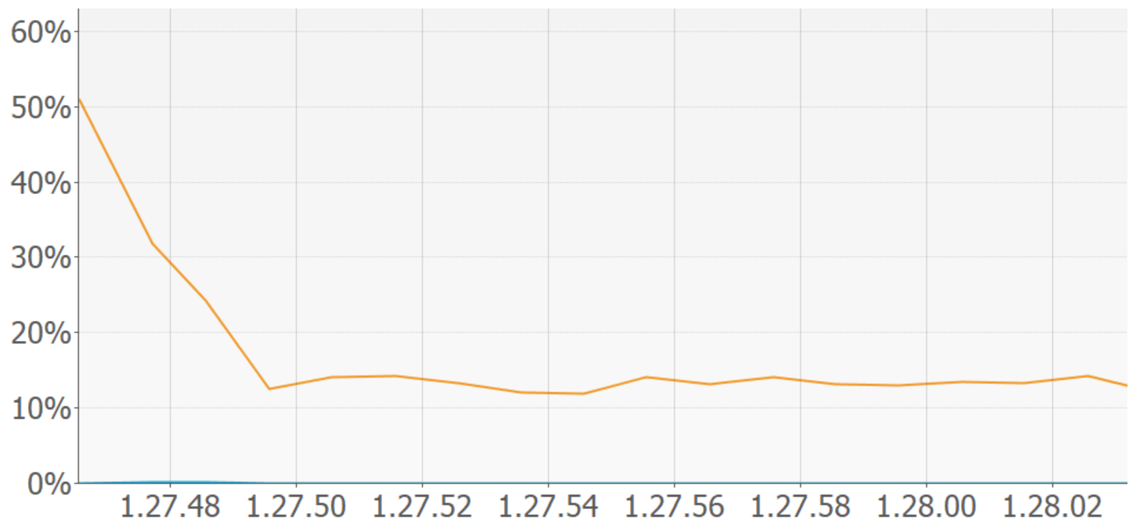


Figure 6.1: Processor usage in the application

Heap memory usage

The heap memory is the part of the memory where the JVM allocates the object instantiated. Heap memory usage is presented in Figure 6.2. We can see that the heap contains a minimum of 160 MegaBytes and a maximum of 416 MegaBytes. When in the diagram the usage decreases, it means that the Garbage Collector of the JVM is freeing the objects no longer referenced. We witness a usage of the memory heap which is relatively healthy, given that the percentage usage is 56%. The results are collected using the MBean Server of Java Mission Control.

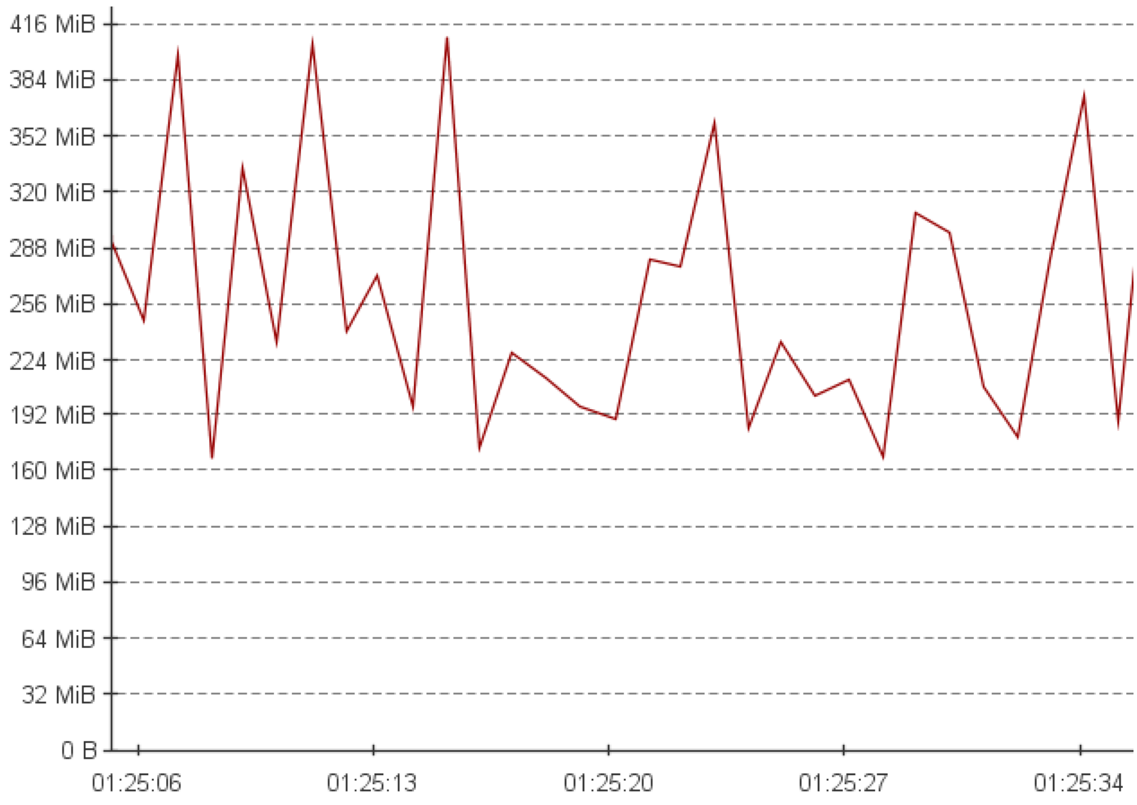


Figure 6.2: Heap memory usage in the application

Threads

We inspected the active threads during the execution and their behavior using VisualVM. Figure 6.3 shows the states of the four threads executed by the executor.

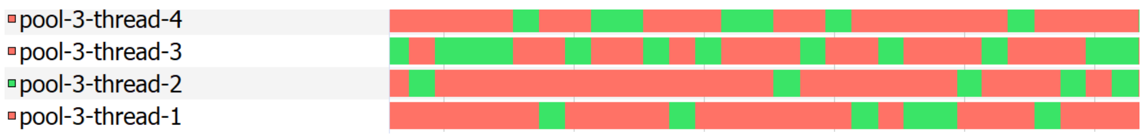
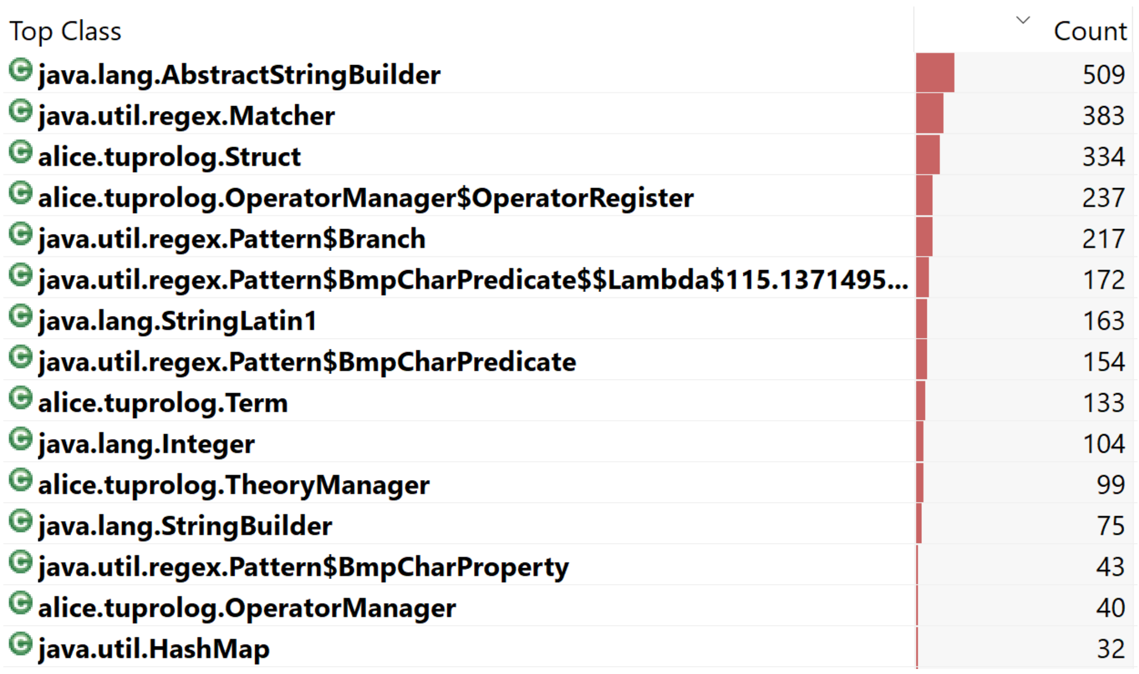


Figure 6.3: Threads states in the application

Method Profiling

To collect detailed runtime information, we also performed a *Flight Recording* of the application with Java Mission Control. In particular, an impressive result comes out from the *Method Profiling* tool of the *Flight Recorder*. The tool collects information about what classes and packages are used the most by the methods running during the recording. We show in Figure 6.4 the top classes used by the methods in the application. It is worth to notice that out of the 15 classes most used, a third of them (5) are belonging to the package *alice.tuprolog*, which is the library used for running the Prolog engine. This package is the most used package during the flight recording, with a total of 894 usages.



Top Class	Count
java.lang.AbstractStringBuilder	509
java.util.regex.Matcher	383
alice.tuprolog.Struct	334
alice.tuprolog.OperatorManager$OperatorRegister	237
java.util.regex.Pattern$Branch	217
java.util.regex.Pattern$BmpCharPredicate$$Lambda$115.1371495...	172
java.lang.StringLatin1	163
java.util.regex.Pattern$BmpCharPredicate	154
alice.tuprolog.Term	133
java.lang.Integer	104
alice.tuprolog.TheoryManager	99
java.lang.StringBuilder	75
java.util.regex.Pattern$BmpCharProperty	43
alice.tuprolog.OperatorManager	40
java.util.HashMap	32

Figure 6.4: Method profiling during a flight recording

6.2 Limitations and future work

Performance issues

From the previous section related to the application monitoring, we can conclude that ProGen is not computationally expensive in terms of processor and memory usage, but it takes much time to generate random programs. The reasons for this problem can be of two different natures. The first reason is that the generation is random and the abstract syntax tree is constructed and pruned every time the Prolog check does not validate the generation. The second reason is that the Prolog engine receives many queries by the different threads and as we can see from the method profiling the *tuprolog* library packages are heavily used, forming a bottleneck for the execution. Future works rely on designing a better multithreading approach and on minimizing the random attempts of the application.

Expanding language features

The next important step for future works is to generate Java modifiers, recursion, and control flow statements. Each of these features requires more precise design and a rethinking in terms of interaction with the Prolog KB. Although the framework can generate large size random programs (to the order of thousands of LOC), the framework still does not generate enough richness in the Java realm. Thus, the direction of future work is in enriching the generated programs through two different approaches.

Enriching the Prolog KB Improving the level of detail of Prolog rules and modeling more JLS rules are essential for enriching the Prolog KB. As of now, we model the generated AST with Prolog facts, but there is a necessity to have more rules that can exploit the model. This approach is the most scalable one and does not add complexity on the application side.

Improving the weighted sampling The approach of using probabilities on the grammar has been used in previous works [34, 32, 38, 44, 10, 37, 43, 16]. This approach is useful for guiding the generation towards a correct generation, but also to perform statistical simulation using with different probability distributions. Thus, the next step in this approach is to enhance weighted sampling for both guiding the generation and perform statistical sampling over probability distributions.

6.2.1 Learning to write programs with Machine Learning

In the last years, research also focused on the generation of meaningful programs with the usage of neural networks and deep learning techniques using input-output examples [7], [20], [29]. The results that can be achieved by our framework could be of help to discover new possibilities for training a machine learning model in the generation of meaningful programs. Although this could seem like a futuristic approach, we believe that the use of existing correct programs could be used to generate meaningful programs, or at least provide useful insights for this research area.

Appendix A

Software and resources used

- Application built with SBT (http://www.scala-sbt.org/download.html), version 1.1.1.
- Installed Java Development Kit, version 9.0.1.
- Scala version 2.12.4 with the IntelliJ Idea Scala plugin.
- TuProlog API for Prolog engine: a Java-based light-weight for Internet applications and infrastructures.
  API: http://tuprolog.sourceforge.net/doc/api/alice/tuprolog/package-sum mary.html
- IDE used: IntelliJ IDEA 2018.2.4 (Ultimate Edition) Build #IU-182.4505.22, built on September 18, 2018 Licensed to Giovanni Agugini.JRE: 1.8.0_152-release-1248-b8 amd64. JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0
- Generex: a library for generating strings from regex expressions. Version 0.0.2. GitHub: https://github.com/mifmif/Generex.
- Java Mission Control: profiling tool for JVM activity.
- Java VisualVM: profiling tool for JVM activity.
- FunSuite: Scala testing framework from the library org.scalatest, version 3.0.1
- FlatSpec: Scala testing framework from the library org.scalatest, version 3.0.1.
- Grizzled-SLF4J: a Scala-friendly SLF4J wrapper for application logging.
  http://software.clapper.org/grizzled-slf4j/.
- Breeze: numerical processing library for performing combinatorial calculation. GitHub: https://github.com/scalanlp/breeze.
- Lightbend config: a configuration library for JVM languages.
  GitHub: https://github.com/lightbend/config.
- gRPC: a high perfomance, open-source universal RPC framework (https://grpc.io/).

- ScalaPB: a Protocol buffer compiler for Scala.
  GitHub: https://scalapb.github.io/.
- GitHub repository of the framework:
  https://github.com/GioAgu17/JavaProGen.
- Github repository of the Protocol buffer service definition:
  https://github.com/GioAgu17/PrologProtobuf.

Bibliography

[1] The fundamentals of prolog. http://www.doc.ic.ac.uk/~cclw05/topics1/prolog2.html. Accessed: 2019-01-15.

[2] Gcc soars past 14.5 million lines of code & i'm real excited for gcc 5. https://www.phoronix.com/scan.php?page=news_item&px=MTg3OTQ. Accessed: 2019-01-30.

[3] The language of languages. http://matt.might.net/articles/grammars-bnf-ebnf/. Accessed: 2018-11-28.

[4] Llvm is at nearly 2.5 million lines of code. https://www.phoronix.com/scan.php?page=news_item&px=MTU1MzY. Accessed: 2019-01-30.

[5] Tiobe index for january 2019. https://www.tiobe.com/tiobe-index/. Accessed: 2019-01-25.

[6] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[8] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[9] Abdulazeez S Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.

[10] Colin J Burgess. The automated generation of test cases for compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, 1994.

[11] Augusto Celentano, S Crespi Reghizzi, P Della Vigna, Carlo Ghezzi, G Granata, and Florencia Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10(11):897–918, 1980.

[12] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Notices*, volume 48, pages 197–208. ACM, 2013.

[13] Paul Chiusano and Rúnar Bjarnason. *Functional programming in Scala*. Manning, 2015.

[14] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[15] Roger F Crew et al. Astlog: A language for examining abstract syntax trees. In *DSL*, volume 97, pages 18–18, 1997.

[16] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium*, pages 120–125. Springer, 2012.

[17] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194. ACM, 2007.

[18] Jim des Rivières. Eclipse apis: Lines in the sand. *EclipseCon Retrieved March*, 18:2004, 2004.

[19] Frank Flederer, Ludwig Ostermayer, Dietmar Seipel, and Sergio Montenegro. Source code verification for embedded systems using prolog. *arXiv preprint arXiv:1701.00630*, 2017.

[20] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

[21] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification, java se 8 edition (java series), 2014.

[22] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[23] Jörg Harm and Ralf Lämmel. Two-dimensional approximation coverage. *Informatica*, 24(3):355–369, 2000.

[24] Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Qing Xie, Sangmin Park, Kunal Taneja, and Mainul Hossain. Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Software: Practice and Experience*, 46(3):405–431, 2016.

[25] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[26] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. The java language specification, 2000.

[27] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *IFIP International Conference on Testing of Communicating Systems*, pages 19–38. Springer, 2006.

[28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[29] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[30] Nuno Lopes. Getting compilers right: a reliable foundation for secure software. https://www.microsoft.com/en-us/research/blog/getting-compilers-right-secure-software/, 2017. [Online; accessed 03/20/2018].

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[32] Peter M. Maurer. Generating test data with enhanced context-free grammars. *Ieee Software*, 7(4):50–55, 1990.

[33] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[34] V Murali and RK Shyamasundar. A sentence generator for a compiler for pt, a pascal subset. *Software: Practice and Experience*, 13(9):857–869, 1983.

[35] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[36] Flash Sheridan. Practical testing of a c99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[37] Emin Gün Sirer and Brian N Bershad. Testing java virtual machines. In *Proc. Int. Conf. on Software Testing And Review*, 1999.

[38] Emin Gün Sirer and Brian N Bershad. Using production grammars in software testing. In *ACM SIGPLAN Notices*, volume 35, pages 1–13. ACM, 1999.

[39] Mathias Soeken and Rolf Drechsler. Grammar-based program generation based on model finding. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–5. IEEE, 2013.

[40] K Sreenivasan and AJ Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, 1974.

[41] Katherine E Stewart and Steven W White. The effects of compiler options on application performance. In *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*, pages 340–343. IEEE, 1994.

[42] Frank Van Harmelen and Dieter Fensel. Formal methods in knowledge engineering. *The knowledge engineering review*, 10(4):345–360, 1995.

[43] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[44] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. Random program generator for java jit compiler test system. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 20–23. IEEE, 2003.

[45] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 36–43. IEEE, 2009.