University of Bergen
Department of Informatics

Generating sample code snippets from Lark grammar specifications

Author: Maren Holm Hundvin
Supervisors: Mikhail Barash & Anya Helene Bagge

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

October, 2022

Abstract

Programming languages are the basis of computer science and are extensively used in computer science education. Many educational materials, such as tutorials, constantly appear on the market with the goal of targeting various user groups. These tutorials, however, oftentimes present language constructs in inconsistent ways, with constructs completely missing from some of the tutorials. This makes it difficult for a language learner to get a holistic overview of the language.

In this thesis, we focus on syntax of programming languages, specified in a form of context-free grammars, and design an algorithm to generate syntactically correct sample code snippets written in the language in question. Our algorithm can be used to comprehensively explore constructs of a programming language. We test our prototype implementation of the designed algorithm on several grammars, including a tiny model programming language and a children-oriented programming language Hedy.

Acknowledgements

First and foremost, I would like to thank my supervisors, Mikhail Barash and Anya Helene Bagge, for all their help, support, and guidance thorough with this thesis. The regular meetings, and their knowledge has been a big help in getting here today.

I would also like to thank my fellow students for their support, great environment, and many breaks that have kept me sane during this process.

Finally, I would like to thank my family for all their support throughout my studies and for motivating me to keep going. To Lars Siri, my parents, I would not have made it here without you. And to my grandmother, who has been a pioneer in computer science, for being an inspiration and for proofreading my thesis.

Maren Holm Hundvin
Tuesday 25$^{\text{th}}$ October, 2022

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Programming languages are the basis of computer science and are extensively used in computer science education, from school pupils to universities to vocational education. To tackle the abundance of existing programming languages, many educational materials (e.g., tutorials) constantly appear with the goal of targeting various user groups. Unfortunately, these tutorials often explain different language constructs in inconsistent ways, and in some of the tutorials, certain constructs are completely missing. This makes it difficult for language learners to get a holistic overview of the language.

This thesis aims to develop an algorithm for generating sample snippets of code based on a grammar specification for that language. The algorithm can then be used to explore complete constructs of a programming language.

The formalism we work with in this thesis is context-free grammars in Extended Backus-Naur Form (EBNF). These grammars are the natural choice since they are *de facto* the standard formalism used to specify syntax of programming languages. This formalisms is extensively discussed in numerous textbooks on compiler construction and formal languages, among others. The EBNF form for context-free grammars is the most widespread notation used to define context-free grammars.

The omnipresence of context-free grammars and EBNF in computer science means that there are many formats to represent them. In this thesis, the representation we choose is *Lark* [15]. Lark is a Python-based parser generator, meaning it has all the functionality we were looking for. It has functionality for creating a parser from a grammar, the parser can then be used to parse the input grammar, giving us a parse tree. Another functionality is that we can traverse the parse tree, building a new, more convenient,

representation of the grammar for us to work with. The fact that Lark is Python-based also factored into our choice, as this shall lower the entry bar for grammar- and parser-developers. Another factor for basing our work on Lark is that it becomes more and more widely used within academic community; for instance, the levels of the gradual programming language Hedy [6] are specified in Lark.

We now showcase the machinery of the algorithm that we developed in this thesis. The input of the algorithm is a context-free grammar in the Lark format, as given below. The grammar specifies a tiny programming language with variable declarations, assignment statements, while loop, and simple arithmetic expressions over integer numbers.

```
start: program

program: "program" ID "{" (var_decl ";")* (statment";")* "}"

var_decl: "var" ID
        | "let" ID
        | "const" ID

statement: assign_statement
         | write_statement

assign_statement: ID "=" expr

write_statement: "write" expr

expr: NUM
    | expr "+" expr
    | expr "-" expr
    | expr "*" expr
    | expr "/" expr

NUM: \[0-9]\
ID: \[A-Za-z]\
```

When executed, our algorithm processes the grammar in multiple stages, and then outputs a sample code snippet for the starting symbol `start`.

```
program A {
    var x;
    const x;
    write Y;
    h = 1;
}
```

As can be seen from this sample generated snippet, the algorithm only focuses on the syntax of the language—any generated snippet will comply with the grammar specification and thus be syntactically correct. Semantic correctness, such as the requirement that all variables have been declared prior to use, or that there are no duplicate declarations, are not satisfied, as they are not specified in a context-*free* grammar.

The thesis is organized as follows. **Chapter 2** presents an introduction to formal grammars: we focus on formalism of our choice—context-free grammars, as well as on the Lark parser generator, which will be used in course of implementation. We then present in **Chapter 3** the designed algorithm for generating sample code snippets and showcase how it is applied to a running example of a grammar. We have implemented the algorithm in Python; we focus on the its implementations details in **Chapter 4**. **Chapter 5** then discusses the related work: other grammar formalisms, the idea of grammarware [9], "Query by Example" and existing tools for learning programming. Finally, we discuss our approach and outlines some directions for future work in **Chapter 6**.

Chapter 2

Background

2.1 Grammars

A language consists of two parts, semantics and syntax. Grammar is the syntax of the language. It is a set of rules that tells us how to build a valid string for the language. However, it is the semantics that gives meaning to the string.

A grammar is a tuple $G = \langle T, N, P, S \rangle$, where T is the set of terminals, N is the set of nonterminals, P is the set of production rules, and S is the start symbol in the grammar. The terminals T of the grammar include all the strings and tokens. The nonterminals N are the names of the production rules P, which are defined as sequences of terminals and nonterminals. The start symbol S is the nonterminal that all valid terminal strings can be generated from.

There are different classes of grammar as described in Chomsky hierarchy [7]. Chomsky hierarchy divides grammars into four types: Type 0: Unrestricted Grammar, Type 1: Context Sensitive Grammar, Type 2: Context-Free Grammar, and Type 3: Regular Grammar.

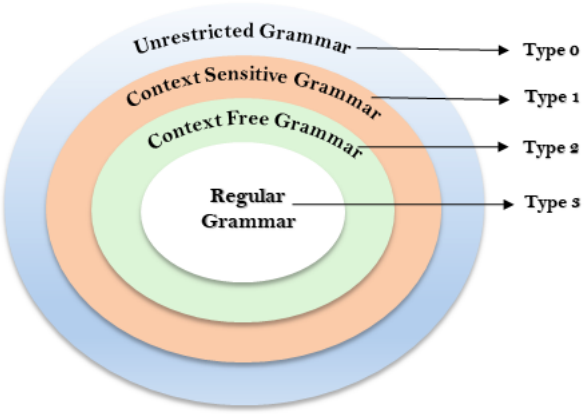

Fig: Chomsky Hierarchy

Figure 2.1: Chomsky hierarchy

Credit: JavaTPoint `https://www.javatpoint.com/automata-chomsky-hierarchy`

As one can see from Figure 2.1, type 0 is the most general class, and type 3 is the most restricted. Every class of a higher type have the same requirements as the ones of lower types. This means that a grammar of type 3 will also go under types 0, 1, and 2.

$A, B \in N$
$a, b \in T$
$\alpha \in (T \cup N)^*$
$\beta \in (T \cup N)^+$

Unrestricted Grammar or Recursively Enumerable language has no restrictions on the rules of the grammar, as one can tell from the name. It only requires the left-hand side of a production not to be empty, it must contain a nonterminal.

$A \to B$
$Ab \to \alpha$

Context Sensitive Grammar requirements are that neither the left-hand side nor the right-hand side of a production can be empty. The left-hand side must also have fewer symbols than the right-hand side.

$A \to B$
$Ab \to \beta$ – where $\mid \beta \mid > \mid Ab \mid$

In Context-Free Grammar, the left-hand side can only consist of one nonterminal, and the right-hand side can contain anything.

$A \to \alpha$

$B \rightarrow \beta$

Regular Grammar, as mentioned, is the most restricted grammar. Here production rules generate regular languages, languages that can be described by a regular expression. In Regular grammar, there are two forms, left-regular grammar and right-regular grammar.

$A \rightarrow Bb$
$A \rightarrow bB$

In this thesis, we are focusing on context-free grammars, two common ways of presenting context-free grammars are Backus-Naur Form (BNF), and Extended Backus-Naur Form (EBNF). These are ways of representing grammar, the syntax of the grammar. As one can tell by their names, EBNF is the same as BNF only with more *syntactic sugar*. Meaning that EBNF adds more syntax for representing the grammar.

BNF & EBNF:

- Alternatives:
  $A ::= a \mid b \rightarrow A$ is either a or b.

EBNF:

- Optional:
  $A ::= a? \rightarrow a$ is repeated 0 or 1 time.
  $A ::= [\alpha] \rightarrow \alpha$ is repeated 0 or 1 time.
- Repetition:
  $A ::= a^* \rightarrow a$ is repeated 0 or more times.
  $A ::= a^+ \rightarrow a$ is repeated 1 or more times.
- Group:
  $A := (a \mid b)\alpha \rightarrow (a \mid b)$ is treated as a single element.

2.2 Grammar derivation

From the grammar of languages, we can get terminal strings. *Terminal strings* are strings that are valid according to the grammar. To get terminal string for a grammar, we start with the start symbol for the grammar and go through its production rules, replacing

Listing 2.1: Example of a grammar

```
Rule 1: Expr → Expr + Expr
Rule 2: Expr → NUM
Rule 3: Expr → CHAR
Rule 4: NUM  → \[0-9]\
Rule 5: CHAR → \[A-Za-z]\
```

Listing 2.2: Example of leftmost derivation for terminal string: 1 + a + 3

```
Expr
Expr + Expr            (Rule 1 on Expr)
NUM + Expr             (Rule 2 on first Expr)
1 + Expr               (Rule 4 on first NUM)
1 + Expr + Expr        (Rule 1 on Expr)
1 + CHAR + Expr        (Rule 3 on first Expr)
1 + a + Expr           (Rule 5 on CHAR)
1 + a + NUM            (Rule 2 on Expr)
1 + a + 3              (Rule 4 on NUM)
```

every nonterminal with one of its production rules until there are no nonterminals left. The steps taken to get from the start symbols production rule, to a terminal string is called *derivation*.

There are two types of derivation, *leftmost derivation* and *rightmost derivation*. With leftmost derivation, the "*leftmost*" nonterminal, the nonterminal farthest to the left, is always chosen to be replaced by one of its production rules. The rightmost derivation will always replace the nonterminal farthest to the right. In Listing 2.1, we have given a grammar with five rules. The start symbol here is `Expr`. A terminal string for `Expr` is `1 + a + 3`, a leftmost derivation for this string is given in Listing 2.2, and a rightmost derivation in Listing 2.3.

Listing 2.3: Example of rightmost derivation for terminal string: 1 + a + 3

```
Expr
Expr + Expr            (Rule 1 on Expr)
Expr + Expr + Expr     (Rule 1 on last Expr)
Expr + Expr + NUM      (Rule 2 on last Expr)
Expr + Expr + 3        (Rule 4 on NUM)
Expr + CHAR + 3        (Rule 3 on last Expr)
Expr + a + 3           (Rule 5 on CHAR)
NUM + a + 3            (Rule 2 on Expr)
1 + a + 3              (Rule 4 on NUM)
```

2.3 Lark

Lark is a parser generator for Python, that can parse arbitrary context-free grammars [15]. *Lark* also has its own context-free EBNF grammar that we will be using in the implementation of this algorithm.

The *Lark* grammar specification has two forms of nonterminal, *tokens* and *rules*/nonterminals. The main difference between these is that *tokens* are closer to terminal string then *rules*. This is because the tokens are a form of nonterminals where the production rule is a sequence of regular expressions, terminals, and/or tokens. *Tokens* are defined by capital letters, as shown in the example below:

```
NUMBER: /[0-9]/
NEG_NUMBER: "-" NUMBER
INT: NUMBER | NEG_NUMBER
COMMENT: "#" /[a-zA-Z]+/
```

The *rules*, or nonterminals, are the names of the production rules that are defined as sequences of terminals, nonterminals, and/or tokens, they have a longer path for getting to terminal string:

```
decimal: INT "." NUMBER
expr: expr "+" expr | expr "-" expr
    | expr "*" expr | expr "/" expr
    | decimal | INT
```

Chapter 3

The Algorithm to Generate Sample Strings

In this section, we will show our algorithm, *Sample Generation Algorithm*, and explain its different parts. While explaining the *Sample Generation Algorithm* we will also give an example of a grammar that we can take through the algorithm, showing how it works. An overview of the different parts of the algorithm can be seen in Table 3.1.

The grammar we are working with in this *Sample Generation Algorithm* is a context-free EBNF grammar. We will demonstrate various parts of the algorithm using a running example presented in Listing 3.1. The *Sample Generation Algorithm* only works on and modifies the production rules of the grammar; all other parts of the grammar specification stay the same.

,

Listing 3.1: An example grammar for the algorithm.

```
Σ : a, b, c, d, e
N : A, B, D
R :
    A → a B+ (c | D)* [e]
    B → b [B]
    D → d | d D
S : A
```

Algorithm	**Explanation**
The grammar (Algotithm **??**)	The representation of the grammar used in the algorithm.
Extract groups (Algorithm 1)	Loops over every constituent part of every production rule, looking for group instances, and extracts them into there own rule if found.
Splitting the grammar (Algotithm 2)	Given a grammar, without groups, splits the grammar into two parts, one whiteout cardinality modifiers and cycles, and one with.
Handling repetition (Algotithm 4)	Given a grammar, it loops over every constituent part of every production rule, looking for instances of repetition elements (Kleene star, Kleene plus, optional), and add the content of the element x number of times to the rule, where x is decided based on the element, replacing the repetition element that was there originally.
Detecting cycles (Algotithm 3)	Takes a nonterminal and a production rule to check whether the rule contains the given nonterminal; makes recursive calls in cases of nonterminals that are not the one we are looking for.
Inline nonterminals (Algotithm 5)	Given a production rule, for every nonterminal in that rule, replaces it with one of its production rules.
Contains nonterminal (Algotithm 6)	Returns true or false based on whether or not a production rule contains a nonterminal.
Generate terminal string (Algotithm 7)	Generates a finished terminal string for every nonterminal in a grammar.
Generate constituent part (Algotithm 8)	Generates and returns terminal string for the given constituent part.

Table 3.1: The different parts of the algorithm.

Listing 3.2: The grammar definition

Let G be a EBNF grammar.
$G = (\Sigma, N, R, S)$
$A \in N$
$\alpha, \beta \in (\Sigma \cup N)*$
ρ is a regular expression
$\gamma \in \{\alpha, \alpha^*, \alpha^+, \alpha \mid \beta, [\alpha], (\alpha), \rho\}$
$A \to \gamma_1 \dots \gamma_2$

Listing 3.3: Set of rules before and after extracting groups.

R :	R with extracted groups:
A → a B^+ (c | D)* [e]	A → a B^+ $A_3{}^*$ [e]
B → b [B]	B → b [B]
D → d | d D	D → d | d D
	A_3 → c | D

The first step of the algorithm is to extract groups, that is, a collection of grammar symbols that are grouped together, represented by parenthesis "(" ")", being treated as a single symbol. Algorithm 1 presents the implementation of this. The algorithm loops over every constituent part of a production rule for every production rule in the given grammar, looking for instances of groups. In the case when a group is detected, a new production rule is created and added to the set of rules in the grammar. The new production rule will have the same nonterminal-name as the nonterminal-name of the rule it was detected in, only with an added number. As an example, if the nonterminal of the production rule at hand is A, and the group is the second element of the rule, then the new nonterminal for that extracted group would be named A_2.

The group's content will be the right-hand side of the new production rule. The *extract_group* function is called recursively on the group's content, before the group content is added to the grammar. This recursive call is done in case there is a group inside of another group—in that case both groups should be extracted. The new nonterminal is added to the production rule we are looking at, replacing the group. The other cases are when an element of star, plus or optional element contains a group; then the same extraction is applied as with just a group, and the new nonterminal for the extracted group is replacing the group element inside of the star, plus or optional element. The result after this algorithm is a set of rules where every group has been replaced by a new nonterminal, containing the content of the group it replaced.

If we run our example grammar through Algorithm 1, "Extracting groups", we would get one new production rule and modify one of the existing rules. The group `(c | C)` in A will be extracted out into its own rule A_3. In Listing 3.3, the original set of rules can be seen to the left, and the set of rules with extracted groups to the right.

After groups have been extracted from the grammar, the next step is to divide the new set of rules into two new sets, where one of the sets will contain rules without constituent parts such as Kleene star, Kleene plus, and optional elements. The goal is to simplify the rules so we have a set we can choose from when the goal is to finish generating as

Algorithm 1 Extracting groups

Let G be a context-free EBNF grammar. Let $A \to \gamma_1 \dots \gamma_n$ be a production rule, where A is the nonterminal and $\gamma_1 \dots \gamma_n$ is the sequence of constituent part making up the rule.

```
1: G = (Σ, N, R, S)
2: A ∈ N
3: α ∈ (Σ ∪ N)*
4: β ∈ (Σ ∪ N)(Σ ∪ N)+
5: for A → γ1 ... γn do
6:     for i = 1 to n do
7:         switch γi do
8:             case (α)
9:                 R = R ∪ {Ai → γi}
10:                γi = Ai
11:            case (α)*
12:                R = R ∪ {Ai → γi}
13:                γi = Ai*
14:            case (α)+
15:                R = R ∪ {Ai → γi}
16:                γi = Ai+
17:            case [β]
18:                R = R ∪ {Ai → γi}
19:                γi = [Ai]
20:        end switch
21:    end for
22: end for
```

quickly as possible. Therefore, will this set also not contain rules with cycles, here we define cycles as a path from a nonterminals production rule back to that nonterminal. We will call the grammar holding this set *cycle-free* grammar. The other set will contain the rest of the rules: namely, the original rules before modifications, and every rule that contains one or more cycles. The grammar with this set of rules will be called *cyclical* grammar.

The process for this algorithm can be seen in Algorithm 2, "Splitting the grammar". The algorithm loops over every constituent part of every production rule in the grammar with extracted groups. A new rule is constructed for every production rule, with possible modifications. Then, for every constituent part of a rule we have three cases. The first case is for optional and star elements; here nothing is added to the new rule. This is because the minimum number of repetitions for both of these elements is zero. In the algorithm, the empty string (designated by ϵ) represents nothing being added. The second case is for star elements: star elements are repeated one or more times, so one is the minimum number of repetitions for star elements. This means that we add the content of the star element to the new rule. The last case is for anything not covered by the previous two cases, just adding the constituent part as it is to the new production rule. For example, for a rule "$X \rightarrow b^* \ c^+ \ [d] \ e$", a new rule "$X \rightarrow c \ e$" will be constructed.

When a new production rule is completed, it will be compared to the original rule. If the rules are not the same, meaning that the new rule is a modification of the original rule. Then the original rule is added to the set of rules for the cyclical grammar. We then check the new production rule for cycles. This is done by the function *has_cycle*, presented in Algorithm 3. If the new rule has a cycle, it is added to the rule set in the cyclical grammar. Otherwise, if it does not have any cycles, it is added to the rule set of the cycle-free grammar.

The *has_cycle* function is shown in Algorithm 3, "Detecting cycles". This function takes a production rule and a nonterminal as parameters. The production rule is the sequence of constituent parts that the function will loop over, looking for the given nonterminal. While looping over the constituent parts, an action is performed if it is a nonterminal, or an element containing a nonterminal, like a Kleene star, Kleene plus, or an optional element. In these cases, we look at that nonterminal, if it is the nonterminal we were given, then the function will return *true*. If its not the nonterminal we are looking for, we check if it is in the set of visited nonterminals, a set of nonterminal we have already checked, none of the production rules for the nonterminals in this set have a reference to

Algorithm 2 Splitting the grammar

Let G be the grammar with extracted groups.

1: $G = (\Sigma, N, R, S)$
2: $G_{\otimes} = (\Sigma, N, R_{\otimes}, S)$
3: $G_{\circlearrowleft} = (\Sigma, N, R_c, S)$
4: $G = G_{\otimes} \cup G_{\circlearrowleft}$
5: $A \in N$
6: $\alpha \in \Sigma \cup N$
7: **for** $A \to \gamma_1 \dots \gamma_n \in R$ **do**
8: **for** $i = 1$ to n **do**
9: **switch** γ_i **do**
10: **case** $[\alpha], \alpha^*$
11: $\gamma_i^f = \varepsilon$
12: **case** α^+
13: $\gamma_i^f = \alpha$
14: **case** α, ρ
15: $\gamma_i^f = \gamma_i$
16: **end switch**
17: **end for**
18: **if** $A \to \gamma_1 \dots \gamma_n \neq A \to \gamma_1^f \dots \gamma_n^f$ **then**
19: $R_{\circlearrowleft} = R_{\circlearrowleft} \cup \{A \to \gamma_1 \dots \gamma_n\}$
20: **end if**
21: **if** has_cycle($A \to \gamma_1^f \dots \gamma_n^f$, A) **then**
22: $R_{\circlearrowleft} = R_{\circlearrowleft} \cup \{A \to \gamma_1^f \dots \gamma_n^f\}$
23: **else**
24: $R_{\otimes} = R_{\otimes} \cup \{A \to \gamma_1^f \dots \gamma_n^f\}$
25: **end if**
26: **end for**

Listing 3.4: Set of rules with simplified duplicates.

```
R with extracted groups :        Newly built rules :
    A → a B+ A3*  [e]                A → a B
    B → b [B]                        B → b
    D → d
    D → d D
    A3 → c
    A3 → D
```

the nonterminal we are looking for. If it is not in this set, we add it to the set of visited nonterminals, so we do not end up in a loop, checking the same nonterminals over again. Then we call *has_cycle* recursively for every production rule of that nonterminal, with the nonterminal we are looking for. Returning *true* if one of them have a reference to the original nonterminal. If we have not returned by the end of the loop, no path back to the given nonterminal was found, so the function returns *false*.

Algorithm 3 Detecting cycles

1: $G = (\Sigma, N, R, S)$
2: $A, B \in N$
3: let $\hat{N} = \phi$
4: **function** HAS_CYCLE($A \to \gamma_1 \dots \gamma_n$, A)
5: **for** $i = 1$ to n **do**
6: **if** $\gamma_i = B$ or $\gamma_i = B^*$ or $\gamma_i = B^+$ or $\gamma_i = [B]$ **then**
7: **if** $B == A$ **then**
8: **return** true
9: **else if** $B \notin \hat{N}$ **then**
10: $\hat{N} = \hat{N} \cup \{B\}$
11: **for** $B \to \omega_1 \dots \omega_m \in R$ **do**
12: **if** has_cycle($B \to \omega_1 \dots \omega_m$, A) **then**
13: **return** true
14: **end if**
15: **end for**
16: **end if**
17: **end if**
18: **end for**
19: **return** false
20: **end function**

Say we are starting with the set of rules we got from extracting the groups of the sample grammar, we build simplified duplicates for some of the rules in the first part of the algorithm, see Listing 3.4. The next step of the algorithm is to add all the original rules, that have a new modification, to the cyclical grammar. For the example grammar,

Listing 3.5: Rule with and without cycle.

```
Rule containing a cycle:        Rule not containing a cycle:
    D → d D                         A_3 → D
                                    D → d
                                    D → d D
```

Listing 3.6: The two sets of rules after splitting.

```
Cyclical grammar:               Cyclefree grammar:
    A → a B+ A_3* [e]               D → d
    B → b [B]                       A_3 → c
    D → d D                         A_3 → D
                                    A → a B
                                    B → b
```

that is the rules for A and B. Then we have to check the remaining rules for cycles, an example of a rule containing a cycle and one not containing a cycle is done in Listing 3.5.

In our example grammar, D is the only remaining rule that contains a cycle, after adding the original rules for A and B to the cyclical grammar. In Listing 3.6 the rule list for the cyclical grammar is to the left, and the rules of the cycle-free grammar to the right.

Before the grammar with extracted groups, the cycle-free grammar, and the cyclical grammar can be used to inline nonterminals, repetition elements must be dealt with. Repetition elements are *Kleene Star*, *Plus* and *Optional*. In Algorithm 4, "Handle repetition", we loop over every production rule in the rule sets of these three grammars, the grammar with extracted groups, the cycle-free grammar, and the cyclical grammar, calling the function *handle_repeater* on the *Sequence* making up the rule.

The *handle_repeater* function loops over every constituent part of the given rule, looking for repetition elements. The function has four cases for the constituent part it is processing, all of which make a recursive call on the content of the constituent part. Three for the repetition element, and the last and fourth case is for a *Sequence* element. For this case, the *Sequence*, the constituent part is updated with the return of the recursive call on the content of the *Sequence*. In two of the other three cases, the repetition cases, the constituent part is updated with a *Sequence* of the output from the recursive call. For a *Plus* element this is a *Sequence* of one or more repetition of the output, and for a *Star* element this is a *Sequence* of zero or more repetition of the

Listing 3.7: Grammar with extracted groups before and after handling repetition

R with extracted groups:	R with handled repetition:
A → a B^+ A_3^* [e]	A → a B B B A_3 A_3 e
B → b [B]	B → b B
D → d	D → d
D → d D	D → d D
A_3 → c	A_3 → c
A_3 → D	A_3 → D

Listing 3.8: Cyclical grammar before and after handling repetition

Cyclical grammar:	Cyclical grammar(handled repetition):
A → a B^+ A_3^* [e]	A → a B B B A_3 A_3 e
B → b [B]	B → b B
D → d D	D → d D

output. The last case is for a *Optional* element. Here the constituent part is updated to the output of the recursive call or the empty string, ϵ. In Listings 3.7 and 3.8, the rule sets for extracted groups and cyclical grammar can be seen before and after going through this algorithm for handling repetition elements.

The next step of our algorithm is to inline nonterminals for all the production rules in the grammar, with extracted groups (Algorithm 1) and handled repetition elements (Algorithm 4). To do this, we use the two new grammars we got from splitting the grammar (Algorithm 2). We create a function *inline_nonterminals* that takes a production rule as a parameter and a number d for how deep the function should go. The function returns the production rule with all of its nonterminals inlined. This function can be seen in Algorithm 5, "Inlining the grammar". The number d is used to represent how big/long the generated terminal string should be, if we should fill the empty set $\widetilde{R}$ with the rules from the cycle-free grammar or the cyclical grammar.

In this function, we define a variable h, h is the number d divided by two. d is the depth we are given, and h is halfway. The function *contains_nonterminal* (Algorithm 6) is used in the while-loop of the *inline_nonterminals* function. This is to keep the function going until every nonterminal has been inlined, replaced by one of its production rules. Inside this while-loop, we loop over every constituent part of the rule, if the constituent part we are processing is a nonterminal, or an element containing a nonterminal, a *sequence*, we have to update the rule set $\widetilde{R}$.

If the variable d is greater than h, we are early in the inline process, and the set $\widetilde{R}$ will be updated with the rule set from the cyclical grammar. If d is less than h but greater

Algorithm 4 Handle repetition

Let G be the grammar with extracted groups. Let $G_{\otimes}$ be the cycle-free grammar. Let $G_{\circlearrowleft}$ be the cyclical grammar.

$G = (\Sigma, N, R, S)$
$G_{\otimes} = (\Sigma, N, R_{\otimes}, S)$
$G_{\circlearrowleft} = (\Sigma, N, R_{\circlearrowleft}, S)$
$A \in N$
$\alpha \in (\Sigma \cup N)^*$
for $A \to \gamma_1 \dots \gamma_n \in R$ **do**
    $A \to$ handle_repeater$(\gamma_1 \dots \gamma_n)$
end for
for $A \to \gamma_1 \dots \gamma_n \in R_{\otimes}$ **do**
    $A \to$ handle_repeater$(\gamma_1 \dots \gamma_n)$
end for
for $A \to \gamma_1 \dots \gamma_n \in R_{\circlearrowleft}$ **do**
    $A \to$ handle_repeater$(\gamma_1 \dots \gamma_n)$
end for

function HANDLE_REPEATER$(\gamma_1 \dots \gamma_n)$
    for $i = 1$ to n **do**
        switch γ_i **do**
            case α^+
                $\lambda =$ handle_repeater(α)
                $\gamma_i = \lambda^k$, with $k =$ random_int$(1, \infty)$
            case α^*
                $\lambda =$ handle_repeater(α)
                $\gamma_i = \lambda^k$, with $k =$ random_int$(0, \infty)$
            case $[\alpha]$
                $\lambda =$ handle_repeater(α)
                $\gamma_i =$ choose_random(λ, ϵ)
            case α
                $\lambda =$ handle_repeater(α)
                $\gamma_i = \lambda$
        end switch
    end for
    return $\gamma_1 \dots \gamma_n$
end function

than zero $\widetilde{R}$ is updated to contain the rules from both sets, from both the cyclical and cycle-free grammars. This is because we are no longer in the beginning phase of inlining, nor in the end, so it does not have a lot to say if we take the fastest path to finish inlining or not. When d is less than zero, we have to take the fastest path to finish inlining, so in this case we update $\widetilde{R}$ with the rule set of the cycle-free grammar.

After $\widetilde{R}$ have been updated with the correct set(s) we have to retrieve all the production rules for the nonterminals in $\widetilde{R}$, the set of these production rules are saved in $\widetilde{R_B}$, and decrease d by one. We can then choose randomly from the set $\widetilde{R_B}$ which rule that will replace the nonterminal. There are three possibilities of a nonterminal occurrence, and the function has three cases for them. Making sure that the production rule that is replacing the nonterminal is placed correctly in the rule, directly in the rule or inside of a sequence. When every nonterminal has been replaced by one of its production rule, the new production rules without nonterminals is returned.

Algorithm 6, "Production rule containing nonterminal", has the *contains_nonterminal* function, that is used by Algorithm 5, "Inlining the grammar", in the while loop. this function takes in a production rule and returns *true* or *false* based on whether or not it contains a nonterminal.

To find out if a production rule contains a nonterminal we loop over every constituent part of the rule. There are two cases to think about, the first is if it is a nonterminal, then we return *true*. The second is if it is a *sequence* element, an element containing something else then we return the result of a recursive call on that content. If the constituent part is neither, then there is nothing to do. If we loop over every constituent part without detecting a nonterminal, the function returns *false*.

In Listing 3.9, an example is shown for a rule going through this process. It shows the rule $A \rightarrow a\ B^{+}\ A_3^{*}\ [e]$, with a depth d of five, as it is being transformed into a rule not containing nonterminals. In Listing 3.10, a complete list of the rules for the example grammar with extracted groups is to the left, with their inlined version to the right. This is just one result of the inlining step on these rules (Algorithm 5), but there are many choices underway, so there exist multiple alternatives for the result of this algorithm.

The final step of the algorithm is to do the actual generating. This is done in Algorithm 7, "Generate terminal string". Before starting the actual generating, we have to call the *inline_nonterminals* function (Algorithm 5), on every production rule in the grammar with extracted groups and handled repetition elements, saving the returned rules in a new set, R'. We can then loop over this set of rules, generating terminal strings

Algorithm 5 Inlining the grammar

Let $G_{\otimes}$ be the cycle-free grammar, and $G_{\circlearrowleft}$ be the cyclical grammar we got from splitting the grammar in Algorithm 2, and after handling repetition elements in Algorithm 4. Let $\widetilde{R}$ be an empty set that we can fill with rules from $G_{\otimes}$ and/or $g_{\circlearrowleft}$.

$G_{\otimes} = (\Sigma, N, R_{\otimes}, S)$
$G_{\circlearrowleft} = (\Sigma, N, R_{\circlearrowleft}, S)$
$\widetilde{R} = \emptyset$
$A, B \in N$
function INLINE_NONTERMINALS($A \to \gamma_1 \dots \gamma_n$, d)
    let $h = d/2$
    while contains_nonterminal($A \to \gamma_1 \dots \gamma_n$) **do**
        for $i = 1$ to n **do**
            if $\gamma_i = B$ or $\gamma_i = \alpha B \beta$ **then**

                if $d > h$ **then**
                    $\widetilde{R} = R_{\circlearrowleft}$
                else if $d > 0$ **then**
                    $\widetilde{R} = R_{\otimes} \cup R_{\circlearrowleft}$
                else
                    $\widetilde{R} = R_{\otimes}$
                end if

                $\widetilde{R_B} = \{\omega_1 \dots \omega_n \mid B \to \omega_1 \dots \omega_n \in \widetilde{R}\}$
                $d = d - 1$

                switch γ_i **do**
                    case B
                        $\gamma_i =$ choose_random_uniformly($\widetilde{R_B}$)
                    case $\alpha B \beta$
                        $\gamma_i = \alpha$ choose_random_uniformly($\widetilde{R_B}$) β
                end switch
            end if
        end for
    end while
    return $A \to \gamma_1 \dots \gamma_n$
end function

Algorithm 6 Production rule containing nonterminal

```
1: A, B ∈ N
2: α ∈ Σ ∪ N
3: function CONTAINS_NONTERMINAL(A → γ_1 ... γ_n)
4:     for i = 1 to n do
5:         switch γ_i do
6:             case B
7:                 return true
8:             case α
9:                 return contains_nonterminal(α)
10:        end switch
11:    end for
12:    return false
13: end function
```

Listing 3.9: Example of inlining

```
A → a B B B A3 A3 e          d=5          Cyclical grammar:
A → a bB B B A3 A3 e         d=4              A → a B B B A3 A3 e
A → a bB bB B A3 A3 e        d=3              B → b B
A → a bB bB b A3 A3 e        d=2              D → d D
A → a bB bB b D A3 e         d=1
A → a bB bB b D c e          d=0          Cyclefree grammar:
A → a bb bB b D c e          d=-1             A → a B
A → a bb bb b D c e          d=-2             A3 → c
A → a bb bb b d c e          d=-3             A3 → D
                                              B → b
    A → a bb bb b d c e                       D → d
```

Listing 3.10: Inlined rule set

```
R:                                  R with inlined nonterminals:
    A → a B B B A3 A3 e                 A → a bb bb b d c e
    B → b B                             B → b b b b b b
    D → d                               D → d
    D → d D                             D → d d d d d
    A3 → c                              A3 → c
    A3 → D                              A3 → d d d d d d
```

for them. This is done by looping over the production rule's constituent parts, concatenating the results from calling the *generate* function (Algorithm 8) on all of them, and then printing the result.

Algorithm 7 Generate terminal string

Let G be grammar after extracting groups and repetition. Let G' be the grammar we end up with after inlining the rules in R.

1: $G = (\Sigma, N, R, S)$
2: $G' = (\Sigma', N', R', S')$
3: **for** $A \to \gamma_1 \dots \gamma_n \in R$ **do**
4: $\quad R' = R' \cup$ inline_nonterminals$(A \to \gamma_1 \dots \gamma_n, d)$
5: **end for**
6: **for** $A \to \gamma_1 \dots \gamma_n \in R'$ **do**
7: $\quad$ let $t =$ ""
8: $\quad$ **for** $i = 1$ to n **do**
9: $\quad\quad t = t +$ generate(γ_i)
10: $\quad$ **end for**
11: $\quad$ **print**(t)
12: **end for**

The *generate* function is shown in Algorithm 8, "Generate constituent part". It takes a constituent part as a parameter and returns a generated terminal string for that part. There are three cases for the constituent part, sequence, terminal and regular expression. In the case of a sequence, we do the same thing we do for a production rule in Algorithm 7, loop over the different parts of the sequence calling the *generate* function recursively on every part, and then concatenating the results into one terminal string that is returned by the function. Terminals and regular expressions are the base cases of this *generate* function, this means that they do not make recursive calls but return output directly. For a terminal string, it is returned as is, because it already is a terminal string. For regular expression, we use an existing function to generate a string from the expression, more on this in the next section, "Implementation" 4.

In Listing 3.11, an example of generated terminal strings is given. There is not much of a difference between the left and right-hand sides, but to the right, before generating, we have *sequence* elements. The entire rule is one sequence, and different parts inside of the rule can also be sequences. For the nonterminal `A` the different `bb` are different sequences, so is `b`. There could have been more `b`'s in the different sequences of the rule, but the reason they do not have it in this example is because of the variable d, in the inline nonterminals Algorithm 5. The variable d is used to decide which rule is to replace a nonterminal, and with the value for d we had in this example, this is what we got.

Algorithm 8 Generate constituent part

```
1: δ is a constituent part.
2: a ∈ Σ
3: ρ is a regular expression
4: function GENERATE(γ)
5:     switch γ do
6:         case δ
7:             let s = ""
8:             for a in δ do
9:                 s = s + generate(a)
10:            end for
11:            return s
12:        case a
13:            return a
14:        case ρ
15:            return RegExp(ρ)
16:    end switch
17: end function
```

Listing 3.11: Example of generated terminal string.

```
R with inlined nonterminals:      Generated terminal text:
    A → a bb bb b d c e               A : abbbbbdce
    B → b b b b b b                   B : bbbbbb
    D → d                             D : d
    D → d d d d d                     D : ddddd
    A3 → c                            A3: c
    A3 → d d d d d d                  A3: dddddd
```

Chapter 4

Implementation & Case Study

4.1 Implementation

In the previous section, we explained the algorithm, *Sample Generation Algorithm*, Furthermore, we showed an example grammar being taken through the algorithm. Now we have implemented this algorithm in Python, and will in this section look at how we implemented the different parts, making up the complete algorithm.

Our implementation[1] of the *Sample Generation Algorithm* takes a grammar specification in the format of a parser generator *Lark* [15], and outputs a set of strings which can be derived by each of the Nonterminal and Token[2] symbols in the grammar.

To parse grammar specifications made in Lark and construct their corresponding parse trees, we use a Lark grammar in the format of Lark [16]. This way, we can use the parsing facilities from the library to create a parser for Lark grammars. The library provides functionality to traverse the parse trees of the grammar specifications, and to build various representations of Lark grammars that are more convenient to work with in certain contexts. In our case, the data structure of choice is a dictionary, where the keys are the *Nonterminal* or *Token* symbols, and the values are lists of grammar rules for a corresponding *Nonterminal* (or *Token*).

Different constituent parts of a Lark-formatted EBNF grammar, such as nonterminals, groups of terminals and nonterminals, cardinality modifiers, thereof such as Kleene star modifier, Kleene plus modifier, optionality modifier, and others, are represented as Python classes, a complete list of these classes can be seen in Table 4.1.

[1] `https://github.com/mhhundvin/Master_Project`

[2] The difference between *nonterminal symbols* and *tokens* in Lark is described in Section 2.3.

Listing 4.1: An example of a simple Lark grammar.

```
start: list | dict | STRING | NUMBER | "null"
list: "[" [start ("," start)*] "]"
dict: "{" [pair ("," pair)*] "}"
pair : STRING ":" start
STRING: /[a-zA-Z0-9 ]*/
NUMBER: /[0-9]+/
```

All of these classes encapsulate common behaviour expressed in the following methods:

- `to_string` — to get the original grammar representation;
- `get_arg` — to get the arguments of the class, what it contains;
- `generate` — to generate terminal string for the arguments;
- `contains_cycle` — to check if the arguments contains a cycle.

The *generate* function is only implemented in six of the eleven classes. This is because we will have removed instances of Group and repetition elements (Plus, Star, Optional, Repeat) before we get to the generating step.

Instances of these classes are created when a parse tree of a Lark grammar is traversed: the dictionary corresponding to a grammar is populated with the respective instances of the classes following the structure of the constituent parts in a grammar. In Listing 4.1, an example grammar in Lark is given, with its corresponding parse tree in Listing 4.2, and the dictionary created by traversing the tree is given in Listing 4.3.

After the dictionary corresponding to a given grammar has been created, the next step is to simplify the grammar. This is done by the extract groups Algorithm 1, which performs group extraction [17, p. 182] from the rules. The algorithm loops through every production rule of every *Nonterminal* (and *Token*) and looks for grouping expressions. In the implementation, this comes down to having a function that for every key–value pair in the dictionary loops through the value list (i.e., a list of productions) and calls `extract_group` function on every production rule. The `extract_group` function, in its turn, loops over the elements in a production rule, and rebuilds that rule with groups extracted, and added to the grammar dictionary as new rules.

The next step of the implementation is splitting the grammar, as specified in Algorithm 2. Our implementation uses two more dictionaries, one for the cycle-free grammar, and one for the cyclical grammar.

Class	Explanation
`Generatable`	An interface for the rest of the classes. They implement its functions, and makes it easier to check if an element is one of our implemented classes, have the given functions.
`Literal_Range`	The arguments of this element is a start and end value for a list of elements, generating for this element is a random value for the given range. A: ”a” .. ”z” | ”0” .. ”9”
`Repeat`	The argument of this element is repeated as many times as specified, between two numbers. A: elem∼1 .. 3
`Optional`	The argument of this element is either generated or not. A: [elem] | elem?
`Star`	The argument of this element is repeated zero or more times. A: elem*
`Plus`	The argument of this element is repeated one or more times. A: elem+
`Group`	The arguments of this element is grouped together, into one argument/element. A: (elem elem2)
`Sequence`	A sequence of arguments, that, unlike a group, is not treated as one argument.
`Regexp`	The argument of this element is a regular expression, anything that is valid according to the expression is a valid terminal string for this element.
`Nonterminal`	The argument of this element is a lower-cased ”name” for a production rule.
`Token`	The argument of this element is a capitalized ”name” for a production rule.
`Terminal`	The argument of this element is a string, this is the base structure of a grammar.

Table 4.1: The different python classes that we implemented.

Listing 4.2: Parse tree for the example grammar.

```
rule
start
expansions
    name    list
    name    STRING
    name    NUMBER
    literal "null"

rule
list
  expansion
    literal "["
    maybe
        expansion
          name      start
          opexper
            group
                expansion
                  literal   ","
                  name      start
            *
    literal "]"

token
STRING
  literal   /[a-zA-Z0-9 ]*/

token
NUMBER
  literal   /[0-9]+/
```

Listing 4.3: The dictionary created from the example grammar.

```
{
Nonterminal(start) : [Nonterminal(list),
                      Token(STRING),
                      Token(NUMBER),
                      Terminal(null) ],
Nonterminal(list) : [Sequence(Terminal([),
                        Optional(Nonterminal(start),
                          ↪ Star(Group(Terminal(,),
                          ↪ Nonterminal(start)))),
                        Terminal(]))],
Token(STRING) : [Regexp([a-zA-Z0-9 ]*)],
Token(NUMBER) : [Regexp([0-9]+)]
}
```

In the implementation, we have a function that loops over every constituent part of every rule in the grammar, and builds a new rule for every production rule, a list *new_alternative*. If we come across an optional element, *Star* or *Optional*, we have a Boolean variable, *multiple_options*, that is updated to *true*, this variable is set to *false* for every new rule we iterate over. If the constituent part is a Plus, the *multiple_options* variable is set to *true*, and the content of the Plus element is added to the list *new_alternative*. For every other case, the constituent part is added as it is to the *new_alternative* list.

After looping through every constituent part of a rule, and building the list *new_alternative*, we have to add them, the new rule and the original rule, to the fitting grammar dictionary. If the *multiple_options* is true, then we add the original rule to the cyclical grammar dictionary, and check the new rule, *new_alternative*, for cycles, after making it a *Sequence*. We check the new rule for cycles using the *contains_cycle* function every class has. If the new rule does not have any cycles, it is added to the cycle-free grammar dictionary, otherwise, it too is added to the cyclical grammar dictionary. If *multiple_options* is false, we only care about the original rule, checking it for cycles and adding it to the correct grammar dictionary based on the result of calling the *contains_cycle* function on the rule.

The *contains_cycle* function works by calling it on every element of a class except for the classes Terminal and Regular expression, these classes returns false because they can not contain cycles. The *contains_cycle* function for the *Nonterminal* class is the only one that can return true. It checks if it is the Nonterminal we are looking for (started with), and if it is, the function returns *true*. If its not the one we are looking for, we check if we

have looked at that Nonterminal before, by seeing if it is in a list of visited nonterminals. If we have not looked at it before, we add the Nonterminal to the visited nonterminals list, and check every production rule for that nonterminal, seeing if they have a path back to the original Nonterminal.

After we have created the two new dictionaries for the cycle-free and cyclical grammar, we deal with repetitions (cf. Alg. 4). The implementation for this is a function that loops over every key–value pair of a grammar dictionary. This function is called for the three grammar dictionaries we have, the extracted groups, cycle-free and cyclical grammar dictionaries. For every production rule in the value list for a *Nonterminal* (or *Token*), another function is called, *handle_repeater*, that takes the production rule as a parameter.

The *handle_repeater* function loops over every constituent part of the production rule, building a new rule to replace it in the grammar, a list *new_alternative*, acting if it comes across a repetition element. Repetition elements are instances of the *Plus*, *Star*, *Optional*, and *Repeat* classes. The function makes a recursive call on the content of the constituent part for all of these cases, and saves the returned output in a variable. The function then creates a *Sequence* of the output before adding that *Sequence* to the new rule, *new_alternative*. The number of times the output from the recursive call is repeated in the *Sequence* depends on what element the constituent part was. For *Plus* elements, the output is repeated a random number of times between one and 10. It is almost the same for *Star*, but the random number is between zero and 10, and it is only added if another random number, between zero and nine, is an even number. In the case that the constituent part is an *Optional* element, the output is only repeated once, and only if a random number between zero and nine is even, same as with *Star* elements. The final repetition element, the *Repeat* element, have a start and an end field, as well as the arguments. In this case, the arguments of the *Repeat* element are repeated a random number of times between the start and end number of the *Repeat* element.

The function also acts for instances of *Sequence* elements. In this case a recursive call is made on the content of the *Sequence*, and the returned output is added to the new rule, the *new_alternative* list, as a *Sequence*. In all other cases, the constituent part is added to the *new_alternative* list as they are. After the function is done iterating over the rule, the *new_alternative* list is made an instance of a *Sequence* before it is returned. The returned *Sequence* replaces the original production rule in the grammar dictionary.

The next step after dealing with repetition is to inline *Nonterminals* (and *Tokens*) (cf. Alg. 6). For this step, the two new grammar dictionaries returned by splitting the

grammar, with handled repetition, will be used to inline the *Nonterminals* (and *Tokens*). We have the function, *inline_nonterminals*, that takes in these two dictionaries (cycle-free grammar dictionary and cyclical grammar dictionary), a production rule, and a depth. We loop over every element of the production rule replacing *Nonterminals* (and *Tokens*) with a random one of their rules, inlining them [17, p. 181], and calling the function recursively in cases of *Sequence*. In the case of a *Nonterminal* or *Token*, another function is called to decide from which grammar we should choose the production rule to replace the *Nonterminal*/*Token* with. If the depth is greater than the start depth divided by two, we choose the replacement from the cyclical grammar. If it is below this, but still greater than zero, we choose randomly from both the cycle-free and cyclical grammars. If the depth is zero or below, we only choose from the cycle-free grammar. In cases where the Nonterminal (or Token) has no rule in the preferred grammar, the other grammar is chosen. For every Nonterminal we replace, we decrease the depth by one. When we know which grammar to look at, we choose one of the production rules for the Nonterminal randomly. The loop that loops over the constituent parts of the rules is inside of a while loop, this while loop will run until there are no nonterminals left, meaning that we will keep iterating over the rule replacing *Nonterminals* and *Tokens* until there are none left in the rule. To check for *Nonterminals* (and *Tokens*) in the rule we have a function *contains_nonterminal*(cf. Alg. 6). This function loops over the constituent parts of the rule, it returns true if one of the constituent parts is a *Nonterminal* or *Token*, and it is called recursively in the case of a *Sequence* element, to check if the *Sequence* contains a *Nonterminal* (or *Token*). If no *Nonterminal* (or *Token*) is detected it returns false. After every *Nonterminal* (and *Token*) have been replaced the new production rule is returned.

In the implementation, we have a *Parser* class that creates the *Lark* parser, and parses the grammar giving the parse tree. Another class, a *Compiler* class, imports this tree and traverse it, building the grammar dictionary. After the grammar dictionary is created, the extract groups function is called, updating the grammar dictionary. The next step in the implementation is to call a *generate* function. This *generate* function starts by splitting the grammar, using the splitting function described earlier. Then it calls the function for handling repetition for the grammar with extracted groups, the cycle-free grammar, and the cyclical grammar dictionaries, updating their production rules. This *generate* function now have three grammar dictionaries without groups and repetition elements. The next step is then to loop over the grammar with extracted groups calling the *inline_nonterminals* function on every production rule for every *Nonterminal* (and *Token*), using the cycle-free grammar dictionary and the cyclical grammar dictionary. This function, the *generate* function, takes in the grammar with extracted groups, and the users wanted depth. A new dictionary is built by the new production rules returned

by the *inline_nonterminals* function calls. The final step is then to loop over this new dictionary, generating terminal string for every production rule of every *Nonterminal* (and *Token*). To do this, we loop over the constituent parts of the rules, calling their *generate* function, and concatenating the returned terminal strings.

The *generate* function works similarly to the *contains_cycle* function. Where the *generate* function is called for every element of an object's argument, concatenating the results before returning it. The function calls go all the way "down" until a terminal or a regular expression is reached. The argument of *Terminals* are strings, so their value is returned directly as terminal strings. The *generate* function for regular expressions is more complicated.

The *generate* function for *RegExp* elements uses a library to try and match the expression against a string. The strings we try to match the regular expression with are decided by using the name of the rule. We have one list for comments and one for identifiers. If the name corresponds to something that makes us believe it is a comment, we try to match the expression against the list of comments. We do the same with the list of identifiers if the name corresponds to something that makes us believe it is an identifier (such as "var", "identifier", etc.). In other cases, we have a text file we try to match the regular expression against. This text file contains different types of text, a story, binary numbers, and other types of numbers. If the regular expression is not matched against any of the mentioned methods, we use the library to generate a random string from the regular expression. This is the last option because the goal is to generate somewhat of a sensible result.

4.2 Case Study

In this section, we will give two examples of grammars, and their results for the generated terminal string. The two grammars we have chosen are a *test grammar* we created and the grammar for *Hedy* level 1. Hedy is a gradual language. On their website, kids and other learners, can learn programming gradually with this language [6]. It consists of different levels, where different constructs of programming are added for each new level. This way, the learner can focus their learning on one thing at a time.

In Figure 4.1, a section of the level 1 Lark grammar for Hedy is shown, the full grammar can be seen in Appendix A [5]. This section is the main part of the grammar, with

the start symbol, `start`, and what it builds. Using the implementation of the algorithm, *Sample Generation Algorithm*, we generate terminal strings for every nonterminal of the grammar. Here we have chosen to include the terminal string for the nonterminals `start` and `program`, this can be seen in Figure 4.2 for the `start` nonterminal, and in Figure 4.3 for the `program` nonterminal. As can be seen in the grammar for Hedy 4.1, the production rule for the nonterminal `start` is the nonterminal `program`. This means that anything generated for `program`, can also be generated for `start`. So the two examples given can be generated from the grammar's start symbol, `start`.

```
start: program
program: _EOL* (command _EOL+)* command?
command: print | ask | echo | turtle | error_invalid_space | error_invalid

print: _PRINT (text)?
ask: _ASK (text)?
echo: _ECHO (text)?
turtle: _FORWARD ((NUMBER | text))? -> forward
      | _TURN ((left | right ))? -> turn
      | _COLOR ((black | blue | brown | gray | green | orange
                | pink | purple | red | white | yellow | text))? -> color
error_invalid_space: _SPACE any
error_invalid: textwithoutspaces text?

any: /.+/ -> text

COMMENT: _HASH /([^\n]+)/
%ignore COMMENT

_EOL: "\r"?"\n"
```

Figure 4.1: Lark grammar for Hedy level 1

The generated output for `start` is shown in Figure 4.2, it is a `program`, and in the generated terminal string, we can see that it started with two `_EOL`, followed by two of the group `(command _EOL+)` and finishes off with a `command`. We can see this because it starts with three line breaks, followed by three `commands`. the first line break is not part of the generation, it is added when printing the generated terminal string, hence two line breaks are generated from `_EOL*`. Since there is no line break after the last `command`, it can not be from the group because the group ends with at least one line break. The two first `commands` are `turtle`, as can be seen by the "*forward*" keyword. The last `command` is `echo`, as seen by the "*echo*" keyword.

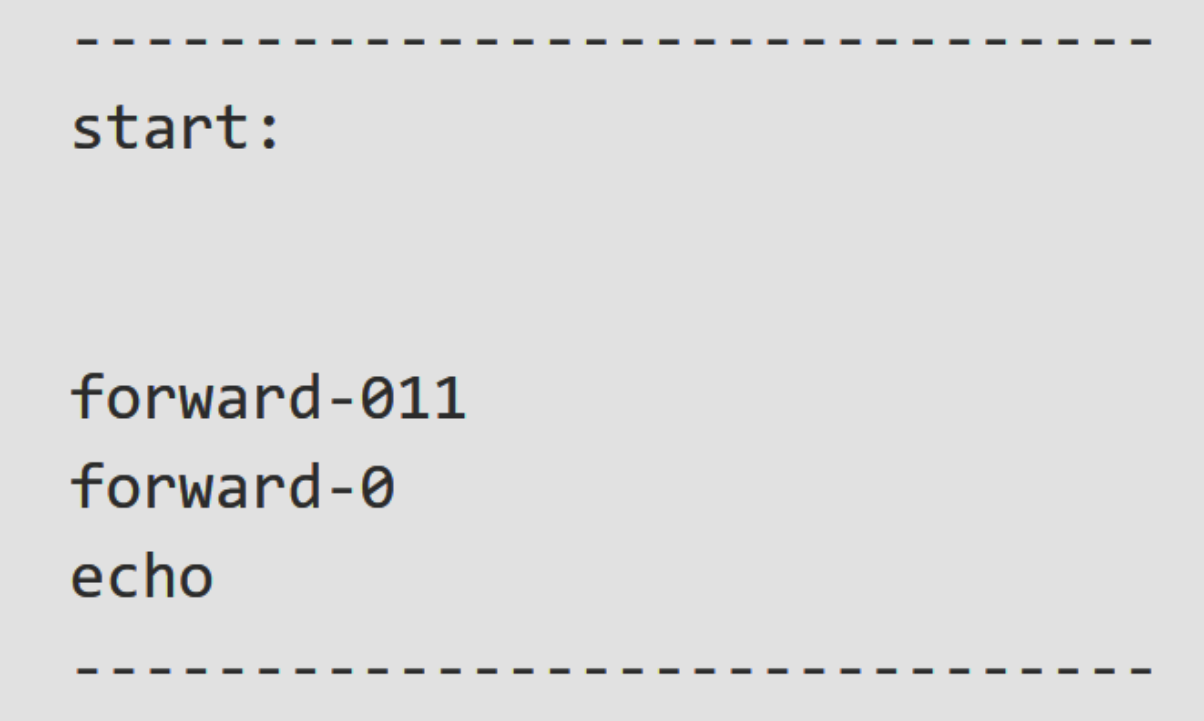

Figure 4.2: Generated terminal string for nonterminal `start` for Hedy level 1

For the nonterminal `program`, the generated terminal string is shown in Figure 4.3. It does not include the optional element `command` (at the end of the rule), nor the star element `_EOL`, star elements are repeated zero or more times, here it is repeated zero times. But it does contain 16 repetition of the star element `(command _EOL+)`. The first three generated output for the group `(command _EOL+)`, have multiple repetition of `_EOL`, while the rest only have one repetition, which is the minimum number of repetitions possible for plus elements. Here three different options for `command` is used, ask, `echo` and turtle, as can be seen by the keywords: "*ask*", "*echo*", "*forward*" and "*turn*". The keywords "*forward*" and "*turn*" are two different options for `turtle`.

Same as with the *Hedy* grammar, a section of the main part of our *test grammar* is given in Figure 4.4, and the full grammar is given in Appendix B. In Figures 4.5, 4.6 and 4.7 examples of generated terminal strings are given for the nonterminals: `start`, `variable_decl` and `statement`. From both the grammar and the generated output one can see that the terminal string for this test grammar has spaces, this is because of the *token* `SPACE`. Our algorithm does not fill in spaces, so it is up to the user to add them to the grammar if they are needed.

Figure 4.5 has the generated output for the nonterminal *start*, there are four different generated results, one for each production rule alternative. The first terminal string is

```
------------------------------
program:

ask3


echoa


ask0



echo0
echo1
forwardB
askd
turnleft
echo9
turnright
echo6
print1
printh
echo9
print9
ask7

------------------------------
```

Figure 4.3: Generated terminal string for nonterminal `program` for Hedy level 1

```
start: program
     | expr
     | variable_decl
     | statement

program: "program" SPACE CNAME SPACE "{" SPACE (variable_decl ";" SPACE)+ (statement ";" SPACE)+ "}"



expr: NUMBER
    | (expr _NEWLINE expr)
    | expr SPACE "+" SPACE expr
    | expr SPACE "-" SPACE expr
    | expr SPACE "*" SPACE expr
    | expr SPACE "/" SPACE expr
    | "(" expr ")"


variable_decl: "var" SPACE CNAME
             | "let" SPACE CNAME
             | "const" SPACE CNAME



statement: while_statement
         | if_then_statement
         | print_statement
         | assignment_statement
```

Figure 4.4: Lark grammar for our *test grammar*

```
------------------------------
start:

program _a { const __; const _G; print(1); print(2); print(7); r_ = 0; print(8); print(0); _R = 4; print(4); }
------------------------------
------------------------------
start:

4 * 4 + 7
------------------------------
------------------------------
start:

let HQ
------------------------------
------------------------------
start:

if ( 7 ) { _ = 3; print 2; P = 7; _ = 6;  }
------------------------------
------------------------------
```

Figure 4.5: Generated terminal string for nonterminal `start` for our *test grammar*

generated from `program`, where the *plus* element containing `variable_decl` has been repeated two times, followed by the *plus* element containing `statement`, that is repeated eight times. The second terminal string is generated from `expr`, for multiplication, "`*`", and addition, "`+`". The third terminal string is from the `variable_decl`, and the last from `statement`, a `if_statement`.

Generated output for `variable_decl` can be seen in Figure 4.6. `variable_decl` also has multiple options for generating, three different keywords: "*var*", "*let*" and "*const*". The algorithm generated one example for each option. The nonterminal `statement` also have multiple production rules, *while*, *if*, *print* and *assign*. In Figure 4.7, examples of generated terminal strings for these four options are shown.

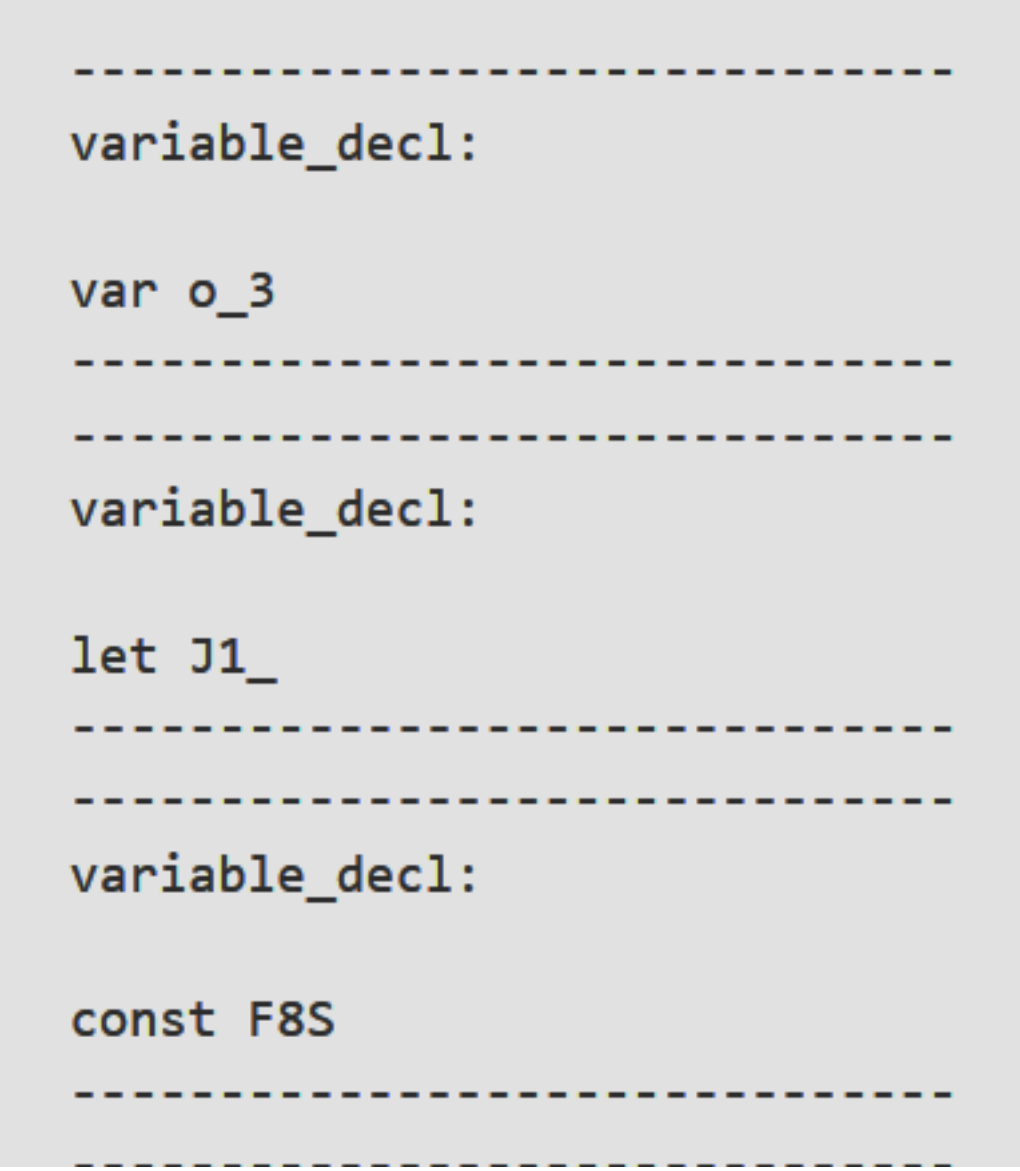

Figure 4.6: Generated terminal string for nonterminal `variable_decl` for our *test grammar*

```
---------------------------------
statement:

while ( 9 + 4 ) { print 8; print 4;  }
---------------------------------
---------------------------------
statement:

if ( (3) ) { print(9); print(7);  }
---------------------------------
---------------------------------
statement:

print(8 * 4 - 9)
---------------------------------
---------------------------------
statement:

w_vk_ = 3
---------------------------------
```

Figure 4.7: Generated terminal string for nonterminal `statement` for our *test grammar*

Chapter 5

Related Work

5.1 Other Grammar Formalisms

For the algorithm we had the context-free grammar in mind, or more specifically the grammar formalism EBNF. But the algorithm could work for multiple types of grammars, not just one. Meaning, it could be generalized to work with other grammars. Such as conjunctive grammars, Boolean Grammars or grammars with context.

For the implementation we went for the parser generator Lark, but here we also have other options. A parser generator, or compiler compilers as it is also called, is a tool that takes a grammar and creates a parser for that grammar[2]. A parser is a code that given a sequence try to match it against the grammar to see if it is valid[10]. If it is valid the parser divides the sequence into parts matching the grammar. For example subject, verb and object for a sentence. The returned result after parsing is a parse tree, that shows how the sequence matches the grammar. Examples of other parser generators are; ANTLR, Yacc/Bison, CoCo/R, etc.

5.2 Prior art

There exists several relevant papers on the subject of grammars in the context of software engineering. Some of these papers cover the broad sens of grammars and other focuses on certain parts. One article that looks at grammar in a general way is "Toward an Engineering Discipline for Grammarware" by Paul Klint, Ralf Lammel and Chris Verhoef [9]. I will look a little more into this article before moving on to other works.

5.2.1 Grammarware

The article "Toward an Engineering Discipline for Grammarware" [9], look at current software engineering practices for grammarware and the way to evolve them. Within this article, the term grammarware is defined as to include both grammar and grammar-dependent software. Here grammar is used for all established grammar notations and formalisms, and grammar-dependent software for software that, in an essential manner, involves grammar knowledge. The goal of the authors is to shine a light on the missing engineering discipline for grammarware, and by doing so, opening up for improving the overall quality of grammarware. To achieve this, they focuses on identifying the current challenges within grammarware, and then suggesting further research topic to address the identified challenges.

In software systems *structural description* is what we use the term *grammar* for. By doing so we apply some informal, non-strict, assumptions. The assumptions are non-strict as to not exclude any grammar forms that do not yet exist, or that we have not thought of. As grammar is used for structural description, software component with grammatical structure is used for *grammar dependency*.

The article also talks about grammar formalisms, and grammar notations. The grammar formalism context-free grammars, algebraic signatures and regular tree and graph grammars, is what the authors looks at as the foundations for grammar. Grammar notation is what the structural description is given in, for example: Syntax Definition Formalism (SDF), Backus-Naur Form (BNF) and Extended Backus-Naur Form (EBNF).

For grammar formalisms there are different operation one can do to map one formalisms to another, this can be used to convert between different formalisms. It is also possible to do the same for some grammar notations. For example: one can convert EBNF to BNF, and BNF to EBNF. This is a bidirectional conversion, in other cases the conversion might be uni-directional.

The purpose of a structural description is what we refer to as the *grammar use case*. There are two types of use cases that we differentiate: abstract and concrete. A grammar requires a commitment to a concrete use case before it can be executable. Software that supports concrete use cases we call *meta-grammarware*.

Despite the fact that grammars is a big part of development processes and software systems, there exists no set of best practices for grammarware. Because of this lack of beat

practices there is no engineering discipline for grammarware. This means that working with grammarware is more like hacking than engineering. To change from hacking to engineering there are several things that can be done. Lack of best practises is not the only thing missing from grammarware. There is a lack of coverage in university curriculum, or in books. It is also missing a foundation.

The overall goal of grammarware engineering is to improve the quality of grammarware. To achieve this the authors mentions four promises; *Increased Productivity*, *Improved Evolvability*, *Improved Robustness* and *Fewer Patches, More Enhancements*. A grammarware engineering will increase productivity via grammar recovery, systematic processes and automation in grammarware life cycle. Grammarware engineering can also help connect grammars, so changes in one grammar use case can be transferred to other use cases, this will help to improve evolvability. Robustness in grammarware will be improved by the same link between grammars as just mentioned, and also by testing and reuse of grammars. The last promise, fewer patches, more enhancements, will be achieved by there being less failures in grammarware. This will lead to fewer patches which in turn gives more time for enhancements.

Further, the article goes on to name principles that grammarware engineering should be based on. These principles can help grammarware life cycle which will again reinforce the software life cycle. These principles are:

- **Start from base-line grammars.** Base-line grammars, in the grammarware life cycle, are pure grammars. It is from these grammars grammarware shall start from. Without commitment to implementation, use case or technology.
- **Customize for grammar use cases.** Then from these base-line grammars we can develop new grammars via customization. There exist ways of doing this customization, but they are manual and only created when needed.
- **Separate concerns in grammarware.** Grammarware can be separated into grammar-based concerns and grammar concerns. There exist research that have concluded in techniques for this, but more research and development is needed.
- **Evolve grammarware by transformation.** In grammarware the evolution of a grammar will not only affect the grammar, but also grammar-dependant software and other data. So we need automated transformations that will not only transform the grammar, but all the other aspect of grammarware as well.
- **Reverse-engineer legacy grammarware.** Base-line grammar will not in all cases be available. So we need a way to reverse-engineer data containing grammar knowledge.

- **Ensure quality of grammarware.** We need ways of ensuring quality in grammarware. This also has to be divided into grammar and grammar-dependent software.

This article looks at grammar in a very abstract and general manner, where we look at a specific grammar formalism and how to work with it. But with further work on this algorithm, it should be capable to generalize it to work on other grammar formalisms as well.

5.2.2 Other Papers

Another relevant paper is "Recovery, Convergence and Documentation of Languages" by Vadim Zaytsev [17]. This is a PhD thesis that talk about 5 fields of research:

- grammar recovery,
- grammar extractions,
- grammar convergence,
- grammar transformations and
- language documentation.

Grammar recovery is a way to go from some data, containing information about a grammar, to a grammar. In Zaytsev's PhD, they took inspiration from grammar recovery that start with the extraction of this knowledge. [17] talks about grammar extraction as *abstraction by extraction* and *grammar extractors*. Where extractors are software components that process software artefacts and produce grammar. Grammar convergence is, as Zaytsev says in [17]; "a verification method that establishes and maintains grammatical correspondence constraints on software". Grammar transformation have been along for a long time, but Zaytsev is more interested in automated grammar transformation. The last research field in his PhD thesis is an elaboration on language documentation.

5.2.3 Example-Driven Approaches in Programming Languages

When a programmer is looking at code, most will mentally run an example through the code to try and understand what it does. Running examples through code is also a great way for beginner programmers to see what their code actually dose. Or to learn how different constructs of a language actually works. Not just seeing the end result after

running the code, but what actually happens during runtime. What, how and where different variables get updated, in which order functions are called, etc.

A tool that gives a visualization of examples running through ones code, while coding, would be a good way to learn, and also improve the coding experience for every coder. Updating the examples automatically if changes are made to the code. One would get immediate feedback on the changes. There already exist different tools for this, with different ways of showing the examples. In the implementation in [4] the examples are shown in a separate view from the one showing the coding. while in [11] it is added as comments in the code.

The implementation in [4] lets the user see the steps of the run through, and have functionality for stepping back and forth seeing the state of the program at different times. Like a more advanced form of debugging, Which is another usage for the example visualization. This can further be used for testing, as well as teaching, debugging and improving coding experience, so why is it not used more?

In [11] they argue that the reason it is not used more is because it takes to much effort to make it work for multiple languages and multiple runtime environments. The [4] implementation is a plugin for Eclipse and only works for Java. To make a similar tool for another language will require a lot of the same implementation a new. [11] uses some other technology, Language Server Protocol (LSP) and language implementation frameworks, to make a tool that will work regardless of the language being used. But it does not have all the functionalities as the [11] implementation. The article also discusses the benefits and limitation of this language independent implementation.

5.3 Existing Tools for Assisting Learning Programming Languages

PyCharm Edu is an existing tool for learning and teaching python[8]. One can learn/teach both theory and practice, through interactive tasks. PyCharm Edu offers different kind of tasks such as multiple choice, fill in the blank spaces and more comprehensive coding assignments. One can also get feedback on ones responses with hint and error messages. An educator can create their own courses with tasks and choose whom to share their courses with. One can make it public, so everyone that wants can take it, or private and only share it with a chosen group of people, for instance, a teacher can create a course

and only shear it with his/her students. As a learner, one can try every public course, and private courses that have been shared with you.

Another existing tool is CodingBat, CodingBat is a free online resource that can help someone learn Java and/or Python[14]. This is a website consisting of a lot of smaller coding task to help the person build a coding foundation. Since this is a website, there is no need to install any software. Just code ones answer directly on the web page, and get an immediate response on whether or not the answer was correct. CodingBat was created by a computer science lecturer at Stanford, *Nick Parlant*, as a tool for practicing pure coding, without any distraction. The task at CodingBat is good for beginner programmers, and for practicing.

Pythontutor.com is another online website that can help someone learn programming[1]. It is a vizualisation tool, that works for Python, Java, C, C++ and JavaScript. Here one can paste inn, or write code, and see what actually happen when it runs. What variables the code have, and their current values, what is printed etc. This tool can be used to debug code, or to learn how different constructs of a programming language actually works.

Chapter 6

Conclusion and Future Work

In this thesis we consider the most straightforward grammar formalism to specify syntax of programming languages, context-free EBNF grammars. With this grammar formalism in mind we designed an algorithm for parsing the grammar and generating syntactically correct sample strings for the programming language of the grammar. This thesis gives a formal presentation of the grammar and its different parts.

A prototype implementation of the algorithm is given in Python. For the implementation, we focused on a particular EBNF format, namely, Lark. The choice of using this particular parser generator is motivated by the fact that it is Python-based, which lowers the entry bar for grammar and parser developers. In addition, Lark is currently gaining some popularity; it is used, for instance, to specify the programming language Hedy [6]. The implemented prototype is tested on different Lark grammars for programming languages, such as Hedy level 1 and a simple programming language grammar.

We have identified the following directions for future work:

- extending the class of grammars that the algorithm supports;
- implementing a user-friendly GUI for the algorithm, for example, as a web-based tool;
- conducting a large-scale user study on usability of our algorithm for various examples of grammars;
- implementing heuristics to generate meaningful regular expressions;
- enabling users to modify generated code samples, with these modifications being propagated back to the grammar specification.

Listing 6.1: An example for conjunctive grammars.

```
Correct by context          Not correct by context

    S -> A B                    S -> A B
    A -> hello                  A -> hi
    B -> world & < D            B -> world & < D
    D -> hello                  D -> hello

    S = hello world             S = hi world
```

The first direction is to **extend the class of grammars that the algorithm supports**. Here two ideas can be explored: one is to include static semantics (i.e., context conditions) into the syntax specifications, and the other is to modify the algorithm to support other grammar classes beyond context-free grammars as the input to our algorithm.

As the algorithm is designed now, it can not generate *meaningful* examples, but rather only *syntactically correct* examples. The algorithm has no way of knowing that a variable needs to be declared before it is used; this is because this information—which constitutes the semantics of a language—is not included in its grammar. Thus, a valuable direction for future work is to find a way to include the semantics of the language in the input for the algorithm.

Expanding the grammar classes that the algorithm supports beyond context-free grammars would make the algorithm more versatile. Other grammar classes the algorithm could be extended to support are grammars with contexts, conjunctive grammars, and Boolean grammars. When talking about grammars with contexts, there are two forms, grammars with one-sided context (left or right), and grammars with two-sided contexts [3]. There are two types of contexts, regular context and extended context. Left context says what should prefix the symbol, while right context says what should follow the symbol. With extended context, the symbol itself is included in the context. For instance, with left context, one could say that an object needs to be declared before use, or after use with right context. In Listing 6.1, one can see an example where the symbol `S -> A B` is correct according to the left context of `B` to the left, and not correct according to the left context of `B` to the right. More examples of grammars with context can be found in [3, p. 21-30].

Conjunctive grammars are grammars where the production rules have *intersections* [12].

Listing 6.2: An example for conjunctive grammars.

```
X → A B & C D                X' → A' B' & C' D'

A → h e l l o w              A' → h e l l o w
B → o r l d                  B' → o r l d
C → h e l                    c' → h e l
D → l o w o r l d            D' → l o w

X : h e l l o w o r l d & h e l l o w o r l d
X': h e l l o w o r l d & h e l l o
```

Listing 6.3: An example for Boolean grammars.

```
X → A B & not C D            X' → A' B' & not C' D'

A → h e l l o w              A' → h e l l o w
B → o r l d                  B' → o r l d
C → h e l                    c' → h e l
D → l o w o r l d            D' → l o w

X : h e l l o w o r l d & not h e l l o w o r l d
X': h e l l o w o r l d & not h e l l o
```

$$\mathbf{A} \rightarrow \mathbf{B} \ \& \ \mathbf{C}$$

When parsing conjunctive grammars, it has to be parsed according to every *intersection*. For `A`, both `B` and `C` have to be valid for `A` to be valid. Boolean grammars are conjunctive grammars with *negation* [13].

$$\mathbf{A} \rightarrow \mathbf{B} \ \& \ \mathbf{not} \ \mathbf{C}$$

Here `A` is valid if `B` is valid and `C` is not valid. For example, with conjunctive grammars, the grammar to the left in Listing 6.2 is valid for `X` since `A B` is the same as `C D`, even if `A` is not equal to `C`. Furthermore, the grammar to the right is invalid for `X'`, since `A' B'` is not the same as `C' D'`, $h\ e\ l\ l\ o\ w\ o\ r\ l\ d \neq h\ e\ l\ l\ o\ w$. With Boolean grammars we have *negation*, so if we add *negation* to the grammar, as we have done in Listing 6.3, the opposite is true, `X` is invalid and `X'` is valid. `X` is invalid because `A B` and `C D` are the same, and we want them not to be, which is why `X'` is valid.

The current implementation of the algorithm is done in Python, and the running of the code is done in the terminal. To make the algorithm more usable, and accessible to more users, it would be valuable to **implementing a user-friendly GUI for the algorithm**, for example, as a web-based tool. Using such a tool would make it easier to use for programmers of all levels, even for those who only explore their first steps in programming.

The third direction, **conducting a large-scale user study**, would give the chance to survey potential users' preferences. One could have different groups of user to include in such a survey: for example, inexperienced programmers, or, if our algorithm is framed as a tool for learning programming language's syntax, the target group of a survey could be programming teachers. Another target group could be users interested in designing grammars, and using our algorithm to test whether their specifications comply with the examples they get.

The fourth direction is to **implement heuristics to generate meaningful regular expressions**, that is, to focus on how one can generate sensible output for a given regular expression. This implementation could then be used in our algorithm for generating meaningful and user-friendly strings that comply with regular expressions. For example, for a token specification `IDENTIFIER: /[a-zA-Z_]\w*/`, meaningful samples of generating strings are `var_a`, `apples`, or `maxValue`, rather than just random sequences of symbols that unsystematically comply with the specification.

The final direction is to **enable users to modify generated code samples, with these modifications being propagated back to the original grammar specification.** Based on these changes by the user, an algorithm would modify the original grammar to match the new modified example.

$$\mathbf{G} \xrightarrow{\textbf{generate}} \mathbf{Ex} \xrightarrow{\textbf{modify}} \mathbf{Ex'} \xrightarrow{\textbf{generate}} \mathbf{G'}$$

Glossary

Eclipse Eclipse is an integrated development environment (IDE), mostly used for Java, but can be used for other programming languages as well..

Hedy Hedy is a gradual language, on their website kids can learn programming gradually with there language. It consist of different levels, where different concepts of programming is added for each new level. This way the learner can focus their learning on one thing at the time..

List of Acronyms and Abbreviations

BNF Backus-Naur Form.

EBNF Extended Backus-Naur Form.

IDE integrated development environment.

LSP Language Server Protocol.

SDF Syntax Definition Formalism.

Bibliography

[1] Learn Python, JavaScript, C, C++, and Java.
URL: https://pythontutor.com/.

[2] Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Reading 18: Parser Generators, 2015.
URL: https://web.mit.edu/6.005/www/fa15/classes/18-parser-generators/.

[3] Mikhail Barash. Defining Contexts in Context-Free Grammars. Sep 2015.
URL: https://www.utupub.fi/bitstream/handle/10024/113793/TUCSDissertationD204.pdf.

[4] Jonathan Edwards. Example centric programming. *SIGPLAN Not.*, 39(12):84–91, dec 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052894.

[5] Felienne. hedy/grammars/level1.lark, aug 2022.
URL: https://github.com/Felienne/hedy/blob/main/grammars/level1.lark.

[6] hedy Felienne. Hedy, A gradual programming language, sep 2022.
URL: https://www.hedycode.com/.

[7] JavaTpoint. Chomsky Hierarchy, 2021.
URL: https://www.javatpoint.com/automata-chomsky-hierarchy.

[8] JetBrains. PyCharm Edu, 2000-2022.
URL: https://www.jetbrains.com/pycharm-edu/.

[9] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, jul 2005. ISSN 1049-331X. doi: 10.1145/1072997.1073000.

[10] Ben Lutkevich. parser, 2019-2022.
URL: https://www.techtarget.com/searchapparchitecture/definition/parser.

[11] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. Example-Based Live Programming for Everyone. Nov 2020.

[12] Alexander Okhotin. Conjunctive Grammars. *Journal of Automata, Languages and Combinatorics*, July 2001.
URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.733.2795&rep=rep1&type=pdf.

[13] Alexander Okhotin. Boolean grammars. *Information and Computation*, 194(1): 19–48, 2004. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2004.03.006.

[14] Nick Parlante. CodingBat.
URL: https://codingbat.com/about.html.

[15] Erez Shinan. Welcome to Lark's documentation!, 2020.
URL: https://lark-parser.readthedocs.io/en/latest/.

[16] Erez Shinan. lark.lark, 2020.
URL: https://github.com/lark-parser/lark/blob/master/lark/grammars/lark.lark.

[17] Vadim Zaytsev. *Recovery, Convergence and Documentation of Languages*. Phd thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 2010.

Appendix A

Lark grammar for Hedy level 1 [5]

```
// symbols. they start with an underscore so they don't appear in the parse tree
// (Lark convention)

_SPACE: " "+
_COMMA: _SPACE? (","|""|"") _SPACE? // support latin and arabic and Chinese commas
                                   // and always allow these to be surrounded by spaces
_COLON: _SPACE? ":" _SPACE?
_LEFT_BRACKET : _SPACE? "(" _SPACE?
_RIGHT_BRACKET : _SPACE? ")" _SPACE?
_LEFT_SQUARE_BRACKET :  _SPACE? "[" _SPACE?
_RIGHT_SQUARE_BRACKET : _SPACE? "]" _SPACE?
_HASH: "#"
_SMALLER : _SPACE? "<" _SPACE?
_LARGER: _SPACE? ">" _SPACE?
_EQUALS: _SPACE? "=" _SPACE? //always allow = to be surrounded by spaces
_DOUBLE_EQUALS: _SPACE? "==" _SPACE?
_NOT_EQUALS: _SPACE? "!=" _SPACE?
_SMALLER_EQUALS : _SPACE? "<=" _SPACE?
_LARGER_EQUALS: _SPACE? ">=" _SPACE?
_EXCLAMATION_MARK: "!"
_QUESTION_MARK: "?"
_PERIOD: "."
_SINGLE_QUOTE: "’" | "‘" | "’"
_DOUBLE_QUOTE: "\""
_QUOTE: _SINGLE_QUOTE | _DOUBLE_QUOTE
_PLUS: _SPACE? "+" _SPACE?
```

```
_MINUS: _SPACE? "-" _SPACE?
_MULTIPLY: _SPACE? "*" _SPACE?
_DIVIDE: _SPACE? "/" _SPACE?

start: program
program: _EOL* (command _EOL+)* command?
command: print | ask | echo | turtle | error_invalid_space | error_invalid

print: _PRINT (text)?
ask: _ASK (text)?
echo: _ECHO (text)?
turtle: _FORWARD ((NUMBER
      | text))? -> forward
      | _TURN ((left | right ))? -> turn
      | _COLOR ((black | blue | brown | gray | green | orange
                | pink | purple | red | white | yellow | text))? -> color
error_invalid_space: _SPACE any
error_invalid: textwithoutspaces text?

any: /.+/ -> text

COMMENT: _HASH /([^\n]+)/
%ignore COMMENT

_EOL: "\r"?"\n"

NEGATIVE_NUMBER: _MINUS /[0-9]+/ ("." /[0-9]/)?
POSITIVE_NUMBER: /[0-9]+/ ("." /[0-9]/)?
NUMBER: NEGATIVE_NUMBER | POSITIVE_NUMBER
INT: _MINUS? /[0-9]+/

//anything can be parsed except for a newline and a comment hash
text: /([^\n#])([^\n#]*)/ -> text
// to properly deal with tatweels,
// we also need to prevent text from starting with a tatweel,
// otherwise we might parse    as the printing of ___ (see #2699)
//anything can be parsed except for a new line, spaces and a comment hash
```

```
textwithoutspaces: /([^\n #]+)/ -> text


// FH Sept 2021: More info on this variable format:
// https://www.unicode.org/reports/tr31/tr31-1.html
// Exact grammar stolen from:
// https://lark-parser.readthedocs.io/en/latest/classes.html

NAME: LETTER_OR_UNDERSCORE LETTER_OR_NUMERAL*

LETTER_OR_UNDERSCORE: /[A-Za-z]+/
LETTER_OR_NUMERAL: LETTER_OR_UNDERSCORE | /[A-Za-z0-9]+/

// Internal symbol added by the preprocess_blocks function
// to indicate the end of blocks
_END_BLOCK: "end-block"



// keywords-en.lark
// https://github.com/Felienne/hedy/blob/main/grammars/keywords-en.lark

_PRINT: ("print" | "print") _SPACE?
_ASK: ("ask" | "ask") _SPACE?
_ECHO: ("echo" | "echo") _SPACE?
_FORWARD: ("forward" | "forward") _SPACE?
_TURN: ("turn" | "turn") _SPACE?
left: ("left" | "left") _SPACE?
right: ("right" | "right") _SPACE?
black: ("black" | "black") _SPACE?
blue: ("blue" | "blue") _SPACE?
brown: ("brown" | "brown") _SPACE?
gray: ("gray" | "gray") _SPACE?
green: ("green" | "green") _SPACE?
orange: ("orange" | "orange") _SPACE?
pink: ("pink" | "pink") _SPACE?
purple: ("purple" | "purple") _SPACE?
```

```
red: ("red" | "red") _SPACE?
white: ("white" | "white") _SPACE?
yellow: ("yellow" | "yellow") _SPACE?
_IS: _SPACE ("is" | "is") _SPACE
_SLEEP: ("sleep" | "sleep") _SPACE?
_ADD_LIST: ("add" | "add") _SPACE
_TO_LIST: _SPACE ("to" | "to") _SPACE
_REMOVE: ("remove" | "remove") _SPACE
_FROM: _SPACE ("from" | "from") _SPACE
_AT: _SPACE ("at" | "at") _SPACE
random: ("random" | "random") _SPACE?
_IN: _SPACE ("in" | "in") _SPACE
_IF: ("if" | "if") _SPACE
_ELSE: "else" | "else"
_AND: _SPACE ("and" | "and") _SPACE
_REPEAT: ("repeat" | "repeat") _SPACE
_TIMES: _SPACE ("times" | "times")
_FOR: ("for" | "for") _SPACE
_RANGE: ("range" | "range") _SPACE?
_TO: _SPACE ("to" | "to") _SPACE
_STEP: "step" | "step"
_ELIF: _SPACE? ("elif" | "elif") _SPACE
_INPUT: ("input" | "input")
_OR: _SPACE ("or" | "or") _SPACE
_WHILE: ("while" | "while") _SPACE
_LENGTH: "length" | "length"
_COLOR : ("color" | "color") _SPACE?
```

Appendix B

Lark grammar for our test grammar

```
start: program
     | expr
     | variable_decl
     | statement


program: "program" SPACE CNAME SPACE "{" SPACE (variable_decl ";" SPACE)+
          (statement ";" SPACE)+ "}"


expr: NUMBER
    | (expr _NEWLINE expr)
    | expr SPACE "+" SPACE expr
    | expr SPACE "-" SPACE expr
    | expr SPACE "*" SPACE expr
    | expr SPACE "/" SPACE expr
    | "(" expr ")"


variable_decl: "var" SPACE CNAME
             | "let" SPACE CNAME
             | "const" SPACE CNAME


statement: while_statement
         | if_statement
```

```
          | print_statement
          | assignment_statement


while_statement: "while" SPACE "(" SPACE expr SPACE ")" SPACE "{" SPACE
                   (statement ";" SPACE)* SPACE "}"
             //    | "while" SPACE "(" SPACE expr SPACE ")" SPACE
             //       ["do" SPACE "{" (statement ";" SPACE)* SPACE "}"]
             //       "until" "(" expr ")"


if_statement: "if" SPACE "(" SPACE expr SPACE ")" SPACE "{" SPACE
                (statement ";" SPACE)* SPACE "}"
            //  | "if" SPACE "(" SPACE expr SPACE ")" SPACE "{" SPACE
            //     (statement ";" SPACE)* SPACE "}"
            //     "else" "{" SPACE (statement ";" SPACE)* SPACE "}"


print_statement: "print" "(" expr ")"
// print_statement: "print" SPACE expr


assignment_statement: CNAME SPACE "=" SPACE expr


SPACE: " "


// From GitHub:
// https://github.com/lark-parser/lark/blob/master/lark/grammars/common.lark
//
// Basic terminals for common use

//
// Numbers
//
```

```
DIGIT: "0".."9"

SPACE: " "

HEXDIGIT: "a".."f"|"A".."F"|DIGIT

INT: DIGIT+
SIGNED_INT: ["+"|"-"] INT
DECIMAL: INT "." INT? | "." INT


_EXP: ("e"|"E") SIGNED_INT
FLOAT: INT _EXP | DECIMAL _EXP?
SIGNED_FLOAT: ["+"|"-"] FLOAT

NUMBER: INT //FLOAT | INT
SIGNED_NUMBER: ["+"|"-"] NUMBER

//
// Strings
//
_STRING_INNER: /.*?/
_STRING_ESC_INNER: _STRING_INNER /(?<!\\)(\\\\)*?/

ESCAPED_STRING : "\"" _STRING_ESC_INNER "\""


//
// Names (Variables)
//
LCASE_LETTER: "a".."z"
UCASE_LETTER: "A".."Z"

LETTER: UCASE_LETTER | LCASE_LETTER
WORD: LETTER+

CNAME: ("_"|LETTER) ("_"|LETTER|DIGIT)+
```

```
//
// Whitespace
//
WS_INLINE: (" "|/\t/)+
WS: /[ \t\f\r\n]/+

CR : /\r/
LF : /\n/
NEWLINE: (CR? LF)+


// Comments
SH_COMMENT: /#[^\n]*/
CPP_COMMENT: /\/\/[^\n]*/
C_COMMENT: "/*" /(.|\n)*?/ "*/"
SQL_COMMENT: /--[^\n]*/

NAME: /[a-zA-Z_]\w*/

STRING : /[A-Za-z]+/
// /[ubf]?r?("(?!"").*?(?<!\\)(\\\\)*?"|'(?!'').*?(?<!\\)(\\\\)*?')/i
LONG_STRING: /[A-Za-z]+/
// /[ubf]?r?("""".*?(?<!\\)(\\\\)*?"""|'''.*?(?<!\\)(\\\\)*?''')/is

DEC_NUMBER: /0|[1-9]\d*/
HEX_NUMBER.2: /0x[\da-f]*/
OCT_NUMBER.2: /0o[0-7]*/
BIN_NUMBER.2 : /0b[0-1]*/
FLOAT_NUMBER.2: /((\d+\.\d*|\.\d+)(e[-+]?\d+)?|\d+(e[-+]?\d+))/
IMAG_NUMBER.2: /\d+j/ | FLOAT_NUMBER "j"

TAB: /\t/
```