Cutting Languages Down to Size

Student Project

at the Cooperative State University Baden-Württemberg Stuttgart

by

Nahku Saidy and Hanna Siegfried

06/08/2020

Time of Project	07/10/2019 - 06/08/2020
Student ID; Course	8540946, 6430174; TINF17ITA
Advisors	Prof. Dr. Stephan Schulz and Prof. Geoffrey Sutcliffe, Ph.D.

Erklärung

Wir versichern hiermit, dass wir die Studienarbeit mit dem Thema: *Cutting Languages Down to Size* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Wir sind uns bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Stuttgart, 06/08/2020

Nahku Saidy und Hanna Siegfried

Die Software wurde mittels Paarprogrammierung entwickelt.
Die schriftliche Ausarbeitung der Arbeit kann wie folgt aufgeteilt werden:

Kapitel	**Nahku Saidy**	**Hanna Siegfried**	**Zusammen**
1	•	•	gesamt
2	•	•	gesamt
3	3.7 - 3.9, 3.12 - 3.13	3.4 - 3.6, 3.10 - 3.11	3.1 - 3.3
4	4.4 - 4.6, 4.7, 4.9	4.1 - 4.3, 4.6, 4.8	•
5	•	•	gesamt
6	•	•	gesamt

Abstract

Computer languages are likely to grow over time as they are getting more complex when their functionality is extended. An example for that is the TPTP language for automated theorem proving. Over time various forms of classical logics ranging from Clause Normal Form (CNF) to Typed First-order Form (TFF) have been included in and extended the TPTP language. This research describes a tool called Synplifier (Syntax simplifier) that automatically extracts sub-languages from the TPTP language. Automatic extraction instead of manually maintaining sub-languages has the advantage of avoiding maintenance overhead as well as unnoticed divergences from the full language. Sub-languages of interest are for example CNF or First-order Form (FOF) and are extracted based on the user's selection which part of the language to maintain. Synplifier has been successfully tested by extracting CNF from the TPTP language.

Contents

Acronyms I

List of Figures II

List of Tables IV

Listings V

List of Algorithms VI

1 Introduction 1
- 1.1 Problem statement and goals . . . 1
- 1.2 Structure of the Report . . . 2

2 Background and Theory 3
- 2.1 Formal languages . . . 3
- 2.2 Backus-Naur Form (BNF) . . . 7
- 2.3 TPTP language . . . 7
- 2.4 Lexing . . . 8
- 2.5 Parsing . . . 9
- 2.6 Lex and Yacc . . . 11
- 2.7 Python . . . 12

3 Concept 14
- 3.1 Approach . . . 14
- 3.2 Requirements . . . 14
- 3.3 Overview . . . 17
- 3.4 Lexer . . . 19
- 3.5 Parser . . . 21
- 3.6 Graph generation . . . 26
- 3.7 Control file . . . 28
- 3.8 Maintaining comments . . . 30
- 3.9 Extraction of a sub-syntax . . . 33
- 3.10 Output generation . . . 35
- 3.11 GUI . . . 38
- 3.12 Command-line interface . . . 41
- 3.13 Counting the number of rules in the TPTP syntax . . . 42

4 Implementation **44**
4.1 Lexer . . . 44
4.2 Parser . . . 46
4.3 Graph generation . . . 53
4.4 Maintaining comments . . . 58
4.5 Extraction of a sub-syntax . . . 66
4.6 Input . . . 73
4.7 Output generation . . . 74
4.8 GUI . . . 81
4.9 Command-line interface . . . 95

5 Validation **96**
5.1 Automated parser generation . . . 96
5.2 Syntax size comparison . . . 98

6 Conclusion **100**
6.1 Future Work . . . 100

Bibliography **i**

Acronyms

ASCII	American Standard Code for Information Interchange
ATP	Automated Theorem Proving
BNF	Backus-Naur Form
CFG	Context-free grammar
CNF	Clause Normal Form
EBNF	Extended Backus-Naur Form
FOF	First-order Form
PLY	Python Lex-Yacc
Synplifier	Syntax simplifier
TFF	Typed First-order Form
THF	Typed Higher-order Form
TPTP	Thousands of Problems for Theorem Provers
URL	Uniform Resource Locator

List of Figures

2.1 Finite automaton 5

3.1 Procedure of extracting a sublanguage 17
3.2 UML diagram of the architecture of Syntax simplifier (Synplifier) . 19
3.3 Parsing procedure 22
3.4 Maintaining comments flow chart 33
3.5 Procedure of generating a Syntax string representation from the grammar graph 36
3.6 Import menu 38
3.7 GUI 39
3.8 View menu 39
3.9 Reduce menu 40
3.10 Save menu 41
3.11 Control file menu 41

4.1 Rule data type UML class diagram 49
4.2 Comment block data type UML class diagram 49
4.3 Production element data type UML class diagram 50
4.4 Production property data type UML class diagram 50
4.5 Production data type UML class diagram 50
4.6 Productions list data type UML class diagram 51
4.7 Grammar list data type UML class diagram 51
4.8 Parsing example 52
4.9 Graphbuilder UML diagram 54
4.10 NTNode UML diagram 55
4.11 RuleType UML diagram 55
4.12 Split comment block by top of page line flow chart 65
4.13 Check if node value is in terminating set 70
4.14 Save ordered rules from graph 75
4.15 View UML class diagram 82
4.16 Reduce and save Thousands of Problems for Theorem Provers (TPTP) syntax 88

4.17 Reduce and save TPTP syntax from control file 90

List of Tables

3.1 TPTP language rule types [7] 20
3.2 Command-line interface parameters 42

5.1 TPTP syntax size comparison 99

Listings

3.1 Rule example 21
3.2 Example of rules defined over multiple lines 21
3.3 Productions of the nonterminal symbol $< formula\_role >$ 27
3.4 Control file 29
3.5 Comment in the TPTP syntax 30
3.6 Comment lines split by a *Top of Page* line in the TPTP syntax . . 30
3.7 Comment syntax definition in the TPTP syntax 37
3.8 Making the comment syntax reachable 37

4.1 Lexer tokens 44
4.2 Lexer regular expressions 45
4.3 Lexer error function 46
4.4 Parser function of productions list 47
4.5 Grammar expression 48
4.6 Production element 52
4.7 Multiple rules with the same left-hand side of the rule 56
4.8 Import TPTP syntax from web 73
4.9 Read text from file 74
4.10 Print List Entry 74
4.11 Production alternatives in the TPTP syntax 78
4.12 Implementation of menu bar 82
4.13 Implementation of menu actions 83
4.14 Init Tree View 84
4.15 Reduce TPTP syntax with selection 86
4.16 Argparse command-line parser configuration 95

5.1 Control file to extract Clause Normal Form (CNF) 97
5.2 *annotated_formula* production rule 97
5.3 CNF parser main-function 98

List of Algorithms

1 Graph Generation Algorithm: buildNodesDictionary 56
2 Graph Generation Algorithm: buildGraphRek 57
3 Algorithm for extracting productions from productions list: searchProductionsListForNT 57
4 Algorithm for appending children to node: searchProductionForNT 58
5 Assign comments to rules 61
6 Append comments to node 62
7 Extend node comment block 63
8 Removing blocked productions 67
9 Removing non terminating symbols 68
10 Find non terminating symbols 69
11 Delete non terminating productions 72
12 Delete non terminating nodes 72
13 Output Algorithm: create_print_list 76
14 *NTNode* string creation 77
15 *PRODUCTIONS_LIST* string creation 78
16 *PRODUCTION* string creation 79
17 *PRODUCTION_ELEMENT* string creation 80
18 GUI Pseudo Code: init_tree_view 84
19 GUI Algorithm: toggle_comments 86
20 GUI Pseudo Code: load_controlfile 92
21 GUI Algorithm: produce_controlfile 94

1 Introduction

1.1 Problem statement and goals

Computer languages are likely to grow over time as they are getting more complex when their functionality is extended and more use cases are covered. On the one hand that leads to a more powerful language capable of handling a wide range of use cases. On the other hand increased complexity makes a language harder to learn and to use. Especially new users are discouraged to implement tools in that language.
One example of a language that has been expanding is the TPTP language for automated theorem proving. Over time various forms of classical logics ranging from CNF to Typed First-order Form (TFF) have been included in and extended the TPTP language.
This report describes a tool called Synplifier that is able to automatically extract sub-languages from the TPTP language. Sub-languages of interest are for example CNF or First-order Form (FOF) and are specified by the user using the application. The goal is maintaining the expressiveness of the whole TPTP language but allowing users to extract a sub-syntax to simplify the language for their particular use case. Synplifier processes a given grammar of a language in multiple steps. First it parses the formal grammar into a structured internal representation using Python Lex-Yacc (PLY). The processed grammar is presented to the user via a GUI. The user can select a start symbol and disable productions that should not be included in the desired sub-syntax. Using the users input, the developed application extracts the sub-syntax from the TPTP syntax and presents the sub-syntax in the same format as the original TPTP syntax. Also, comments present in the TPTP syntax are maintained and associated with the corresponding rules in the reduced syntax.

1.2 Structure of the Report

The first chapter introduces the problem of complex computer languages and the goal of this report that is extracting smaller sub-languages. The second chapter provides necessary background information including the TPTP language, formal grammars, lexing and parsing. The third chapter outlines the concept of Synplifier utilising the information from chapter 2. Based on this, the implementation of Synplifier is featured in the fourth chapter. The fifth chapter presents an evaluation of the effectiveness of Synplifier. Considering the evaluation, the sixth chapter sums up the results, compares them to the defined goals in the first chapter and offers an outlook for possible future research.

2 Background and Theory

This chapter introduces the technologies and background that will be utilized in the following chapters. First, an introduction into the TPTP language is given. Then, formal grammars and the BNF are described. Following that, the foundations of lexing and parsing are outlined. Finally, Python and relevant Python modules that are used in the implementation are presented.

2.1 Formal languages

This section introduces terms and concepts in the area of formal languages that are necessary to understand the concept of Synplifier.

2.1.1 Alphabet

An alphabet is a finite, non-empty set of symbols usually represented by an uppercase sigma $\sum$. An example is the binary alphabet $\sum = \{0, 1\}$. [8]

2.1.2 String

A string, also called word, is a finite sequence of symbols from some alphabet. For example the string *101* is a string from the binary alphabet $\sum = \{0, 1\}$. The set of all strings over an alphabet $\sum$ is denoted as $\sum^*$. [8]

2.1.3 Language

If $\sum$ is an alphabet and L is a subset of $\sum^*$, then L is a language over $\sum$ [8]. An example for a language $L(\sum)$ is the American Standard Code for Information Interchange (ASCII) which is a language over the alphabet $\sum = \{a - zA - Z, 0 - 9, !, =, \#, \%, \$, \&, ', (,), *, +, ,, -, ., /, :, ;, <, >, =, ?, @, [,], \backslash, \hat{}, \_, `, \{, \}, |, \sim\}$. In the

example the abbreviations $a - zA - Z$ and $0 - 9$ are used. The first abbreviation represents all alphabetic letters in lower and upper case. The second abbreviation represents one-digit numbers.

2.1.4 Finite automata

A finite automaton can recognize words of a language. The language represented by an automaton is the set of all accepted words. An automaton consists of a set of states, a set of input symbols and transition functions. A transition function takes as input a state and an input symbol and returns a state based these input arguments. States are usually represented by circles and transitions by labelled arrows. There are two special states: The start state and final state(s). The start state is the state in which the automaton starts processing the input symbols. Starting from the start state the automaton processes the first input symbol by evaluating the belonging transition function that takes the start state and the first input symbol as arguments. The automaton continues to process the input data by evaluating transition functions until the automaton has read every input symbol. The input word is part of the language the automaton represents if the automaton has reached a final state. If the automaton has not reached a final state, the input words is not part of the language the automaton represents.
Deterministic and non-deterministic automata can be distinguished. Non-deterministic automata can have multiple transitions for the same input. Deterministic automata have unambiguous transitions for a given input. The concept of finite automata be applied to a lexical analyzer for recognizing tokens (see section 2.4). [8]
Figure 2.1 shows an automaton consisting of the set of states $\{z_0, z_1, z_2\}$ and several transitions. For convenience not all transitions are printed. If the first input symbol would be a zero the automaton would transit in a so called error state which can not be left regardless the input symbols.
The automaton accepts words consisting of a 1, an arbitrary amount of zeros and a 1.

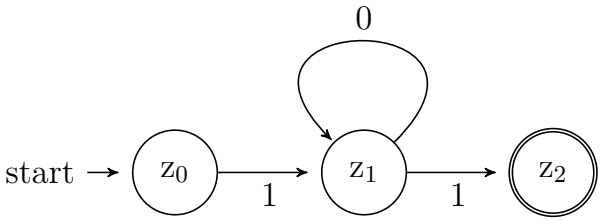

Figure 2.1: Finite automaton

2.1.5 Regular expression

A regular expression is an algebraic description of a regular language. Regular expressions declare strings that are part of the language and can describe the same words that can also be represented by a finite automaton. In comparison to finite automata, the words can be described in an algebraic way. Due to the similarities between regular expressions and finite automata, it is possible to convert regular expressions to finite automata and backwards. Regular expressions are often used to describe tokens that should be recognized by a lexer (see section 2.4). [8]
For example the regular expression **10*** denotes the language consisting of a single 1 followed an arbitrary number of 0's.
Given an alphabet $\sum$, regular expressions are formally defined as followed [8]:

1. The constants ϵ and $\emptyset$ are regular expressions denoting the empty string and the empty set.
2. If a is a symbol of the alphabet $\sum$, then a is a regular expression.
3. If a and b are regular expressions, then the alternation $a|b$ (either a or b), the concatenation ab (a followed by b) and the Kleene star $a*$ (an arbitrary number of a) are regular expressions.

2.1.6 Formal grammars

Formal grammars describe how words of a language can be generated. In comparison to regular expressions, more classes of languages can be described. While regular expressions only describe regular grammars, formal grammars are able to describe all types of grammars of the Chomsky hierarchy.

The description of a grammar consists of four components:

- A set of symbols that defines words of the language. These symbols are called terminal symbols.
- A set of variables called nonterminal symbols. Each nonterminal symbol represents a set of words.
- One nonterminal symbol represents the language that is being defined. This nonterminal symbol is called the start symbol. Other nonterminal symbols help to define the language of the start symbol.
- A set of productions/rules. They recursively describe the language. Each production consists of a nonterminal symbol on the left-hand side that is being substituted by a sequence of terminal symbols and/or other nonterminal symbols on the right-hand side. There can also be empty productions, meaning a nonterminal symbol is substituted by nothing.

[8]

Context-free grammars

A context-free grammar is a grammar that only allows productions rules that are in the form of $V \rightarrow \beta$ with V being a nonterminal symbol and β being a sequence of nonterminal and terminal symbol with an arbitrary length. Context-free grammars are often the basis of a parser (see section 2.5). [8]

Reduced grammars

Grammars are called reduced if each nonterminal symbol is terminating and reachable [9].
Given the set of terminal symbols $\sum$, a nonterminal symbol ξ is called terminating if there are productions $\xi \underset{\rightarrow}{*} z$ so that z can be derived from ξ and $z \epsilon \sum^*$.
In other words, a nonterminal symbol ξ is terminating if there exist production rules so that ξ can be replaced by a string of terminal symbols. [9]
Given the set of terminal symbols $\sum$ and the start symbol S, a nonterminal symbol ξ is called reachable if there are production rules $S \underset{\rightarrow}{*} u\xi v$ so that S can be derived

to $u\xi v$ and $u, v \epsilon \sum^*$.
In other words, a nonterminal symbol ξ is reachable if there exist production rules so that the start symbol can be replaced by a word containing ξ. [9]

2.2 Backus-Naur Form (BNF)

The Backus-Naur Form (BNF) is a language to describe context-free grammars. In the Backus-Naur Form (BNF) nonterminal symbols are distinguished from terminal symbols by being enclosed by angle brackets, e. g. $<TPTP\_File>$ denotes the nonterminal symbol $TPTP\_File$. Productions are described using the " ::= " symbol and alternatives are specified using the "|" symbol. [10] An example for a BNF production would be $TPTP\_File$ ::= $<TPTP\_Input>$ | $<comment>$.
Using this pattern of notation whole grammars can be specified.
The Extended Backus-Naur Form (EBNF) extends the BNF by with following rules:

- Optional expressions are surrounded by square brackets.
- Repetition is denoted by curly brackets.
- Parentheses are used for grouping.
- Terminals are enclosed in quotation marks.

[11]

2.3 TPTP language

The Thousands of Problems for Theorem Provers (TPTP) is a library of problems for Automated Theorem Proving (ATP). Problems within the library are described in the TPTP language. The TPTP language is a formal language and its syntax is specified in an EBNF. [1]

Originally the TPTP used only CNF [2], but over the years expanded to various other types of logics including include FOF [2], Typed FOF [3, 4] and Typed Higher-order Form (THF) [5, 6].

The TPTP syntax uses an extended BNF, with distinct rules for syntax, semantic constraints, tokens, and character classes which will be described in more detail in section 3.4. A small tool chain based on Lex/Yacc for building a basic parser from the syntax is available [7]. While the TPTP is a major success story, the resulting syntax is large and complex. Understanding the syntax is non-trivial. For new developers who want to implement "only" a first theorem prover for CNF, identifying the relevant parts of the syntax and implementing a parser is a significant barrier to entry.

2.4 Lexing

Lexing or a so-called lexical analysis is the division of input into units called tokens [12].
The input is a string containing a sequence of characters. The lexer groups characters of the input string into sequences that are meaningful. These sequences are called lexemes. From each lexeme, the lexer produces a token, which consists of the token name and the lexeme string. The token name is an abstract symbol which is used during parsing. [13]
The tokens are then passed to the parser for syntax analysis.
A lexer needs to distinguish different types of tokens and furthermore decide which token to use if there are multiple ones that fit the input [14].
A simple approach to build a lexer would be building an automaton for each token definition and then test to which automata the input corresponds.
However, this would be inefficient because in the worst case the input needs to pass all automata before the belonging automata is identified. More suitable is building a single automaton that tests each token simultaneously. This automaton can be built by combining all regular expressions by disjunction. Each final state from each regular expression is marked to know which token has been identified.
Potentially final states overlap as a consequence of one token being a substring of another token.
For solving such conflicts, a lexer is separating the input in order to divide it into tokens. Per convention the lexer chooses the longest input that matches any token. [14]
Furthermore, a precedence of tokens can be declared. Usually the token that is

being defined first has a higher precedence and thus will be chosen if possible token matches have the same length. [14]

Besides of writing a lexer manually it can also be generated by a lexer generator. A lexer generator takes a specification of tokens as input and generates the lexer automatically. The specification is usually written using regular expressions.

2.5 Parsing

The aim of parsing is to establish a relationship among tokens generated by a lexer [12]. For doing so, a parser builds a parse tree out of the generated tokens [14]. Similar to lexers, parsers can be generated automatically. A parser generator takes as input a description of the relationship among tokens in form of a formal grammar (see section 2.1.6). The output is the generated parser. [12]
During the syntax analysis a parser takes a string of tokens and forms a syntax tree by finding the matching derivations. The matching derivation can be found using different approaches. The approaches that are used in this report will be introduced in the following.

2.5.1 Bottom-up parsing

In bottom-up parsing a parse tree for an input string is constructed beginning at the leaves working up to the root. The idea is to reduce a string to the start symbol of a grammar, constructing a deviation in reverse. [13]
At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that productions (reduction is reverse step of deviation). The key decisions are when to reduce and what production to apply. These decisions depend on the parser implementation. An example are shift-reduce parsers that are described in the following.

2.5.2 Shift-reduce parsing

Shift-reduce parsers are one class of bottom-up parsers. In shift-reduce parsing a stack is used to hold grammar symbols and an input buffer holding the rest of the

input string. During parsing the input is scanned from left to right. The parser shifts zero or more input symbols on the stack until it can reduce the elements at the top of the stack. The elements at the top of the stack are reduced to the head of the corresponding production. This procedure is repeated until the stack contains the start symbol and the input buffer is empty or an error is detected. [13] There are four actions a shift-reduce parser can perform:

- **Shift**: Shift next input symbol on top of stack.
- **Reduce**: With the right end of a sting to be reduced being at top of the stack and the left end of the string located within the stack, the string is replaced with a chosen nonterminal.
- **Accept**: Declare that input was successfully parsed.
- **Error**: Detect a syntax error.

[13]

There are two kinds of conflicts that can occur during shift-reduce parsing. A shift-reduce conflict occurs if the parser cannot decide whether to shift or reduce. A reduce-reduce conflict occurs if the parser cannot decide which of multiple reductions to make.[13]

2.5.3 LR(k) parsing

LR(k) parsing is the most common type of bottom-up parsing. The L stands for left-to-right, meaning the input is read from left to right, and R stands for constructing a rightmost derivation in reverse. In the context of LR parsing, the rightmost derivation refers to reducing the input to the start symbol. The parameter k refers to the number of inputs symbols that are used as a lookahead. When k is omitted, $k = 1$ is assumed. An LR parser makes shift-reduce decisions based on transitioning to different parsing states. An LR parser consists of a stack and a parsing table defining parsing-action functions and goto functions. An action function takes a state and an input symbol as input and defines which action to perform (Shift, Reduce, Accept, Error). A goto function takes a state and a nonterminal symbol as input and is used to find the next state after a reduction. States represent set of items. An item is a production of the grammar that the parser is based on

with a dot at some position on the right-hand side of the production. An item indicates how much of a right-hand side has already been seen and what the parser is expecting next to apply the production (lookahead). For example $A \to B \cdot C$ indicates that the parsers has already read the input B and expects C in order to reduce the right-hand side to A.[13]
LR parsers are complex to implement, but LR parser generators can be used, which significantly reduces the effort for creating an LR parser. [13]

LALR parsing

An LALR parser (Look-Ahead LR parser) is a simplified LR parser. The simplification consists of merging items that have identical cores. A core is the part of the right-hand side of a productions that is left of the dot. For example an LR parser would separate the items $A \to B \cdot C$ and $A \to B \cdot D$ as their core is identical but their lookahead is different. An LALR parser merges the two items to one item $A \to B \cdot \{C, D\}$. Not considering the lookahead reduces the power of the parser because it might lead to reduce/reduce conflicts. LALR parsers are nevertheless often used because the parsing table is much smaller comparing to other LR parsing techniques due to merging items. [13]

2.6 Lex and Yacc

Lex and Yacc are tools for writing lexers and parsers. They often work together, meaning the parser uses the generated tokens from the lexer as input.

2.6.1 Lex

Lex is a tool written in C that generates a lexer by specifying regular expressions and belonging code fragments. The specified regular expressions are translated into a program that reads an input stream. The program partitions the input stream into strings matching the regular expressions. Once an expression is recognized the corresponding code is executed. For recognizing expressions, Lex builds a deterministic finite automaton. The automaton chooses the longest possible match if several expressions fit the input. [15]

2.6.2 Yacc

Yacc is a tool written in C that can be used for parsing the input of a computer program. For generating a parser using Yacc, a specification that specifies the structure of the input of the parser has to be given. In addition to the specification of the structure, the input to the parser generator also consists of code that is invoked once a structure has been recognized. Yacc turns the input specification into a routine that handles the input. This routine calls Lex to get tokens from the input stream and arranges them based on the specification. [16]

2.7 Python

Since the use cases are all hand-created grammars of relatively small size (dozens to thousands of rules), and the algorithms do not have high complexity, we decided to implement the tool in Python 3. Python is a simple but powerful modern multi-paradigm language with good support and excellent libraries. Python is easy to learn and its power allows it to create complex applications using Python. Libraries that are used in the implementation Synplifier are introduced in the following sections.

2.7.1 PLY

Python Lex-Yacc (PLY) is an implementation of lex and yacc in Python. It is compatible with Python 2 and 3 and is fully implemented in Python. The goal of PLY is to offer a pure-Python implementation of lex and yacc. The implementation aims to rebuild the functionalities of lex and yacc and thus includes:

- Support for LALR parsing (see chapter 2.5.3)
- Support for empty productions
- Support for ambiguous grammars
- Precedence rules
- Error checking and recovery

PLY consists of the two modules *lex.py* and *yacc.py* that are meant to execute together. For example *yacc.py* retrieves tokens from the input stream provided by *lex.py*. *Yacc.py* returns by default an Abstract Syntax Tree. However, it is possible for the user to change the output of *yacc.py* according to his demands. [17] PLY is used to generate the lexer and parser for the TPTP syntax files.

2.7.2 PyQt

PyQt is a Python binding for the cross-platform GUI framework Qt. It is licensed under the GNU GPL version 3. Qt offers a set of C++ libraries and development tools for various things including tools for building GUIs, defining regular expressions, setting up SQL databases, using NFC or interacting via Bluetooth. PyQt implements over 1000 of these Qt tools in Python. Using PyQt a GUI consists of a main window (QMainWindow). A simple GUI can be created by subclassing the QMainWindow, which provides a main window for an application The QTreeView class provides an implementation of a tree view, which can be used to display data in a hierarchically structured way. [18]

2.7.3 argparse

The Python module argparse is a module for creating command-line argument parsers. It provides the means to specify input arguments and automatically creates help and usage messages. It also checks if the given arguments are valid. From the specified input arguments, the module will automatically create a parser for the specified arguments. [19]
This module is used in the implementation of the command-line interface of Synplifier.

2.7.4 Beautiful Soup

Beautiful Soup is a Python library for extracting data out of HTML and XML files. That makes it especially useful for getting information from web pages. [20, 21]
In Synplifier Beautiful soup is used to extract the latest TPTP syntax, which is stored in HTML format, from the TPTP website.

3 Concept

This chapter outlines the concept and the architecture of Synplifier. In section 3.1 a general approach to build Synplifier is outlined and in section 3.2 requirements Synplifier needs to meet are described. Section 3.3 introduces the architecture of Synplifier and its components. The following sections describe the concept of each component that is developed.

3.1 Approach

In order to decompose the full TPTP syntax into smaller sub-syntaxes extracting desired sub-syntaxes by selecting a new start symbol and disabling undesired productions is proposed. The idea is to process the full TPTP syntax in multiple stages. These stages include parsing the syntax into a structured internal representation and presenting it to the user to define undesired productions and the start symbol. To extract the sub-syntax, a recursive algorithm is introduced that checks which productions are still terminating and extracts all reachable productions.

3.2 Requirements

The following section outlines requirements of Synplifier. Requirements are numbered and split into three categories:

- Requirements concerning the GUI are marked by a *G*
- Requirements concerning the Command-line interface are marked by a *C*
- General requirements concerning Synplifier are marked by an *R*

The requirements are described using capitalized keywords like MUST or SHOULD following [22].

[G1]

Synplifier MUST provide a GUI.

[G2]

Synplifier MUST provide the possibility for the user to import a TPTP syntax file into the GUI by specifying a TPTP syntax file path and a desired start symbol.

[G3]

The imported syntax MUST be displayed in the GUI. This includes displaying by the syntax defined productions as well as comments that are associated with these productions.

[G4]

The user MUST be able to select a new start symbol to generate a sub-syntax.

[G5]

The user MUST be able to select which productions should be blocked to generate a sub-syntax.

[G6]

Synplifier MUST provide the functionality for the user to import a control file into the GUI by specifying a control file path.

[G7]

After importing a control file, selections are set according to the control file content in the GUI.

[G8]

Synplifier MUST provide the functionality to display a generated sub-syntax in the GUI.

[G9]

The user MUST be able to export a control file based on the users start symbol selection (G2) and blocked productions selection (G3) in a .txt-format.

[G10]

Synplifier MUST provide the functionality in the GUI for the user to export a generated sub-syntax to .txt format by specifying a file path.

[C1]

Synplifier MUST provide a command-line interface.

[C2]

The command-line interface MUST provide the functionality to extract a sub-syntax with the user specifying the TPTP syntax file and control file path.

[C3]

The extracted sub-syntax is saved in .txt format at a file path that has been specified by the user.

[C4]

The command-line interface MUST provide help information which describe all command-line parameters.

[R1]

When generating a sub-syntax comments referring to rules that are present in the reduced syntax SHOULD be maintained in the sub-syntax.

[R2]

Output TPTP sub-syntaxes SHOULD be compatible to the automated parser generator for the TPTP language [7].

[R3]

A generated sub-syntax MUST only contain terminating and reachable symbols.

3.3 Overview

Figure 3.1 outlines the procedure of extracting a sublanguage of the TPTP language. The first task is to import the TPTP syntax file and extract the tokens inside that file using the lexer. The next phase is for the parser to create a data structure from the tokens, also checking if the syntax in the syntax file was correct. Then, a graph representing the imported TPTP syntax is built.
This graph is subject to manipulation by disabling certain transitions or selecting a new start symbol in the following phase. This includes computation of the remaining reachable and terminating syntax. That new graph represents the syntax of the extracted sub-language. To make this syntax usable, lastly the syntax must be output, based on the new graph, in the same format as the original syntax.

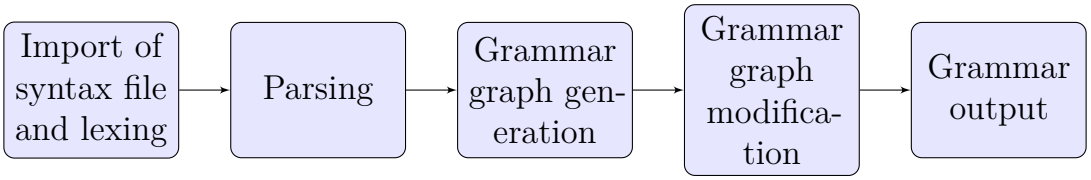

Figure 3.1: Procedure of extracting a sublanguage

3.3.1 Proposed architecture

The proposed architecture for Synplifier consists of seven components:

- A lexer for extracting tokens from the TPTP syntax
- A parser for creating a data structure from the tokens
- A graph builder for building a graph out of the parsers data structure and manipulating this graph
- A graphical user interface
- A console interface
- An input module that imports the filename of the TPTP syntax
- An output module that exports extracted sub syntaxes to local storage

Figure 3.2 contains a high-level UML diagram describing the architecture of Synplifier. The user interacts either with the *Console* or *View* class. The *Console* class provides the command-line interface and the *View* class provides the GUI. Both have a reference on *Input* and *Output* for reading from and writing to files. They also have a reference on the *TPTPGraphBuilder* class. This class is responsible for building a grammar graph and extracting sub-syntaxes by graph manipulation. For that, lexing and parsing are necessary. The *TPTPGraphBuilder* uses the *Parser* class for getting a TPTP syntax representation and the *Parser* uses the *Lexer* to extract the tokens from a TPTP syntax file.

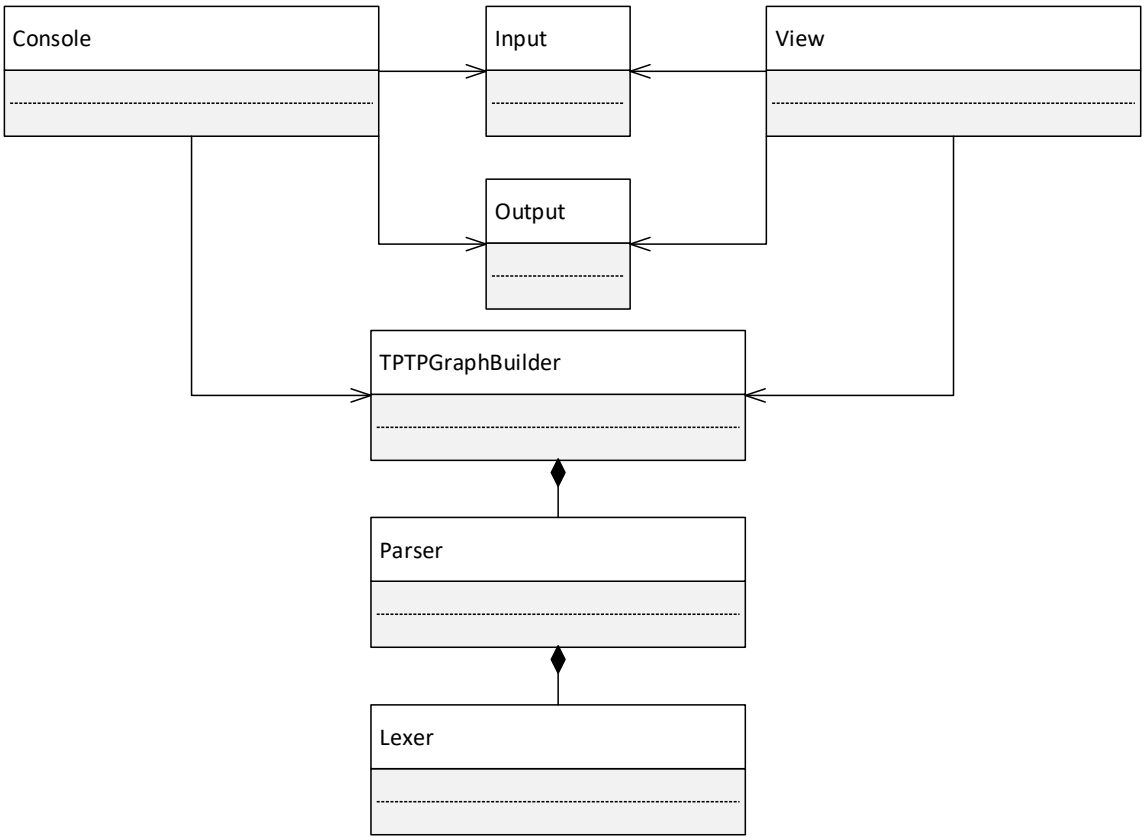

Figure 3.2: UML diagram of the architecture of Synplifier

3.4 Lexer

The lexer is responsible for extracting tokens from the TPTP syntax file. Using PLY a lexer can be generated by specifying tokens using regular expressions. PLY is used because it is the most popular tool for Lex and Yacc in Python. In order to find elementary tokens the TPTP syntax has been analyzed and regular expressions that precisely describe these tokens have been defined.

The standard EBNF only uses one production symbol (” ::= ”). In the TPTP syntax the standard production symbol is used for syntactic rules. Additional symbols for semantic, lexical and character-macro rules have been added. Semantic rules describe rules whose left-hand side is already part of a syntactic rules but from which only a subset makes semantic sense. Any further rules that are reached from the nonterminal symbols on the right-hand side of a semantic rule are semantic

rules as well. Lexical rules are rules that produce tokens. Character-macro rules are rules that define regular expressions for the tokens. The following table contains the production symbols for grammar (syntactic rules), strict (semantic rules), token (lexical rules) and macro (character-macro rules) rule types used in the TPTP syntax.

Table 3.1: TPTP language rule types [7]

Symbol	Rule Type
::=	Grammar
:==	Strict
::-	Token
:::	Macro

The following paragraph introduces the tokens that are recognized by the lexer. Tokens are implemented as Python classes (see chapter 4.1) and written in bold in the following paragraphs.

Following standard BNF, **nonterminal** symbols are enclosed by the < and > symbol. In between there can be any arbitrary sequence of alphanumerical characters and underscores.

A **terminal symbol** does not have any special notation and can be any sequence of characters, excluding whitespace characters, that is not matched to any other token.

There are four **expression** token types (one for each rule type). **Expressions** are defined as a nonterminal symbol followed by a production symbol (::= for grammar, ::- for token, :== for strict, ::: for macro rule type). The nonterminal symbol and the following production symbol are selected to be a single token and are not identified as two separate tokens to clearly identify the start of a new rule and therefore avoid ambiguity while parsing. The listing below features two tokens: A grammar expression and a nonterminal symbol.

```
<formula_role>          ::= <lower_word>
```

Listing 3.1: Rule example

A **comment** is defined as the start of a new line, a percentage sign, arbitrary characters and ends with a newline character. The percentage sign when used as terminal symbol is embedded in square brackets and can therefore never be the first character of a new line.

Additional tokens are the meta-symbols including **open** "(" and **close parentheses** ")", **open** "[" and **close square brackets** "]", asterisks " $*$ " called **repetition symbols** and vertical bars " | " called **alternative symbols**.

In PLY it is possible to declare characters that should be ignored. This means that the characters are ignored in the input stream of the lexer. However, if one of those characters is part of a regular expression of a token it is not ignored and will be used for token matching. In this project tabs, white spaces and newline characters are ignored as they do not have any special meaning other than providing better readability. With exception of the comment token, the information about newline characters is not relevant due to rules being defined over multiple lines, which can be seen in the listing below.

```
<annotated_formula>  ::=    <thf_annotated> | <tff_annotated> |
                            <tcf_annotated> | <fof_annotated> |
                            <cnf_annotated> | <tpi_annotated>
```

Listing 3.2: Example of rules defined over multiple lines

3.5 Parser

The parser takes the tokens from the lexer as input and creates a data structure that represents the structure of the TPTP syntax. The parser is based on a context-free grammar that is described in section 3.5.1.

Figure 3.3 outlines the responsibilities of the parser component and the sequence of its sub-functions. First, the tokens generated by the lexer need to be parsed and based on that the data structure representing the TPTP syntax is to be created. Secondly, the rules in the data structure have to be numbered to maintain the

correct order for output. The third step is the so-called disambiguation of square brackets. In the TPTP syntax, square brackets not necessarily denote that an expression is optional, which is the case in traditional EBNF. In token and macro rules they denote that an expression is optional and in grammar and strict rules square brackets are terminals. Therefore, disambiguation of square brackets is necessary. Finally, the output of the parser is returned.

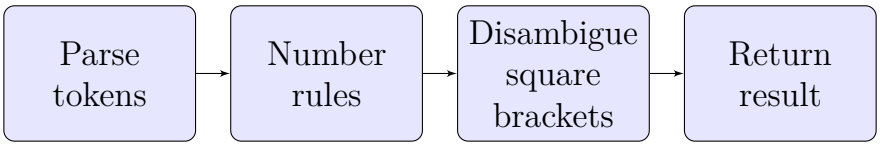

Figure 3.3: Parsing procedure

3.5.1 Defined grammar

The grammar is formally specified $G = (N, \sum, P, S)$.

N is the set of nonterminal symbols. The set includes grammar list, comment block, grammar expression, token expression, strict expression, macro expression, productions list, production, xor productions list, t symbol production and production element. The nonterminal symbols are described in detail in chapter 4.2.

$\sum$ is the set of terminal symbols. Terminal symbols of the specified grammar are the tokens generated by the lexer (see 3.4).

P is the set of production rules that is presented in the following.

A grammar list implies a comment block, the four expressions (grammar expression, token expression, macro expression, strict expression) or a grammar list followed by one of the expressions.

$$
\begin{aligned}
\textit{grammar list} \rightarrow\ & \textit{comment block} \mid \textit{grammar expression} \\
& \mid \textit{token expression} \mid \textit{strict expression} \\
& \mid \textit{macro expression} \\
& \mid \textit{grammar list}\ \ \textit{grammar expression} \\
& \mid \textit{grammar list}\ \ \textit{strict expression} \\
& \mid \textit{grammar list}\ \ \textit{macro expression} \\
& \mid \textit{grammar list}\ \ \textit{comment block}
\end{aligned}
$$

A comment block is either a comment or a comment block with a comment.

$$\textit{comment block} \rightarrow \textbf{comment} \mid \textit{comment block}\ \ \textbf{comment}$$

The four expressions are the expression token followed by their productions list. Grammar expression has a special case without a productions list because the production rule

```
<null>    ::=
```

of the TPTP grammar has no production on the right-hand side.

$$
\begin{aligned}
\textit{grammar expression} \rightarrow\ & \textbf{l grammar expression}\ \ \textit{productions list} \\
& \mid \textbf{l grammar expression}
\end{aligned}
$$

$$\textit{token expression} \rightarrow \textbf{l token expression}\ \ \textit{productions list}$$

$$\textit{strict expression} \rightarrow \textbf{l strict expression}\ \ \textit{productions list}$$

$$\textit{macro expression} \rightarrow \textbf{l macro expression}\ \ \textit{productions list}$$

Productions list and xor productions list imply either a production or a productions list alternative symbol production.

productions list → *production*
| *productions list* **alternative symbol** *production*

xor productions list → *production*
| *xor productions list* **alternative symbol** *production*

T symbol production is either a t symbol or a t symbol/repetition symbol/ alternative symbol embedded in square brackets.

t symbol production → **open square bracket** **t symbol** **close square bracket**
| **open square bracket** **repetition symbol** **close square bracket**
| **open square bracket** **alternative symbol** **close square bracket**
| **t symbol**

A production element can be replaced by a nt symbol or by a nt symbol in square brackets or nt symbol repetition. In the case of repetition or square brackets the production element is categorized as optional when in square brackets or as repetition when followed by the repetition symbol. The same applies to t symbol production. A production element can also only be square brackets only.

production element → **open square bracket** **nt symbol** **close square bracket**
| **nt symbol** **repetition symbol**
| *t symbol production* **repetition symbol**
| **open square bracket** **close square bracket**
| **nt symbol**
| *t symbol production*

production → *production element* | *production* *production element*
| **open parenthesis** *xor* *productions list* **close parenthesis**
| **open parenthesis** *production* **close parenthesis**
| *production* **open parenthesis** *production* **close parenthesis**
| *production* **open parenthesis** *xor productions list* **close parenthesis**
| **open parenthesis** *production* **close parenthesis** *production*
| **open parenthesis** *xor productions list* **close parenthesis** *production*
| **open parenthesis** *production* **close parenthesis** **repetition symbol**
| *production* **open parenthesis** *production* **close parenthesis** **repetition symbol**

The start symbol is not specifically mentioned because it is per convention the left element of the first rule.

3.5.2 Disambiguation of square brackets

As mentioned at the beginning of the parsing section, square brackets have different meanings depending on the rule type. The idea to solve this problem is to treat all rules the same in the first processing step. Square brackets are then interpreted as denoting the optional production property. This production property is then selected for productions that are enclosed by square brackets for all types of rules. In an additional processing step, after creating the grammar list each grammar and strict rule is iterated, exchanging the production property optional by the square bracket terminal symbols. The output of the parser is a list of the rules and the comments from the TPTP syntax file.

3.6 Graph generation

After parsing, the parsed TPTP grammar is stored in a grammar list. The grammar list data structure does not allow easy traversing which is needed for extracting a sub-language. That is why a new data structure is introduced that allows easy modifi

cation and traversing. The data structure that is used is a graph representing the grammar.

In the graph, the nonterminal symbols of the TPTP syntax are represented by a node class that has the following attributes (see also UML class diagram 4.10):

- Value: name of nonterminal symbol
- Productions list: productions list of nonterminal symbols that has been created by the parser
- Rule type: rule type of nonterminal symbol
- Comment block: list of comments belonging to the nonterminal symbol
- Position: position of the production in the input file
- Children: nested list that contains the node of every nonterminal symbol that is part of the productions list grouped by productions in the productions list

Before generating the graph, a dictionary is created that provides an efficient way accessing the nodes in order to build the grammar graph and also during the next steps of the sub-syntax generation. This dictionary contains nodes constructed from the grammar list output by the parser. The combination of the nodes value and rule type form the unique key for each node. While constructing the dictionary, comments in the grammar list are associated to nodes using a heuristic that is described in section 3.8.

Also, a new temporary start symbol is introduced. This is necessary because one nonterminal symbol in the TPTP syntax can be mapped to multiple nodes which the following example 3.3 shows.
The example below shows the productions for the nonterminal symbol $< formula\_role >$. Since this nonterminal symbol has multiple types of rules one node will be created for each rule type.

```
1 <formula_role> ::= <lower_word>
2 <formula_role> :== axiom | hypothesis | definition | assumption |
3                            lemma | theorem | corollary |
                                 conjecture |
4                            negated_conjecture | plain | type |
5                            fi_domain | fi_functors | fi_predicates
                                  | unknown
```

Listing 3.3: Productions of the nonterminal symbol $< formula\_role >$

If a nonterminal symbol that has multiple rule types is selected as the desired start symbol, multiple nodes would represent that start symbol and therefore it would not be possible to select one node as the starting point of the graph generation. To solve this problem, a temporary start symbol is introduced before generating the graph. The temporary start symbol produces the start symbol that the user specified and is used as a starting point for the graph generation. If the non-terminal symbol $< formula\_role >$ that was mentioned before would be selected as start symbol by the user, a temporary start symbol representing the rule

```
<start_symbol>          ::=  <formula_role>
```

would be introduced. This ensures that only one node is representing the start symbol that is used for graph generation.

Starting with the temporary start symbol, the graph is generated recursively. Iterating over each non-terminal symbol in the productions list of the start symbol, the corresponding nodes are identified. These nodes are then appended to the list of children of the start symbol. If a non-terminal symbol corresponds to two nodes with different rule types, the two nodes are stored in a list and are appended to the list of children resulting in a nested list. The identified children may again have children. This process is repeated until a node has no children because there are only terminal symbols in the productions list of a non-terminal symbol.

Since it is possible for a non-terminal symbol to be on the right side as well as on the left side of the same production rule, a node can also be its own child. To avoid revisiting the same node infinitely, it is checked whether a node already has children so that it will not be visited again. This also improves the performance of the tool as a non-terminal symbol that has already been visited will not be visited again independent of circular dependencies.

The following example shows a production rule and an abstract representation of the resulting list of children belonging to the node. Each production alternative is embedded in the children list of the node.

```
<disjunction>  ::=  <literal> | <disjunction><vline><literal>

Output:
node.value:     <disjunction>
node.ruleType:  grammar
node.children:  [[<literal>],[<disjunction>,<vline>,<literal>]]
```

3.7 Control file

A format for specifying the desired start symbol and blocked productions is required. Using a file-based configuration enables the user to store desired configurations so that a manual selection in the GUI is not necessary. It is also essential for using the command line interface, because there manual selection is not possible. The file should be human-readable and -editable. The format should be easy to parse and allow to specify all necessary information. This includes the desired start symbol

and all production rules that should be blocked. The proposed way to describe this information is to:

- Define the desired start symbol in the first line.
- Define blocked productions grouped by the non-terminal symbol and rule type symbol, separating each group by a new line. First defining the non-terminal symbol, then the production symbol and after that the indexes of the alternatives that should be blocked (indexing starts at zero).

Identifying the production symbol is necessary because there are non-terminal symbols that have productions with more than one production symbol.
The example below contains a sample control file. In this file, *<TPTP_file>* is specified as start symbol. The second line indicates that the second grammar production alternative of the non-terminal symbol *<TPTP_input>* should be disabled. Furthermore, the first, second, third and fifth grammar production alternative of the non-terminal symbol *<annotated_formula>* are said to be disabled in line 3.

```
1 <TPTP_file>
2 <TPTP_input>,::=,1
3 <annotated_formula>,::=,0,1,2,5
```

Listing 3.4: Control file

This format is easy to parse and also enables users to specify their desired start symbols and blocked productions without having to use the GUI. The control file can also be generated from selections made by the user in the GUI. It can be exported as a text file. When generating a sub-syntax from the user GUI selection a control file representing that selection is created internally as input for the next processing steps.

Specifying which production should be blocked, and not the ones should be kept, typically results in a significantly smaller file. Storing the indexes of the productions that should be blocked offers that in case productions are renamed the control file would still be valid. On the other hand if productions are added or deleted from the original TPTP syntax, the control file may have to be updated.

3.8 Maintaining comments

In the TPTP syntax there are comments providing supplemental information about the language and its symbols and rules. When generating a reduced grammar maintaining comments is desired. This means that comments from the original language specification should be associated with the rule they belong to and if the rule is still present in the reduced grammar, the comment should be as well.
Therefore, a mechanism has to be designed for the association of comments to grammar rules.

Listing 3.5 features an example of a comment in a TPTP syntax file. This comment begins with a *Top of Page* line which, in the HTML version of the TPTP syntax, contains a hyperlink which leads to the beginning of the syntax file. The next line contains a relevant comment.

```
1 %----Top of Page-------------------------------------------------------------
2 %----TFF formulae.
3 <tff_formula>           ::= <tff_logic_formula> | <tff_atom_typing> |
4                             <tff_subtype> | <tfx_sequent>
```

Listing 3.5: Comment in the TPTP syntax

The heuristic matching comments to rules takes these *Top of Page* lines into account. When there is a *Top of Page* line in between comment lines it generally also splits comments semantically. In listing 3.5 can be seen that the comment in line 2 refers to the rule after. Therefore it would be correct to associate the comment line after the *Top of Page* line to the rule after. Also, if there is one *Top of Page* line in between multiple comment lines it is highly probable that the first part of the comment lines before the *Top of Page* line refer to the rule before the comments and that the lines after the *Top of Page* line refer to the rule after the comment lines. This scenario can be seen in listing 3.6. The *Top of Page* line is in line 28 and the comment lines before refer to the rule before. The comment line after refers to the rule after that line.

```
1 <formula_role>          :== axiom | hypothesis | definition | assumption |
2                             lemma | theorem | corollary | conjecture |
3                             negated_conjecture | plain | type |
4                             fi_domain | fi_functors | fi_predicates | unknown
5 %----"axiom"s are accepted, without proof. There is no guarantee that the
6 %----axioms of a problem are consistent.
7 %----"hypothesis"s are assumed to be true for a particular problem, and are
```

```
%----used like "axiom"s.
%----"definition"s are intended to define symbols. They are either universally
%----quantified equations, or universally quantified equivalences with an
%----atomic lefthand side. They can be treated like "axiom"s.
%----"assumption"s can be used like axioms, but must be discharged before a
%----derivation is complete.
%----"lemma"s and "theorem"s have been proven from the "axiom"s. They can be
%----used like "axiom"s in problems, and a problem containing a non-redundant
%----"lemma" or theorem" is ill-formed. They can also appear in derivations.
%----"theorem"s are more important than "lemma"s from the user perspective.
%----"conjecture"s are to be proven from the "axiom"(-like) formulae. A problem
%----is solved only when all "conjecture"s are proven.
%----"negated_conjecture"s are formed from negation of a "conjecture" (usually
%----in a FOF to CNF conversion).
%----"plain"s have no specified user semantics.
%----"fi_domain", "fi_functors", and "fi_predicates" are used to record the
%----domain, interpretation of functors, and interpretation of predicates, for
%----a finite interpretation.
%----"type" defines the type globally for one symbol; treat as $true.
%----"unknown"s have unknown role, and this is an error situation.
%----Top of Page-------------------------------------------------------------
%----THF formulae.
<thf_formula>           ::= <thf_logic_formula> | <thf_atom_typing> |
                            <thf_subtype> | <thf_sequent>
```

Listing 3.6: Comment lines split by a *Top of Page* line in the TPTP syntax

The flow chart in figure 3.4 shows the process of matching comment blocks, that are consecutive comment lines (see section 4.2.1), to rules. First, the comment block is split into multiple separate comment blocks by using *Top of Page* lines as separators.

- If this results in no comment blocks the comment block consisted only of one line which was a *Top of Page* line. Then no comment block has to be associated to a rule because *Top of Page* lines are not relevant.
- If this results in one comment block, that means that no *Top of Page* line was present in the comment block or that there was a *Top of Page* line at the beginning or end of the comment block. The comment block is associated with the rule after if the comment block is not at the end of the file. If it is at the end of the file, it is associated with the rule before, because no rule after exists.
- If this results in two comment blocks, one *Top of Page* line was present in between comment lines. Then the comment block before the *Top of Page* line is associated with the rule before when possible and the comment block after

the *Top of Page* line is associated with the rule after. If the original comment block was at the beginning of the file, both comment blocks resulting from splitting the original comment block are associated with the rule after. If the original comment block was at the end of the file, the comment blocks resulting from the split are associated with the rule before.

The case of three or more comment blocks after splitting the original comment block is not featured in the flow chart. This case does not occur in the TPTP syntax version 7.3.0. Since it might occur in a future version of the TPTP syntax it is handled by merging all comment blocks starting from the second and then following the procedure of two comment blocks in the flow chart in figure 3.4.

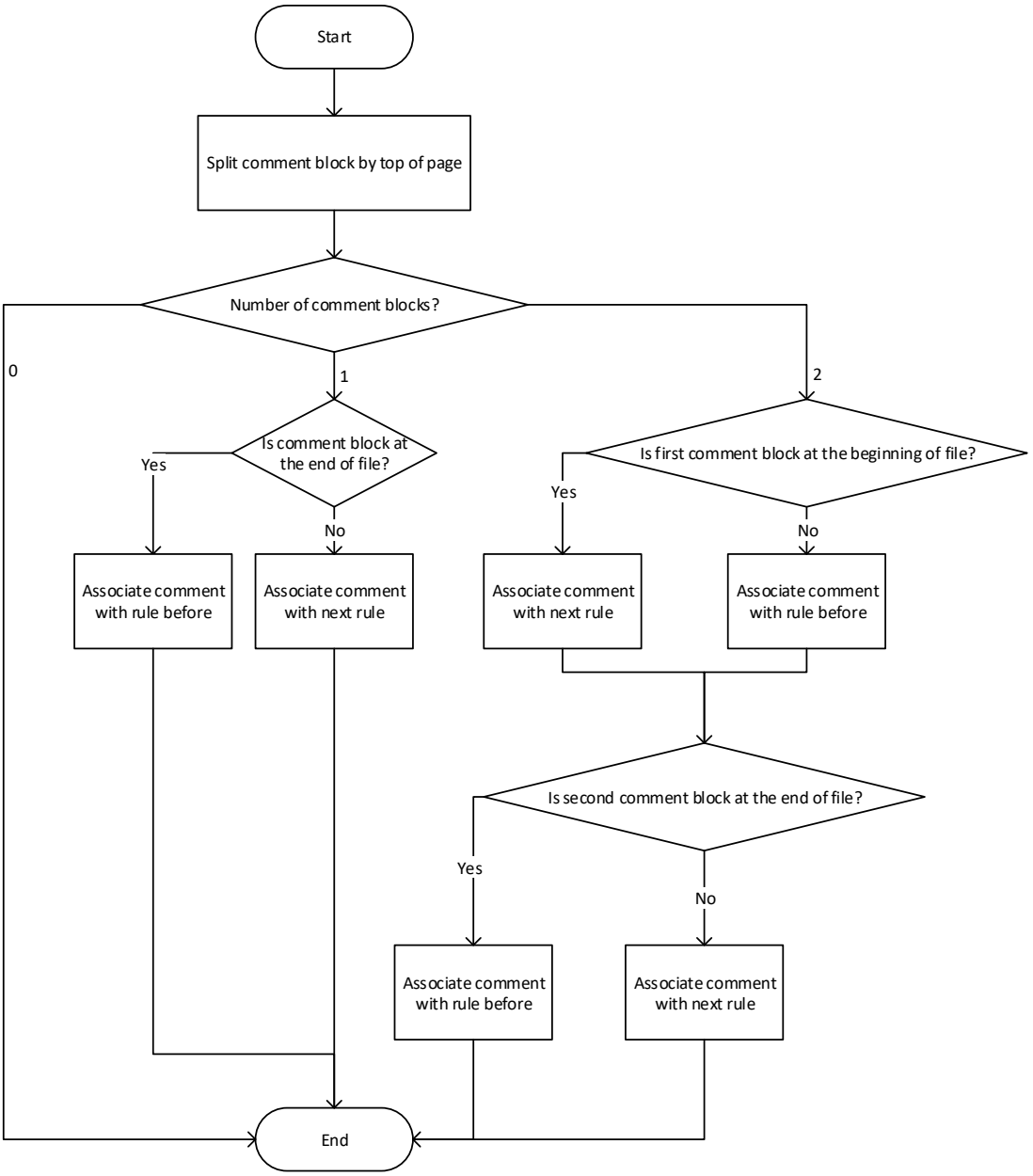

Figure 3.4: Maintaining comments flow chart

3.9 Extraction of a sub-syntax

This section covers the concept of how a sub-syntax is computed from the original syntax. The original syntax is represented by a grammar graph (see section 3.6) and the information on what part of the syntax to extract is described in a control

file. To extract a sub-syntax from the original grammar graph, four steps must be performed:

1. The control file has to be parsed, in order to get information on the desired start symbol and which productions should be blocked.
2. The blocked productions specified in the control file must be disabled and therefore the corresponding transitions must be removed from the grammar graph.
3. The remaining part of the grammar that is terminating must be computed.
4. From the part of the grammar that is terminating, non-reachable symbols must be removed.

3.9.1 Parsing a control file

The control file provides the necessary information for the extraction of a sub-syntax. The format of the control file is described in section 3.7. The start symbol is listed in the first line. Every consecutive line contains a nonterminal symbol, the corresponding rule type symbol, and the indexes of the productions that should be blocked separated by comma. The start symbol will be relevant in determining the remaining reachable part of the grammar (section 3.9.4) whereas the information on the productions that should be blocked will be needed in the next section.

3.9.2 Removal of blocked productions

In the control file, for each nonterminal symbol, productions should be modified, its name, rule type and the indexes of the productions that should be blocked are listed. From all nodes, that are addressed (by nonterminal symbol name and rule type), the indexed productions are removed. This includes deleting the corresponding element from the productions list and from the children list.

3.9.3 Determination of the remaining terminating symbols

After the desired productions have been deleted from the grammar graph, the next step is to remove the nonterminating symbols from the grammar graph. First it must be determined which symbols are terminating and which are nonterminating. Terminating nodes are found by iterating the nodes in the *children_list* of a node. If the *children_list* is empty, the production only consists of terminal symbols. If that is not the case, the node is terminating if all nodes in the *children_list* represent a non-terminal symbol which is terminating. If a non-terminal symbol has multiple rule types, it is represented by multiple nodes. Also, if a non-terminal symbol with multiple rule types is in featured in a production, all nodes representing that non-terminal symbol are in the children list corresponding to this production. However, only one of the nodes needs to be terminating for the non-terminal symbol to be terminating.

After the terminating symbols have been determined, productions that contain a nonterminating symbol must be removed from the nodes of the grammar graph. Because some non-terminal symbols might have productions that contain nonterminating symbols, but also other productions that only contain terminating symbols, only the productions containing nonterminating symbols must be removed.

3.9.4 Determination of the remaining reachable symbols

After removing the productions specified in the control file it has to be determined which part of the grammar still remains reachable. This is done by generating a new grammar graph starting from desired the start symbol. When generating the new grammar graph, all from the start symbol reachable parts of the grammar will be added to the new grammar graph (see section 3.6). All nodes that are not part of the new grammar graph are not reachable and can therefore be removed. They are also removed from the nodes dictionary.

3.10 Output generation

Within Synplifier a syntax is represented by a grammar graph. To export sub-syntaxes that have been extracted from a grammar graph they have to be converted

to the original form of a TPTP syntax file. This process is described in the following subsection. After that, measures to provide compatibility with the automated parser generator are discussed in section 3.10.2.

3.10.1 Create output from grammar graph

There are three steps necessary in order to convert a syntax represented by a grammar graph to the original TPTP syntax file form. These steps are displayed in figure 3.5.

The first step is to traverse the grammar graph by iterating all nodes and getting a string representation of the nodes. For creating the string representation, the Python string class is used [23]. The string class creates a string version of the object passed to the constructor. To do that it calls the $\_\_str\_\_()$ of the passed object, if the object provides that method.

In order to maintain the same order that rules have had in the original TPTP syntax, the position of the nodes is saved together with the node string in a tuple. After generating all node string representations, in the second step all tuples are ordered based on their position.

The ordered list of nodes strings is then, in the third step, written to a file and exported. The node that represents the temporary start symbol is not printed as it does not belong to the TPTP syntax.

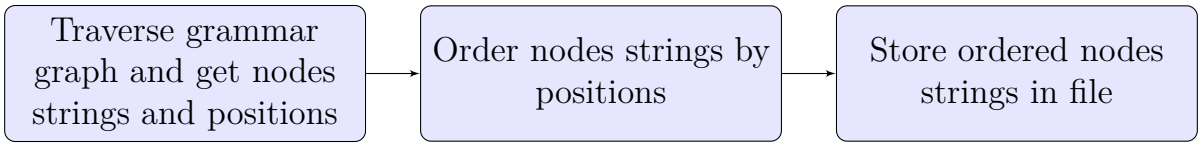

Figure 3.5: Procedure of generating a Syntax string representation from the grammar graph

3.10.2 Automated parser generator compatibility

The automated parser generator for the TPTP syntax [7] takes a TPTP syntax file as input and creates a lex and yacc file corresponding to the specification in the

input syntax. It is uses for validating that an extracted sub-syntaxes is compatible with the original TPTP syntax.
In the TPTP syntax version 7.3.0 there is a definition of the syntax of comments which is shown in listing 3.7.

```
1 <comment>              ::- <comment_line>|<comment_block>
2 <comment_line>         ::- [%]<printable_char>*
3 <comment_block>        ::: [/][*]<not_star_slash>[*][*]*[/]
4 <not_star_slash>       ::: ([^*]*[*][*]*[^/*])*[^*]*
5 <printable_char>       ::: .
```

Listing 3.7: Comment syntax definition in the TPTP syntax

However, this part of the syntax is not reachable from the start symbol $TPTP\_file$. When constructing the grammar graph this part of the syntax is therefore removed. Using a syntax file, in which the comment syntax is not present, with the automated parser generator results in an error because the comment symbol is not present in the lex specification but is referred to in a yacc rule.
There are two ways to include the comment syntax in the output of Synplifier in order to be accepted by the automated parser generator. Either making the *<comment>* nonterminal symbol reachable by for example adding it as an alternative to *<TPTP_input>* which can be seen in listing 3.8 or maintaining the comment syntax separately and adding it to the output syntax even though it is not reachable.

```
1 <TPTP_file>            ::= <TPTP_input>*
2 <TPTP_input>           ::= <annotated_formula> | <include> | <comment>
```

Listing 3.8: Making the comment syntax reachable

Both ways are supported by Synplifier.
If the *<comment>* nonterminal symbol is made reachable in the TPTP syntax, the grammar graph generated by Synplifier contains the comment syntax and no further action is necessary.
To add the comment syntax to the output if the *<comment>* nonterminal symbol is not reachable is done by using an external configuration file that contains the syntax from listing 3.7. From the syntax in this configuration file a separate grammar graph is generated. The output string that can be generated from this grammar graph is added to the output of Synplifier when outputting a generated sub-syntax.

The GUI provides a menu option for appending *<comment>* to the output file (see chapter 3.11.4).

3.11 GUI

The graphical user interface is built using PyQt. Advantages of PyQT are that it offers a treeview that is used for displaying the grammar. In comparison to the standard Python tool kit Tkinter PyQt offers checkboxes in treeviews [24] that are used for selecting blocked productions.

The menu of the GUI consists of five submenus.

3.11.1 Import Menu

The *Import* menu that can be seen in figure 3.6 provides the option to import a TPTP syntax file in txt file format from local storage or to import the latest TPTP syntax version from the TPTP website. The TPTP syntax on the TPTP project website is stored in a HTML format. This file is downloaded and converted to plain text using the Beautiful Soup Python library [20].

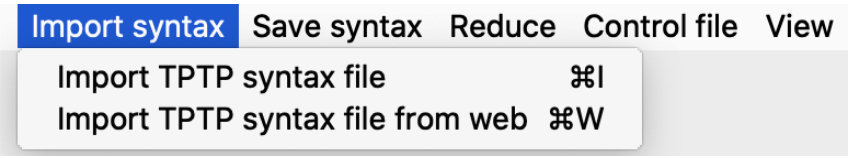

Figure 3.6: Import menu

After selecting a file for import a start symbol needs to be selected. Starting with the start symbol a grammar graph is generated and the corresponding text displayed. Each rule in the TPTP grammar is a top element of the treeview and can be expanded to show the productions alternatives. Right of the nonterminals name the rule type is displayed. Comments that have been assigned to a rule are also displayed. An example of the imported TPTP grammar is shown in figure 3.7.

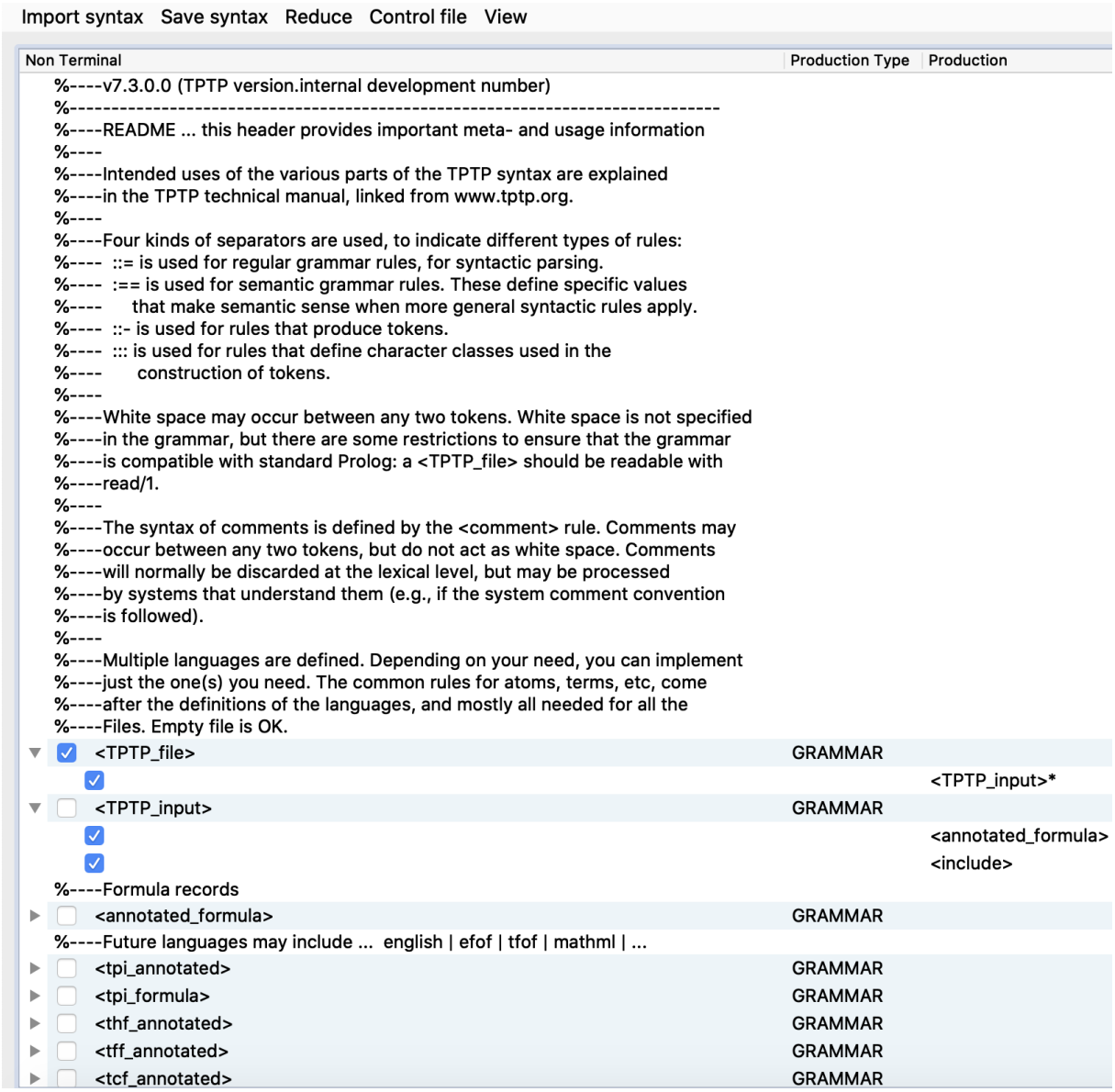

Figure 3.7: GUI

3.11.2 View Menu

The *View* menu, shown in figure 3.8, provides the possibility to toggle the display of comments to improve readability.

Figure 3.8: View menu

In order to extract a sub-syntax the user has to choose a new start symbol by checking one of the check boxes left of the nonterminal symbols name. By default, the start symbol that has been selected after importing the syntax is selected. For selecting blocked productions, the user has the choice of on the one hand expanding the rules and uncheck the checkboxes belonging to the productions the user wishes to block. On the other hand, the user can import a control file. If the user imports a control file, the checkboxes are set accordingly.

3.11.3 Reduce Menu

In the *Reduce* menu that can be seen in figure 3.9 the user can reduce the syntax according to his selections. The syntax is reduced and the reduced syntax is displayed afterwards. The rules maintain the same order as in the original TPTP syntax.

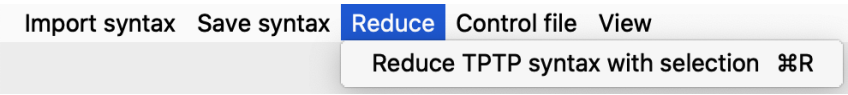

Figure 3.9: Reduce menu

3.11.4 Save syntax Menu

In the *Save syntax* menu that is shown in figure 3.10 the user can reduce the syntax based on his GUI selection or an imported control file. The difference between the reduce option and the save menu is that the save menu saves the generated reduced syntax to local storage and does not display it. If the user wishes that the $< comments >$, which is necessary for using the automated parser generator (see chapter 3.10.2), production is including in the reduced syntax the user has the option to save the reduced syntax with the $< comment >$ production.

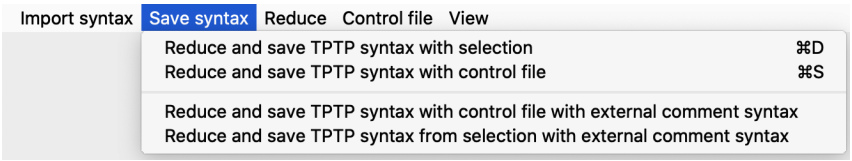

Figure 3.10: Save menu

3.11.5 Control file Menu

The *Control file* menu shown in figure 3.11 gives the user the option to import a control file instead of blocking productions manually. If the user imports a control file, the checkboxes are set accordingly. The user also has the option of generating a control file based on his GUI selection. This is particularly helpful if the user uses the same blocked productions multiple times. This way the user can use the import control file option later instead of selecting the blocked productions in the GUI.

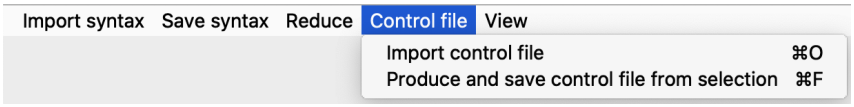

Figure 3.11: Control file menu

3.12 Command-line interface

In addition to the GUI the software tool offers a command-line interface to provide the means for convenient automation of sub-syntax extraction. It takes a TPTP syntax file and a control file as input and outputs the resulting sub-syntax. It is implemented using the Python module argparse[19], which provides a framework for creating command-line argument parsers.
The table below provides an overview about the implemented command-line arguments. The syntax file location and the control file location have to be specified. Specifying an output path and filename is optional, by default the output filename will be *output.txt*. The $-ex$ flag enables additional output of the comment syntax (see section 3.10.2). Additionally, the help description can be opened by using the *-h* option.

Table 3.2: Command-line interface parameters

Name	Short form	Default	Description
--grammar	-g	None	TPTP syntax file path and filename
--control	-c	None	Control file path and filename
--output	-o	output.txt	Output file path and filename (optional)
--external_comment	-ex	False	Flag to include external comment syntax (optional)

The implementation of the command-line interface is described in section 4.9. More complex actions like control file generation can be done more comfortably by using GUI which is why this is not a feature of the command-line interface.
The design decision to separate the GUI from the command-line interface (see section 3.3.1) has the advantage that if a user chooses to solely use the command-line interface, it is not necessary to install the library used in GUI implementation (PyQt 5).

3.13 Counting the number of rules in the TPTP syntax

Rules and productions are counted to compare syntax sizes. The results are described in chapter 5.

3.13.1 Counting rules in a grammar graph

Rules are counted by counting nodes in the grammar graph. Productions are counted by traversing the nodes in the grammar graph and counting the number of elements in each productions list.
In addition to the total number of rules and productions, the number of grammar rules and productions are filtered and counted separately. The total number of rules and productions and the number of grammar rules and productions is printed to the command-line after the import of a syntax and after a sub-syntax generation. Since the grammar graph only contains reachable and terminating symbols, the

total number of rules and productions in the original TPTP syntax version 7.3.0 cannot be counted, since some symbols are not reachable. To compare generated sub-syntaxes to the original TPTP a script has been developed, which is described in the following.

3.13.2 Counting rules in the original TPTP syntax

To count rules of the original TPTP syntax (including non-reachable and nonterminating symbols) a script was developed. It uses the parser to parse the original TPTP syntax which results in a grammar list. This grammar list is iterated and rules and productions are counted in order to get the total numbers. Also, grammar rules and production are counted separate too.
Counting rules based on the full TPTP syntax gives a better overview when comparing syntax sizes.
The script can be used independently of Synplifier

4 Implementation

Based on the previous introduction of the concept of Synplifier, this chapter describes the implementation of Synplifier in detail. The chapter covers lexing and parsing the TPTP syntax as well as building the grammar graph, extracting a sub-syntax and maintaining comments. Furthermore, the implementation of the GUI and the command-line interface is introduced including needed input and output functionalities.

4.1 Lexer

As mentioned in chapter 3.4 the PLY package uses the definition of tokens in form of regular expressions to generate the lexer. The following sections presents defined tokens and the regular expressions that describe these tokens.

In order to build the lexer, the function *lex.lex*() is used. After the lexer has been created, the function *lex.token*() returns matched tokens. The function *lex.token*() is called until it returns a token of the type *None* that marks the end of the input stream.

4.1.1 Token

The names of the tokens indicate what they represent. All tokens that have been defined are combined to a token list. A token list is always required when writing a lexer using PLY [17]. The list of tokens is also used by the parser to identify terminal symbols [17]. Listing 4.1 shows the token list of the implemented lexer. The tokens featured in the token list are described in the following.

```
1 # List of token names
2 tokens = (
3     'LGRAMMAR_EXPRESSION',
4     'LTOKEN_EXPRESSION',
5     'LSTRICT_EXPRESSION',
```

```
    'LMACRO_EXPRESSION',
    'NT_SYMBOL',
    'COMMENT',
    'OPEN_PARENTHESIS',
    'CLOSE_PARENTHESIS',
    'OPEN_SQUARE_BRACKET',
    'CLOSE_SQUARE_BRACKET',
    'ALTERNATIVE_SYMBOL',
    'REPETITION_SYMBOL',
    'T_SYMBOL'
)
```

Listing 4.1: Lexer tokens

4.1.2 Regular expressions

Tokens are specified by regular expressions. Regular expressions, that define tokens, have to be defined in a Python function in order for PLY to recognize these tokens. The name of the function is the name of the corresponding token in the token list with the prefix t_. Inside a function, it is possible to modify the value of the generated LexToken object and then return it. If not modified, the value of the object is not modified and remains to be the actual text that has been matched. The regular expression is specified as a string in the beginning of a function. An extract of all functions can be seen in listing 4.2. The function for the token $COMMENT$ consists of the regular expression that matches a comment which has already been described in chapter 3.4 and returns the LexToken object unmodified. The functions of $OPEN\_SQUARE\_BRACKET$, NT_SYMBOL and T_SYMBOL (and the other tokens in the tokens list) are implemented similar to $COMMENT$. The function of $LGRAMMAR\_EXPRESSION$ and the functions of the other expressions modify the value of the LexToken object. Instead of the value being the actual text matched, first the last three characters are removed. These characters represent the production symbol which indicates the rule type (for an overview of the present rule types see table 3.1). The rule type is already known because depending on the rule type of a production different tokens are matched. Then the Python method $rstrip()$ removes all trailing white space characters from the string value. Thus, the function returns a LexToken object with the value of the text belonging to the nonterminal symbol that the expression represents.

```
def t_COMMENT(self, t):
```

```
    r'^%.*$'
    return t

def t_OPEN_SQUARE_BRACKET(self, t):
    r'\['
    return t

def t_LGRAMMAR_EXPRESSION(self, t):
    r'<\w+>[\s]*::='
    t.value = t.value[:-3]
    t.value = t.value.rstrip()
    return t

def t_NT_SYMBOL(self, t):
    r'<\w+>'
    return t

def t_T_SYMBOL(self, t):
    r'[$\'\\\.,a-zA-Z\_0-9\0-9-\-<>&@!:{}~?^+=/"!^^/@/+-/%;][a-zA-Z\_0-9\-/"!?/@
        /+-/*/%;\->&+=$\'\\\.,]*'
    return t
)
```

Listing 4.2: Lexer regular expressions

The function *t_ignore* is used for declaring characters that should be ignored in the input stream. In this project, white spaces and new lines are ignored (see chapter 3.4).

The function *t_error* is called when an error occurs during lexing. The defined error function shown in listing 4.3 prints the value of the illegal character and skips the character.

```
def t_error(self, t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
)
```

Listing 4.3: Lexer error function

4.2 Parser

The parser takes the matched tokens from the lexer and constructs a grammar list (see chapter 3.5) based on a specified context free grammar. The rules of the context

free grammar are defined by Python functions. The docstrings of the functions contain a specification of the rules. This kind of function accepts one argument which is a list of grammar symbols and contains semantic actions. The nonterminal symbol that is on the left side of the rule(s) is stored in the first element of the input.
An example for a defined function can be seen in listing 4.4. This function defines the rule that has the nonterminal symbol *productions_list* on the left side. The docstring of the function contains the specification of the rule and the code inside the function determines the actions that are taken once the function is called. If the input has the length 2, meaning the input list consists of two entries, a new *PRODUCTIONS_LIST* is created with $p[1]$ and is saved in $p[0]$. If the length of the input equals four, a *productions_list* already exists and is expanded by a new productions alternative. This is done by appending $p[3]$ to the existing list that is stored in $p[1]$. The result is again being stored in $p[0]$.

```
1  def p_productions_list(self, p):
2      """
3      productions_list : production
4                       | productions_list ALTERNATIVE_SYMBOL production
5      """
6      if len(p) == 2:
7        p[0] = PRODUCTIONS_LIST([p[1]])
8      elif len(p) == 4:
9          p[1].list.append(p[3])
10         p[0] = p[1]
```

Listing 4.4: Parser function of productions list

The function that is defined first in the Python file determines the start symbol to which the grammar should be reduced to. The parser stops when no more input exists and the start symbol has been reached.

4.2.1 Nonterminal symbols

The nonterminal symbols of the defined context-free grammar are described in the following.

A *T_SYMBOL _PRODUCTION* consists of the token *T_SYMBOL* (see chapter 4.1.1) and additionally a production property. The production property can take two values and denotes whether a *T_SYMBOL* is embedded in square brackets

or not. In the production rules a *T_SYMBOL* token, either embedded in square brackets or without brackets, can be derived from a *T_SYMBOL _PRODUCTION*. Asterisk and vertical bar tokens embedded in square brackets can be derived from a *T_SYMBOL _PRODUCTION* production as well. The *T_SYMBOL _PRODUCTION* is necessary because usually square brackets denote that the term within the brackets is optional which is not the case with terminal symbols. Square brackets embedding terminal symbols denote that the symbol within the brackets is terminal. For example an asterisk embedded in square brackets refers to the terminal symbol. An asterisk not embedded in square bracket corresponds to the production property *optional* (see figure 4.4).

A *RULE* consists of an expression token (see chapter 4.1.1), a production list and a position. The position denotes at which position in the TPTP syntax the rule was listed. This information is needed to maintain the original order of the rules when printing the reduced syntax. For each rule type there is a data type. This means that *GRAMMAR*, *TOKEN*, *STRICT* and *MACRO_RULE* nonterminal symbols are introduced. They inherit from the base class *RULE* which can be seen in figure 4.1. Listing 4.5 contains an example of a line in a TPTP syntax file that is represented by the *GRAMMAR_RULE* data type. The nonterminal symbol name which is produced is $<tff\_formula>$. The production list consists of two productions, as can be seen in the listing.

```
1 <tff_formula>             ::= <tff_logic_formula> | <tff_atom_typing
```

Listing 4.5: Grammar expression

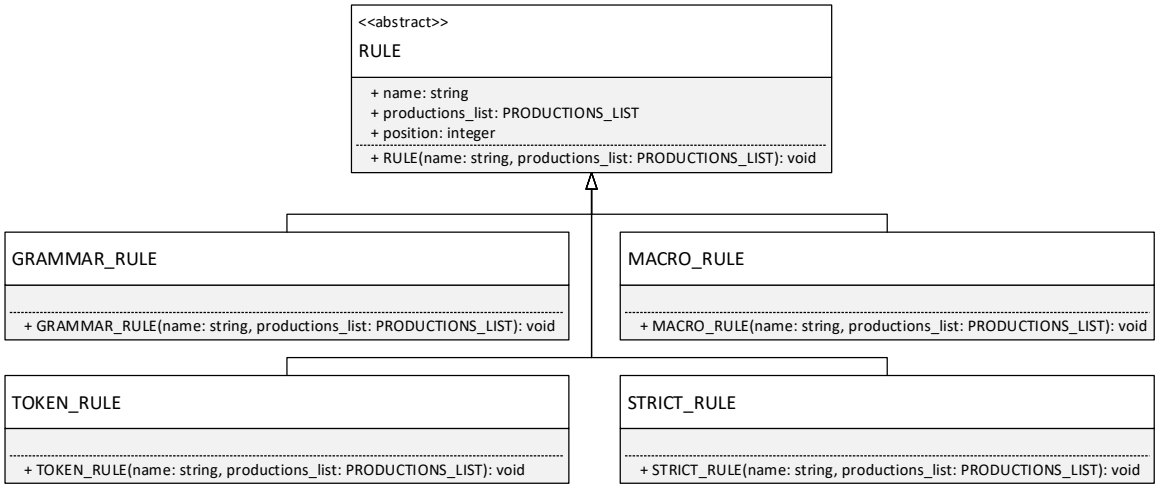

Figure 4.1: Rule data type UML class diagram

A *COMMENT_BLOCK* is a list of consecutive comment lines. Comment lines can also be extended after initialising a comment block. Figure 4.2 shows the UML denotation of a comment block.

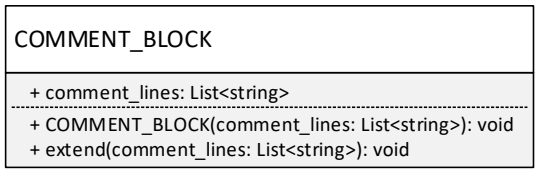

Figure 4.2: Comment block data type UML class diagram

A *T_SYMBOL_PRODUCTION* or a *NT_SYMBOL* is a *PRODUCTION_ELEMENT*. Therefore it has an instance of *SYMBOL* as an attribute which can be seen in figure 4.3. Additionally, a production element has a production property that is shown in figure 4.4. The production property can take one of three values and denotes whether a production is optional, can be repeated multiple times or does not have any special property. In the original TPTP syntax file, the property is represented by square brackets or the repetition symbol.

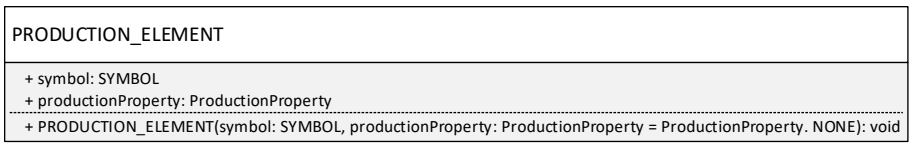

Figure 4.3: Production element data type UML class diagram

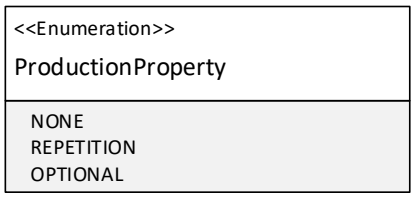

Figure 4.4: Production property data type UML class diagram

A *PRODUCTION* is one of the production alternatives of a rule. As can be seen in figure 4.5 it consists of a list of *PRODUCTION_ELEMENT*s and/or nested *PRODUCTION*s. It also has a *ProductionProperty* as attribute.

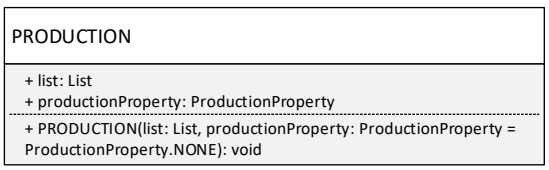

Figure 4.5: Production data type UML class diagram

A *PRODUCTIONS_LIST* contains a list of productions where each production is one alternative in the description of a rule.

The *XOR_PRODUCTIONS_LIST* represents a *PRODUCTIONS_LIST* with multiple alternatives enclosed by parentheses. It contains a list of the alternate

productions. Figure 4.6 shows the UML denotation of a *PRODUCTIONS_LIST* and a *XOR_PRODUCTIONS_LIST* which inherits from the *PRODUCTION_LIST* class.

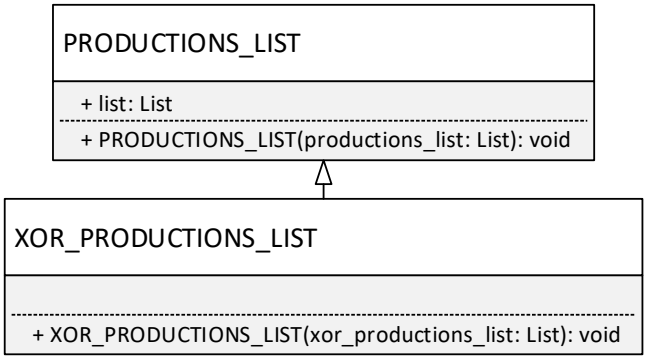

Figure 4.6: Productions list data type UML class diagram

The *GRAMMAR_LIST* class shown in figure 4.7 is the top level data structure. It contains a list of all rules that are in the TPTP syntax file. This includes any type of rules (grammar, token, strict and macro) and comment blocks.

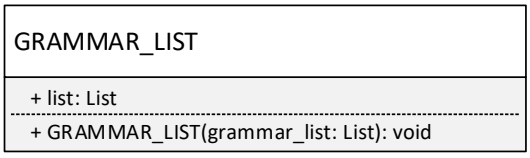

Figure 4.7: Grammar list data type UML class diagram

Example

Listing 4.6 shows the production rule of the nonterminal $< tfxtuple >$ as well as the tokens that have been generated by the lexer. Using this single production, the functionality of the parser should be exemplified.

```
1 <tfx_tuple>            ::= [] | [<tff_arguments>]
2 is made of tokens:
3 l grammar expression open square bracket close square bracket alternative symbol open
      square bracket nt symbol close square bracket
```

Listing 4.6: Production element

The resulting output of the parser can be seen in figure 4.8. The tokens have been reduced to the start symbol grammar list and are displayed in a form of a tree. Internally, the grammar list is stored as a nested list.

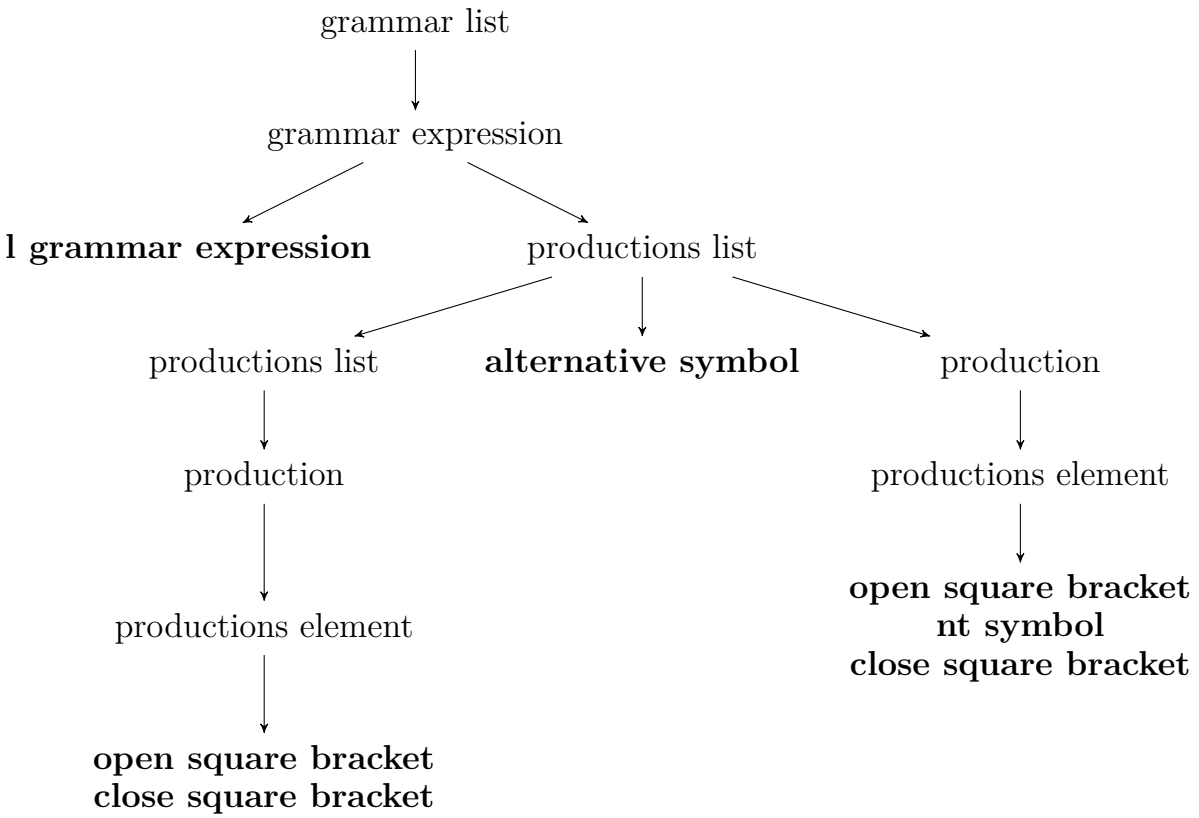

Figure 4.8: Parsing example

4.2.2 Building the parser

To build the parser, the PLY function *yacc.yacc*() is called that constructs a parser from a specified grammar. To run the generated parser *parse*() must be called.

The output of the parser is further modified to include a numbering of rules and to handle square brackets correctly (see chapter 3.5.2). Rules are numbered by assigning the position of a rule within the grammar list as its position. Comment blocks are skipped.
The disambiguation of square brackets replaces the *ProductionProperty Optional* by the terminal symbols open and closing square bracket for *GRAMMAR* and *STRICT* expressions. By doing so, the productions in the productions list of a rule are iterated. If the production has the production property *Optional* the production property is changed to *None* and square brackets are inserted in the production. This is done for productions as well as production elements that have the production property *Optional* inside a production.

4.3 Graph generation

The class *TPTPGraphBuilder* that can be seen in figure 4.9 has an instance of the Parser that has been introduced in chapter 4.2. With the output of the parser, which is a grammar list, the graph builder builds a nodes dictionary and a graph. This is done by calling the *run* method that calls *init_tree* for building the graph. The methods assigned to a) in figure 4.9 are used for building a graph and are in the following. The graph builder is also responsible for reducing the TPTP syntax, the methods used for that are assigned to b) and introduced in chapter 4.5. Moreover, with the graph builder, comments are assigned to rules and nodes. Belonging methods are assigned to c) and described in chapter 4.4.

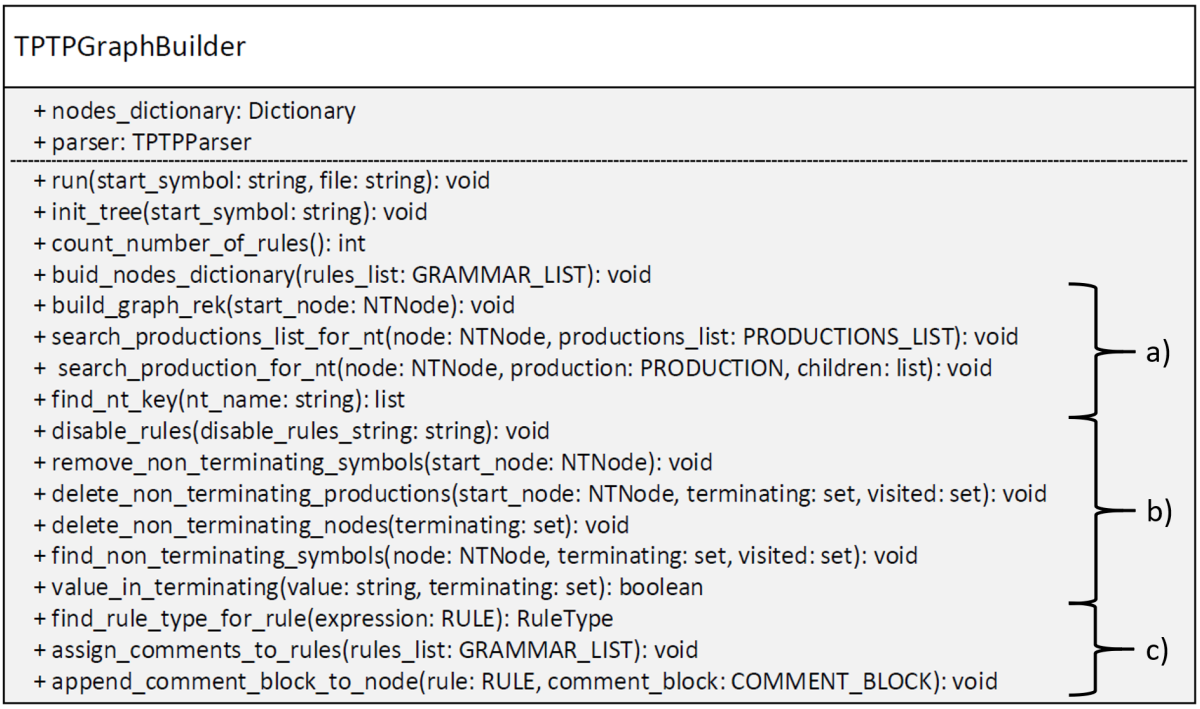

Figure 4.9: Graphbuilder UML diagram

Before generating a graph, a node dictionary consisting of nodes is built.
Figure 4.10 shows the class *NTNode* that has already been introduced in chapter 3.6. The class has three class methods: *Add_children* adds a list of children to the existing nodes children list. *Extend_comment_block* appends a comment block to the existing comment block of a node. The *str* method is used for exporting the grammar.

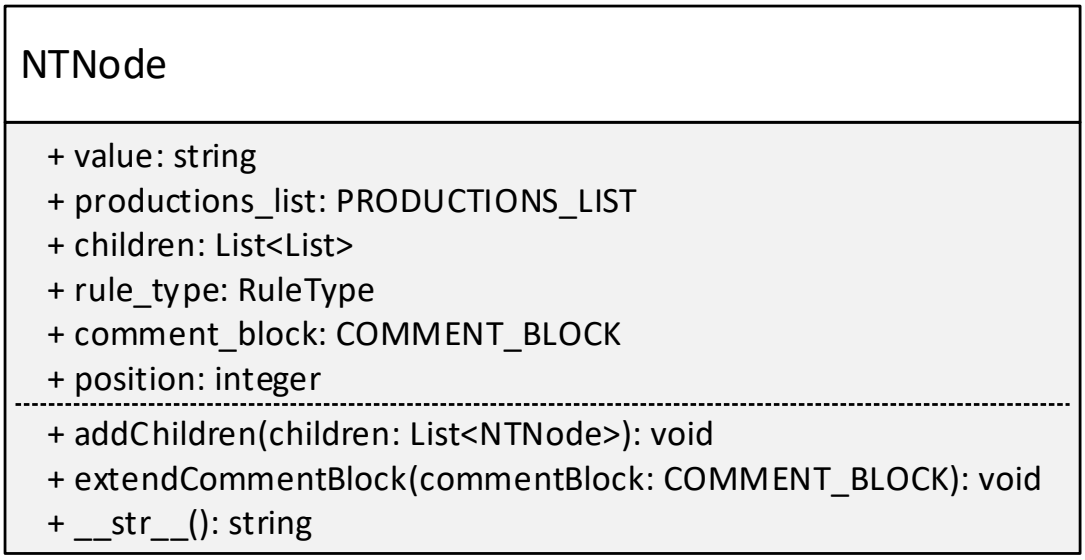

Figure 4.10: NTNode UML diagram

The rule type of a node is either grammar, token, strict or macro and is implemented as an enumeration which can be seen in figure 4.11.

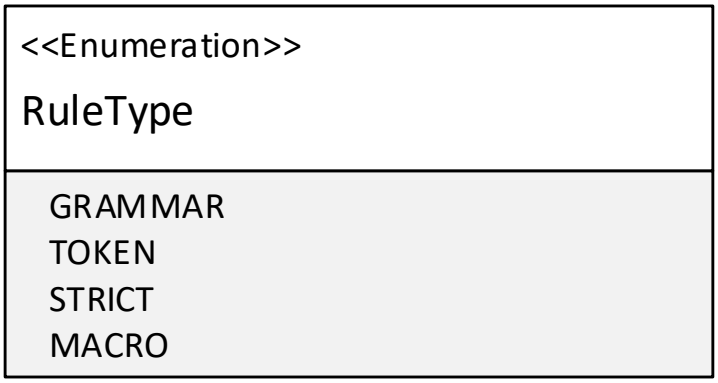

Figure 4.11: RuleType UML diagram

Algorithm 1 shows how the nodes dictionary is build. Iterating the rules list, a new dictionary item is created that consists of the key rule name and rule type and the

NTNode as value. This is only the case if the rule is not a comment block. Listing 4.7 shows that multiple rules can exist that correspond to the same node.

```
<defined_proposition>  :== <defined_predicate>
<defined_proposition>  :== $true | $false
<defined_predicate>    :== <atomic_defined_word>
<defined_predicate>    :== $distinct |
                           $less | $lesseq | $greater | $greatereq |
                           $is_int | $is_rat |
                           $box_P | $box_i | $box_int | $box |
                           $dia_P | $dia_i | $dia_int | $dia
```

Listing 4.7: Multiple rules with the same left-hand side of the rule

If that would not have been considered, the second rule would create a new node and replace the existing one in the dictionary because their dictionary keys are the same. To not overwrite the node, the algorithm checks whether the key already exists in the dictionary. If the key already exists, the productions lists of the two rules are merged into one productions list that is appended to the node. After iterating, comments are assigned (see section 4.4).

Algorithm 1 Graph Generation Algorithm: buildNodesDictionary

Input: rules_list

```
for all rule in rules_list do
    if rule is not a comment block then
        rule_type = this.find_rule_type_for_rule(rule)
        node_key = Node_Key(rule.name, rule_type)
        if node_key is not in this.nodes_dictionary then
            node = NTNode(rule.name, rule.productions_list, rule_type, None,
rule.position)
            this.nodes_dictionary.update(node_key:node)
        else
            node = this.nodes_dictionary[node_key]
            node.productions_list.list.extend(rule.productions_list.list)
        end if
    end if
end for
Assign comments to rules
```

To generate the graph of a given grammar three algorithms are needed that will be explained in the following.

The algorithm *buildGraphRek* calls the function *searchProductionsListForNT* that appends children of a node to the nodes list of children. The algorithm is first called with the start symbol. After the children of a node have been appended to the node, for every child the algorithm is called resulting in appending their children to their children's list.

Algorithm 2 Graph Generation Algorithm: buildGraphRek

```
Input: node
searchProductionsListForNT(node, node.productionsList)
if node has children then
    for all children do
        buildGraphRek(child)
    end for
end if
```

The right side of a production rule is stored in a productions list. For identifying the nonterminal or terminal symbols in the productions lists, a loop iterates all elements of the productions list. Each element is a production and calls the function *searchProductionForNT*. This function identifies the children of the given element which are then appended to the node.

Algorithm 3 Algorithm for extracting productions from productions list: searchProductionsListForNT

```
Input: node, productionsList
for all elements in productionsList do
    children = new empty list
    searchProductionForNT(node, element in productionsList, children)
    append children to node
end for
```

For identifying the nonterminal symbols, it is checked if the production is a nested production and if so, the same function is called again. If the production is a

XOR production list the function *searchProductionsListForNT* is called to break down the productions list. If the production element is a nonterminal symbol, the element is searched in the node dictionary to get the node where the element is on the left side. This element is then appended to a list of children. It is possible that an element appears multiple times on the left side if it is presented by multiple expressions. In this case each element is appended to the list of children.

Algorithm 4 Algorithm for appending children to node: searchProductionForNT

Input: node, productionsElement, children

```
1: for all elements in productionsElementList do
2:     if element is a production then
3:         searchProductionForNT(node, element, children)
4:     else if element is a XOR productions list then
5:         searchProductionsListForNT
6:     else if element is a nonterminal symbol then
7:         find element(s) in node dictionary
8:         append element(s) to children
9:     end if
10: end for
```

4.4 Maintaining comments

After building the nodes dictionary and the grammar graph, comments of the original TPTP syntax are identified and assigned to the corresponding nodes. This section describes the implementation of associating comments with nodes in order to maintain comments in a generated sub-syntax. The implementation follows the concept outlined in section 3.8. Comments are associated after the grammar graph has been generated. They are associated to the node, that represents the rule they belong to. For that, the *NTNode* class has a *COMMENT_BLOCK* attribute (see *NTNode* UML diagram in figure 4.10).
Associating comments with nodes has the advantage, that when the grammar graph is reduced, only the comments associated with the rules that are part of the sub-syntax are maintained and can also be printed when outputting the sub-syntax.

4.4.1 Associating comments with grammar graph nodes

Algorithm 5 describes the procedure of matching comments to nodes. The algorithm follows the concept outlined in section 3.8 and is used within the *TPTPGraphBuilder* after generating the grammar graph.
The input is the *grammar_list*, that the parser created from parsing the TPTP syntax file. The *grammar_list* contains rules and comment blocks in the order they appeared in the original TPTP syntax file. The *grammar_list* is iterated (line 1). If the current element of the *rules_list* is a *COMMENT_BLOCK*, it is split by top of page lines (line 3). The procedure of splitting a comment block by top of page lines is described in section 4.4.2. It returns a list of comment block objects.
In the following if-clause it is checked, how many comment block objects are returned.
In the first case of one comment block, this comment block should be associated with the following rule by default. If that is not possible, because the comment block is at the end of the syntax file, it should be associated with the rule before. In line 5 it is checked whether the comment block is not at the end of the syntax file. To associate the comment block with the following rule, the next rule (rule after the comment block) is obtained by indexing the next element of the *rules_list*. Then, the comment block is appended to the node, specified by the rule, using the *append_comments_to_node* method. This method is described in the following subsection.
If the comment block is at the end of the syntax file, it has to be associated with the rule before. This case is handled in the else branch (line 8). The procedure of adding the comment block to the desired node is essentially the same as described before. The difference is that the previous rule is addressed (line 9).
If splitting the original comment block by top of page lines results in two comment blocks the first is associated with the rule before and the second with the rule after on condition that the original comment block was neither at the end nor the beginning of the syntax file (lines 14, 15, 21, 22).
If the original comment block was at the beginning of the syntax file, the first comment block is appended to the rule after too (lines 17, 18).
If the original comment block was at the end of the syntax file, the second comment block is associated with the rule before.
The case of two or more top of page lines does not occur in the TPTP syntax version 7.3.0. and is not included in the pseudo-code.

This hypothetical case is handled by using the same procedure as in the case of two comment blocks but treating all comment blocks after the second comment block like the second. This means that all comment blocks from the second on would be associated with the rule after, except the case that the original comment block was at the end of the syntax file. In that case all comment blocks would be appended to the rule before.

The algorithm relies on the *rules_list* never having two or more comment blocks in a row (meaning they are always separated by at least one rule). This is ensured by the parser, because all consecutive comment lines in the syntax file are included into one comment block. Therefore consecutive comment block objects do not occur in the *rules_list*.

Algorithm 5 Assign comments to rules

```
Input: grammar_list
 1: for all index, expression in rules_list.list do
 2:   if expression is COMMENT_BLOCK then
 3:     comment_block_list = expression.split_comment_block_by_top_of_page()
 4:     if length(comment_block_list) == 1 then
 5:       if index < length(rules_list.list) - 1 then
 6:         next_rule = rules_list.list[index + 1]
 7:         this.append_comments_to_node(next_rule, comment_block_list[0])
 8:       else
 9:         previous_rule = rules_list.list[index - 1]
10:         this.append_comments_to_node(previous_rule, comment_block_list[0])
11:       end if
12:     else if length(comment_block_list) == 2 then
13:       if index != 0 then
14:         previous_rule = rules_list.list[index - 1]
15:         this.append_comments_to_node(previous_rule, comment_block_list[0])
16:       else
17:         next_rule = rules_list.list[index + 1]
18:         this.append_comments_to_node(next_rule, comment_block_list[0])
19:       end if
20:       if index < length(rules_list.list) - 1) then
21:         next_rule = rules_list.list[index + 1]
22:         this.append_comments_to_node(next_rule, comment_block_list[1])
23:       else
24:         previous_rule = rules_list.list[index - 1]
25:         this.append_comments_to_node(previous_rule, comment_block_list[1])
26:       end if
27:     else if length(comment_block_list) > 2 then
28:       %excluded
29:     end if
30:   end if
31: end for
```

Appending a comment block to a node

In Algorithm 6 it is described, how a comment block is appended to a node from the grammar graph, using information from the original rule created by the parser to address the node.

To associate the *COMMENT_BLOCK* with the node representing the rule, the node needs to be accessed. It is more efficient to use the created *nodes_dictionary* to access the desired nodes than to traverse the grammar graph for each wanted node.

In the *nodes_dictionary*, the key for each node consists of the nonterminal symbol name and the *RuleType* is stored(see section 4.3).
The rule type of the input rule is indicated by the type of the *rule* object (see class diagram in figure 4.11). To index the node in the *nodes_dictionary* the *RuleType* corresponding to the type of the rule object is needed. The *find_rule_type_for_rule* method determines and returns the *RuleType* corresponding to the *rule* object type. Using the determined *RuleType* and the nonterminal name, which is stored in the *rule*, the key for the *Nodes_Dictionary* can be constructed (line 8). Then, the node, to which the *COMMENT_BLOCK* should be added is addressed in the *Nodes_Dictionary* and the *extend_comment_block* method of the node called, with the *COMMENT_BLOCK* as argument. The *extend_comment_block* method is described in algorithm 7. It appends the *COMMENT_BLOCK* to the *COMMENT_BLOCK* attribute of the node object.

Algorithm 6 Append comments to node

Input: rule, comment_block
1: rule_type = find_rule_type_for_rule(rule)
2: node_key = Node_Key(rule.name,rule_type)
3: this.nodes_dictionary[node_key].extend_comment_block(comment_block)

Extending a comment block of a node

Algorithm 7 shows the procedure that is used in the *extend_comment_block* method of the *NTNode* class. If no *comment_block* has not been associated with the node before, the input *comment_block* is associated with the node. If the node has already have a *comment_block*, this *comment_block* is extended, with the comments lines from the input *comment_block*.
This procedure is needed because it is possible that multiple comment blocks, that have been generated on splitting comment blocks, have to be associated with the same node.

Algorithm 7 Extend node comment block

Input: comment_block

```
1: if this.comment_block is None then,
2:   this.comment_block = comment_block
3: else
4:   this.comment_block.extend(comment_block.comment_lines)
5: end if
```

4.4.2 Splitting comment blocks

The flow chart in figure 4.12 shows how a comment block is split into multiple comment blocks separating it by top of page lines.
First the indexes of the top of page lines are collected in a list (using method of *COMMENT_BLOCK*). After that, the *comment_block_list*, which will contain the generated *COMMENT_BLOCK*s, is initialized. If the list, storing top of page line indexes, is empty, the original comment block is added to the *comment_block_list*, because the comment block does not need to be split into multiple comment blocks because no top of page line is present. Then the *comment_block_list* is returned.
If the list of top of page line indexes is not empty, the first top of page line index is retrieved from the top of page indexes list. It is also checked whether the first top of page index is unequal zero.
If that is the case, the first top of page index is not the first line of the original comment block, and a comment block can be constructed from the line(s) before. In this case the first comment block can be constructed from the comment lines of the original comment block from index zero to the first top of page index. This comment block is added to the *comment_block_list*.
Then the *index* variable, which represents an index on the *top_of_page_indexes* list, is initialized. The *index* is used to access the corresponding element of the *top_of_page_indexes* list.
Then it is checked if top of page line that corresponds to the top of page line index is the last line of the original comment block. In that case, no further comment block needs to be added to the *comment_block_list* and the *comment_block_list* is returned.
Otherwise the start index of the next comment block is computed as the current *top_of_page_index* incremented by one (line after the top of page line). There are two cases that have to be considered when determining the end index of the

comment block. Either there is another top of page line, in this case the comment block ends before that, or the currently indexed top of page line was the last one, in that case the end index is the end of the original comment block. Then the new commend block is constructed with the before determined *start* and *end*. This comment block is added to the *comment_block_list*. If the *index* is at the end of the *top_of_page_index* list, the *comment_block_list* is returned. If not, the *index* variable is incremented and the process is repeated with the next top of page index.

Figure 4.12: Split comment block by top of page line flow chart

4.5 Extraction of a sub-syntax

The following sections describe the algorithms used for the extraction of a sub-syntax based on the concept outlined in section 3.9.

4.5.1 Removal of blocked productions

Algorithm 8 takes the text of the control file as an input argument (control file format description can be found in section 3.7) and removes the productions that should be blocked from the nodes. The control file string is split into a list of the lines (line 1). The first line of the control file string, which describes the start symbol, is deleted from the list (line 2) because this information is not needed at this step.
Then the following lines are iterated:
For each line the content is split by the comma symbol, which produces a list of strings. The first element of that list is the nonterminal symbol name of the symbol from which productions should be disabled. The second element is the corresponding rule type symbol. The nonterminal symbol name and the rule type, that is determined from the rules symbol (line 7), are stored in separate variables. Then these elements are removed from the list and the remaining elements are converted to integers. They are the indexes of the productions that should be removed. The elements are ordered descending (line 10). Ordering the indexes in that order has the advantage that, when the list of indexes is iterated to delete productions, the indexes of other productions that should be deleted are not changed by the deletion of productions before. To delete the productions from the specified node the node object has to be accessed. This is done by addressing the node in the *nodes_dictionary* with its key consisting of the nonterminal symbol name and the rule type. From the node, in a loop, the elements of the *productions_list* and the *children* list corresponding to the indexes in the control file are deleted.

Algorithm 8 Removing blocked productions

```
Input: control_string
 1: lines = control_string.splitlines()
 2: delete lines[0]
 3: for all line in lines do
 4:     data = line.splitBy(",")
 5:     nonterminal_name = data[0]
 6:     rule_symbol = data[1]
 7:     rule_type = determineRuleType(rule_symbol)
 8:     delete data[0:2]
 9:     data = parseInteger(data)
10:     data.sortReverse()
11:     node = this.nodes_dictionary.get(Node_Key(nonterminal_name,
    rule_type))
12:     for all index in data do
13:         delete node.productions_list.list[index]
14:         delete node.children[index]
15:     end for
16: end for
```

4.5.2 Determination of the remaining terminating symbols

After the desired productions have been deleted from the grammar graph, the next step is to remove the nonterminating symbols from the grammar graph. Algorithm 9 shows the procedure of removing nonterminating symbols. First it has to be determined which symbols are terminating and which are nonterminating.
On the one hand, starting from the terminal symbols all nonterminal symbols that derive terminal symbols could be found. However, the graph data structure can only be traversed top down not bottom up. Therefore starting at the start node, the graph is traversed and terminating symbols are found. A set of terminating nodes and a temporary set of terminating nodes are initialized. In a while loop the recursive algorithm *find_non_terminating_symbols(start_node, temp_terminating, visited)* finds terminating nodes with the start_node, an initialized set of known terminating symbols and a set of already visited nodes. This algorithm is called with the updated set of terminating symbols until the set does not differ from the run before. When that is the case, all terminating nodes have been found.
After all terminating nodes have been found, the productions that contain nonter-

minating symbols are deleted from the nodes and after that nodes representing nonterminating symbols are removed.

In the following parts of the section, first the *find_non_terminating_symbols* algorithm is described. Then the *delete_non_terminating_productions* algorithm and after that the *delete_non_terminating_nodes* algorithm are outlined.

Algorithm 9 Removing non terminating symbols

```
Input: start_node
 1: terminating = new set()
 2: temp_terminating = new set()
 3: while True do
 4:     visited = new set()
 5:     this.find_non_terminating_symbols(start_node, temp_terminating, vis-
    ited)
 6:     if terminating == temp_terminating then
 7:         break
 8:     else
 9:         terminating = temp_terminating
10:     end if
11: end while
12: visited = new set()
13: delete_non_terminating_productions(start_node, terminating, visited)
14: delete_non_terminating_nodes(terminating)
```

Algorithm 10 recursively determines whether a node can be identified as terminating. The input is a set of already known terminating nodes and nodes that have already been visited. If the node has been not been visited yet, it is added to the visited set. If it has been visited, the method ends to prevent infinite recursion. If it has not been visited yet, the *children* list of the node is iterated for each children list in *node.children*. If a *children_list* is empty, this means that in a production there are no nonterminal symbols and the production only consists of terminal symbols. If that is the case the node is terminating.
If it is not the case, the node is terminating if all nodes in the *children_list* represent a terminating nonterminal symbol. To check that, the algorithm is called recursively for all nodes in the *children_list*. If a nonterminal symbol has multiple rule types, it is represented by multiple nodes. Also, if a nonterminal symbol with multiple rule

types is in featured in a production, all nodes representing that nonterminal symbol are in the children list corresponding to this production. Only one of the nodes needs to be terminating in order for the nonterminal symbol to be terminating. This is considered in the procedure of checking if a node represents a terminating symbol (line 11), which is described in figure 4.13. If every node in the *children_list* represents a terminating symbol, the node is added to the known terminating nodes.

Algorithm 10 Find non terminating symbols

```
Input: node, terminating_set, visited_set
 1: node_key = Node_Key(node.value, node.rule_type)
 2: if node_key not in visited_set then
 3:     visited_set.add(node_key)
 4:     for all children_list in node.children do
 5:         if len(children_list) == 0 then
 6:             terminating_set.add(node_key)
 7:         else
 8:             terminating_flag = True
 9:             for all child in children_list do
10:                 find_non_terminating_symbols(child, terminating_set, visited_set)
11:                 if not value_in_terminating(child.value, terminating_set) then
12:                     terminating_flag = False
13:                 end if
14:             end for
15:             if terminating_flag then
16:                 terminating_set.add(node_key)
17:             end if
18:         end if
19:     end for
20: end if
```

The flow diagram in figure 4.13 shows the procedure of checking whether a nonterminal symbol, described by its name, is in the set that contains information on which nodes are known to be terminating. Theoretically there can be up to four nodes representing a nonterminal symbol, because it can have up to four rule type (and a node represents the combination of nonterminal symbol name and rule type). However, according to the TPTP syntax only the rule types grammar and strict

can belong to one nonterminal symbol.
In the *terminating_set* the *NodeKey*s of known terminating nodes are stored. A nonterminal symbol is known to be terminating when any node representing that nonterminal symbol is known to be terminating. If a *NodeKey* of any rule type and with the nonterminal name is in the *terminating_set* then the symbol is terminating and True is returned, otherwise False is returned.

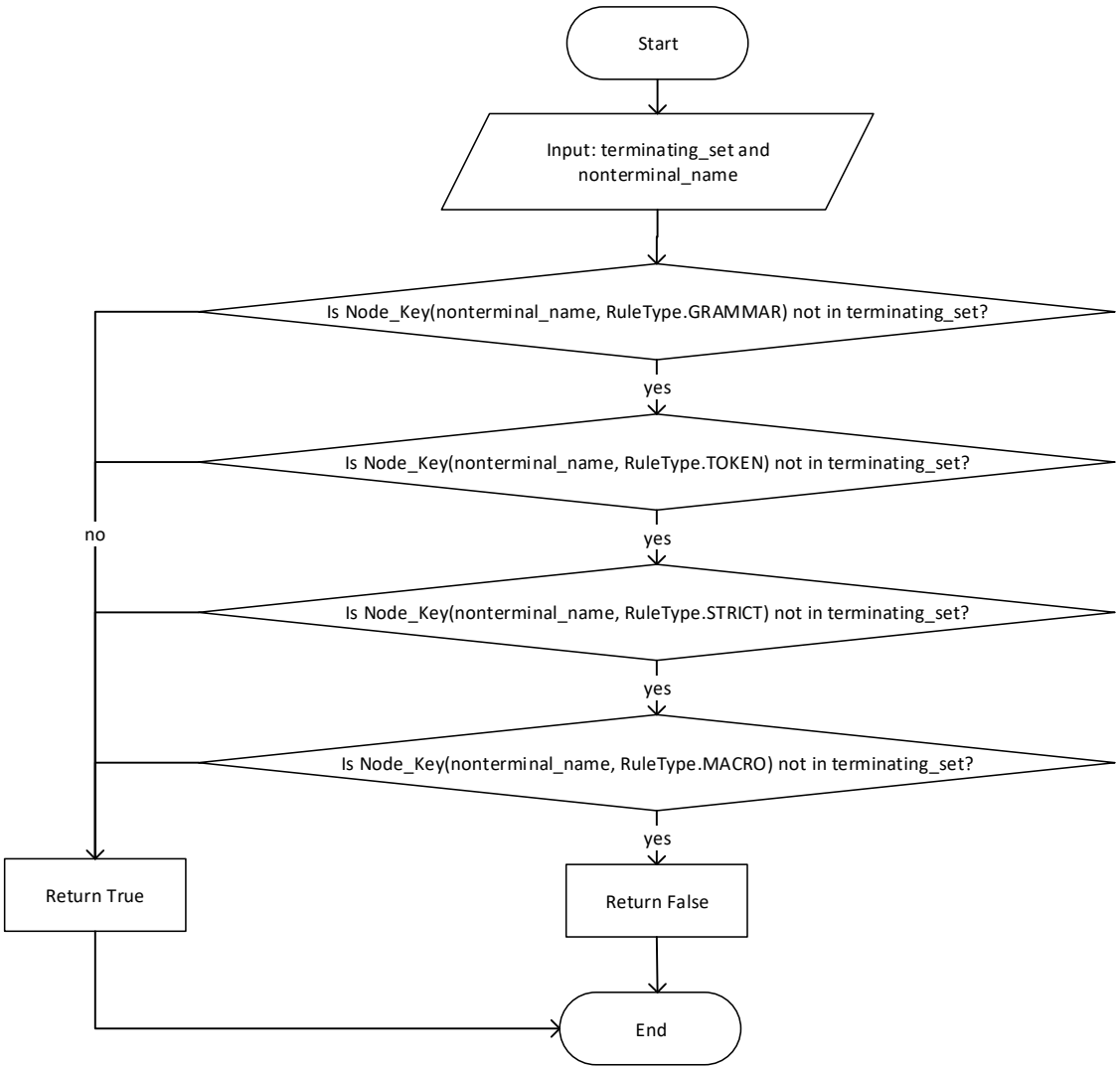

Figure 4.13: Check if node value is in terminating set

After the terminating symbols have been determined, productions that contain a nonterminating symbol have to be removed from the nodes of the grammar

graph. Only productions and not the node itself is deleted because a node might also have productions that terminate. Therefore only the productions containing nonterminating symbols have to be removed.
Algorithm 11 describes how productions that include nonterminating symbols are removed from the grammar graph. The start node of the graph, the set of known terminating nodes and an empty set of already visited nodes are input at the first call of the algorithm. If the input node has been visited before, it is not visited again and the algorithm ends. If it has not been visited before the corresponding *Node_Key* is added to the set of visited nodes. Then the *children* list of the node is iterated (line 5). The index created in line 4 corresponds to the current element of the *children* list, that is iterated. It is needed to delete the *children* list and the corresponding *productions_list*, if a production contains nonterminating symbols. The *children* list is iterated in reverse, because, if a *children* list and *productions_list* is deleted this will not effect the indexing of the next *children* list to analyse. The *children* list contains lists of nodes for each production (see section **??**). Each list, nested in the *children* list, is iterated and for each node the algorithm is called recursively (line 7 and 8).
If a node represents a nonterminating symbol the *not_terminating* flag is set to true. If this flag is true after iterating one element of the *children* list, it means that the production corresponding to the *children* list element contains at least one nonterminating symbol. If that is the case, the corresponding element of the *children* list and of the *productions_list* are removed from the node. This is repeated for all elements of the *children* list.

As a result of this algorithm, all productions that contained nonterminal symbols have been removed from the grammar graph.

Algorithm 11 Delete non terminating productions

```
Input: node, terminating_set, visited_set
 1: node_key = Node_Key(node.value, node.rule_type)
 2: if node_key not in visited_set then
 3:     visited_set.add(node_key)
 4:     index = len(node.children) - 1
 5:     for all children_list in reversed(node.children) do
 6:         not_terminating = False
 7:         for all child in children_list do
 8:             delete_non_terminating_productions(child, terminating_set, vis-
    ited_set)
 9:             if not value_in_terminating(child.value, terminating_set) then
10:                 not_terminating = true
11:             end if
12:         end for
13:         if not_terminating then
14:             delete node.children[index]
15:             delete node.productions_list.list[index]
16:         end if
17:         index = index - 1
18:     end for
19: end if
```

After the productions containing nonterminating symbols have been removed from the grammar graph, the nodes that represent nonterminating symbols can also be removed from the *nodes_dictionary*. Algorithm 12 is responsible for this. It takes the set of *Node_Key*s as input, that specifies all terminating nodes. To remove the nonterminating nodes, in a for-loop, all terminating nodes that are addressed by the *Node_Key*s are stored in a temporary dictionary. This dictionary replaces the original *nodes_dictionary* of the *TPTPGraphBuilder*.

Algorithm 12 Delete non terminating nodes

```
Input: terminating_set
1: temporary_dictionary = {}
2: for all node_key in terminating_set do
3:     value = this.nodes_dictionary.get(node_key, None)
4:     temporary_dictionary.update(node_key: value)
5: end for
6: this.nodes_dictionary = temporary_dictionary
```

4.5.3 Determination of the remaining reachable symbols

Now, that nonterminating symbols have been removed from the grammar graph, the remaining reachable symbols have to be determined. As described in section 3.9.2 the determination of the remaining reachable is done by creating a new grammar graph, with the desired start symbol from the control file. The new grammar graph will only contain the reachable symbols (as described in section 3.9.4).

4.6 Input

There are two methods performing input operations that are explained in the following.

Importing the TPTP syntax from the web

The method *import_tptp_syntax_from_web* shown in listing 4.8 is used to get a textfile of the TPTP grammar from the TPTP website.
For opening a Uniform Resource Locator (URL), the library urllib is used.
Urllib.request opens a specified URL and sends an HTTP request. *Urlopen* returns a response object for the initiated request. The response is a file-like object that can be read using *.read*(). [25]
The file can then be converted from html to text format using BeautifulSoup's html parser. The header from the file is deleted because it does not belong to the TPTP syntax but is appended by urllib. The header is deleted by splitting the textfile of the grammar at every new line and joining all lines excluding the first one.

```
1 with urllib.request.urlopen("http://www.tptp.org/TPTP/SyntaxBNF.html") as url:
2         html_doc = url.read()
3 soup = BeautifulSoup(html_doc, 'html.parser')
4 tptp_grammar = soup.get_text()
5 # Delete Header
6 tptp_grammar = '\n'.join(tptp_grammar.split('\n')[1:])
```

Listing 4.8: Import TPTP syntax from web

Reading TPTP grammar from textfile

Read_text_fromtextunderscorefile in listing 4.9 uses the build-in function *open* to read a text file. The filename of the textfile has to be specified before.

```
with open(filename, 'r') as text_file:
        text = text_file.read()
```

Listing 4.9: Read text from file

4.7 Output generation

For saving the grammar graph in a text file, the graph needs to be traversed and converted to a string format that can be saved in a text document. The grammar graph is converted to a string format by converting every rule to a string and cohering them. Every rule is represented by a node including the nodes children list. For converting the graph to a string format, the graph is traversed and string representation of the nodes are created. The final output is stored in a *print_list*. A *print_list* is a list of *print_list_entries* that can be seen in listing 4.10. A *print_list_entry* is a tuple consisting of the position of the rule that in the original TPTP grammar is represented by the node and a string version of the rule or rather the node and its children list.

```
PrintListEntry = namedtuple("PrintListEntry", ["position", "string"])
```

Listing 4.10: Print List Entry

The detailed process of saving a string version of a rule is described in the following sections.

4.7.1 Saving string representation of graph in text file

In order to create the *print_list*, an empty set *visited* is created that is used for memorising which nodes have already been visited. Moreover, an empty *print_list* and *print_string* are created. The set *visited* and the empty *print_list* are used for calling the function *create_print_list* (see following section) which returns the final *print_list*. Because the rules in the *print_list* are not in the same order

as the rules in the original TPTP syntax, the *print_list* is sorted based on the rule's position that is stored in the first element of the *PrintListEntry* tuple. For converting the *print_list* to one string, the *print_string*, the *print_list* is iterated. Except for the first element of the list, every element of the list is added to the *print_string*. This is done by adding the strings that are stored in the second element of the *PrintListEntry* tuple and separating them by a new line. The first element is not added because it represents the general start symbol that has been added to the grammar (see chapter 3.6) but is not a part of the TPTP grammar. A text document with a filename that has been selected by the user is then opened and the *print_string* is written to the file. The simplified process can be seen in the flow diagram in figure 4.14.

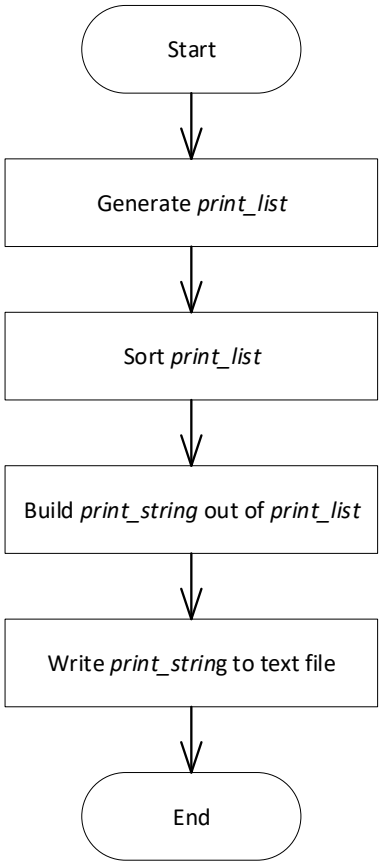

Figure 4.14: Save ordered rules from graph

4.7.2 Traverse graph to create string representations of objects

The graph is traversed using the function *create_print_list*. For every node a new *print_list_entry* is created. The str function that is called to create the *print_list_entry* is described in section 4.7.3. The *print_list_entry* is then appended to the *print_list*. Once visited a node is added to the set *visited* so that it will not be visited again. Recursively all children in the node's children list are traversed. Because a child can once again have a list, every element of the child's list is considered and calls *create_print_list* to complete the final *print_list*. The visited elements are then also added to visited set. The algorithms ends with returning his version of the *print_list* that will be complete list once the first node who called the function return its *print_list*.

Algorithm 13 Output Algorithm: create_print_list

Input: node, visited, print_list

```
1: Create PrintListEntry with node.position and str(node)
2: Append PrintListEntry to print_list
3: Add node to visited
4: for all children in node.children do
5:     for all elements in children do
6:         if element is not in visited then
7:             print_list = create_print_list(element, visited, print_list)
8:             Add element to visited
9:         end if
10:    end for
11: end for
12: return print_list
```

4.7.3 Create string representations of objects

This section covers the string representation creation for the data types that are used within the grammar graph. The algorithms outlined in the following are embedded in the *___str___()* method of each class. The goal is for the generated string representations to be in the same formate as original TPTP syntax.

NTNode string representation creation

An *NTNode* object represents a rule statement. This may include a comment block associated with that rule statement. The UML diagram of the *NTNode* class can be seen in figure 4.10. The algorithm that describes how a string representation of an *NTNode* object is generated is featured in algorithm 14. If a comment block is associated with an *NTNode*, its string representation is added to the *NTNode* string representation at the beginning. Then the value of the nonterminal symbol (left hand side of the production rule the *NTNode* represents) is added to the string representation. After that the corresponding production symbol corresponding to the *RuleType* (see table 3.1) of the *NTNode* is added to the string representation separated to the nonterminal symbol by a space character. To align the productions in the output file, a whitespace padding is added. Next, the string representation of the *productions_list* is added to the string representation of the *NTNode*. The procedure of how the string representation of the *productions_list* is created is outlined in the following after the description of the comment block string generation.

Algorithm 14 *NTNode* string creation

```
1: if this.comment_block is not None then
2:     node_string += str(this.comment_block)
3: end if
4: node_string += this.value
5: add whitespace padding
6: if this.rule_type == RuleType.GRAMMAR then
7:     node_string += " ::= "
8: else if this.rule_type == RuleType.TOKEN then
9:     node_string += " ::- "
10: else if this.rule_type == RuleType.STRICT then
11:     node_string += " :== "
12: else if this.rule_type == RuleType.MACRO then
13:     node_string += " ::: "
14: end if
15: node_string += str(this.productions_list) return node_string
```

COMMENT_BLOCK string representation creation

The *COMMENT_BLOCK* class has a list of comment lines as an attribute (see UML diagram in figure 4.2). The string representation of the *COMMENT_BLOCK* objects is created by concatenating the comment lines, separating the individual comment lines by newline characters.

PRODUCTIONS_LIST string representation creation

The *PRODUCTIONS_LIST* class contains a list of production alternatives of a rule (the right-hand side of a rule). The production alternatives are separated by " | " in the original TPTP syntax, which can be seen in the following listing.

```
1 <annotated_formula>     ::= <thf_annotated> | <tff_annotated> | <tcf_annotated> |
2                             <fof_annotated> | <cnf_annotated> | <tpi_annotated>
```

Listing 4.11: Production alternatives in the TPTP syntax

Algorithm 15 shows the process of creating the string representation of a *PRODUCTIONS_LIST*. The productions in *PRODUCTIONS_LIST* are iterated and the string representation of every production is created. The string of the productions are then separated by the " | " character. The counter in line 3 of algorithm 15 is used to identify the last production and thus not to print alternative symbol after that production.

Algorithm 15 *PRODUCTIONS_LIST* string creation

```
 1: productions_list_string = ""
 2: length = length(this.list)
 3: counter = 1
 4: for all production in this.list do
 5:     productions_list_string += str (production)
 6:     if counter < length then
 7:         productions_list_string += " | "
 8:     end if
 9:     counter += 1
10: end forreturn productions_list_string
```

PRODUCTION string representation creation

The *PRODUCTION* class has a list of elements and a *ProductionProperty* as an attribute (see UML diagram in figure 4.5). The elements of the list can be of type *PRODUCTION*, *PRODUCTION_ELEMENT* or *XOR_PRODUCTIONS_LIST*. Algorithm 16 describes the procedure of creating the string representation of a *PRODUCTION*. First, the *production_string* is initialised. Then, depending on the *ProductionProperty* characters that represent these productions properties are added to the *production_string*. If the *ProductionProperty* is *NONE* no special characters need to be added. In case of the *ProductionProperty REPETITION*, the open parenthesis character "(" is added to the *production_string* in order to enclose the *list* elements. In case of the *ProductionProperty OPTIONAL*, the elements have to be enclosed in square brackets. Therefore the "[" character is added to the *production_string* in that case.
After that, the *list* is iterated and the string representation of each element is created and added to the *production_string*.
Again depending on the *ProductionProperty* additional characters might have to be added. In case of the *ProductionProperty REPETITION* the open parenthesis is closed with the close parenthesis and the repetition symbol is added to denote the repetition property. In case of the *ProductionProperty OPTIONAL* the open square bracket is closed. Then the created *production_string* is returned.

Algorithm 16 *PRODUCTION* string creation

```
1: production_string = ""
2: if self.productionProperty == ProductionProperty.REPETITION then
3:     production_string += "("
4: else if element.productionProperty == ProductionProperty.OPTIONAL then
5:     production_string += "["
6: end if
7: for all element in this.list do
8:     production_string += str(element)
9: end for
10: if self.productionProperty == ProductionProperty.REPETITION then
11:     production_string += ")*"
12: else if element.productionProperty == ProductionProperty.OPTIONAL then
13:     production_string += "]"
14: end if
15: Return production_string
```

XOR_PRODUCTIONS_LIST string representation creation

The procedure to create the string representation of the *XOR_PRODUCTIONS_LIST* class is almost equivalent to the *PRODUCTIONS_LIST* process described in algorithm 15. The only difference is, that generated string are enclosed by parentheses.

PRODUCTION_ELEMENT string representation creation

The *PRODUCTION_ELEMENT* data type consists of a textitSYMBOL and a *ProductionProperty*, which can be seen in the UML diagram in section 4.3. The procedure of creating the string representation of a *PRODUCTION_ELEMENT* is outlined in the following algorithm 17. The string representation is created similar to the creation of the string representation of a *PRODUCTION*.

Algorithm 17 *PRODUCTION_ELEMENT* string creation

```
1: production_element_string = ""
2: if element.productionProperty == ProductionProperty.NONE then
3:     production_string += str(this.symbol)
4: else if element.productionProperty == ProductionProperty.REPETITION
   then
5:     production_string += "("
6:     production_string += str(this.symbol)
7:     production_string += ")"
8:     production_string += "*"
9: else if element.productionProperty == ProductionProperty.OPTIONAL then
10:    production_string += "["
11:    production_string += str(this.symbol)
12:    production_string += "]"
13: end if
14: production_element_string
```

Symbol string representation creation

To create the string representation of a non-terminal symbol, its value can be returned as-is. The terminal symbol data type additionally has a production property as a class attribute.

To create the string representation of a terminal symbol, in the case of the property *OPTIONAL*, square brackets have to be printed around the terminal symbol value. In case of the production property *REPETITION* the asterisk symbol has to be printed after the terminal symbol value. In case of the production property *NONE*, the string representation consists only of the terminal symbol value.

4.8 GUI

The GUI consists of the class View that can be seen in Figure 4.15. The class has the attributes *treeView* which is an instance of the PyQt QTreeWidget, *graphBuilder* which is an instance of the TPTPGraphBuilder and the boolean value *commentStatus*. The class functions and their implementation are described in the following sections based on the five sub-menus that have been introduced in chapter 3.11.

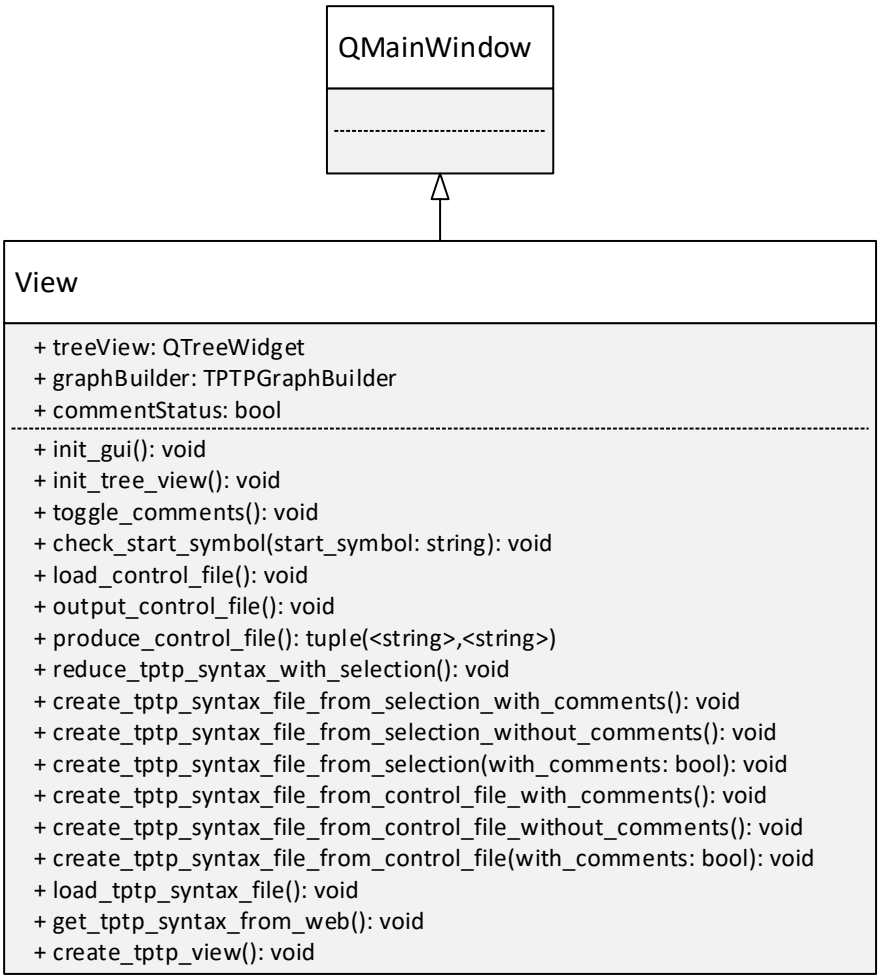

Figure 4.15: View UML class diagram

4.8.1 Main application window

Init GUI

The function *init_gui* sets up the menu bar in the main application window. The menu bar consists of the sub-menus "Import syntax", "Save syntax", "Reduce", "Control file" and "View". To each sub-menu menu actions that have been described in chapter 3.11 are added and implemented using QActions. The following listing 4.12 shows the Python code for adding a menu bar and menu actions for example for the "Import syntax" menu bar.

```
1 import_menu = menubar.addMenu("Import syntax")
2 import_menu.addAction(import_tptp_file_action)
```

```
3 import_menu.addAction(import_tptp_file_from_web_action)
```

Listing 4.12: Implementation of menu bar

Besides a name, a short cut for executing the menu action is defined. Also, triggers have been defined that trigger functions once the menu button is pressed. Listing 4.13 shows how a short cut and a trigger have been defined for the action $ImportTPTPSyntaxfile$.

```
1 import_tptp_file_action = QAction('&Import TPTP syntax file', self)
2 import_tptp_file_action.setShortcut('Ctrl+O')
3 import_tptp_file_action.triggered.connect(self.load_tptp_syntax_file)
```

Listing 4.13: Implementation of menu actions

Init TreeView

Init TreeView is used for displaying a grammar in the main application window. The method is called after importing or reducing a grammar. The nodes of the grammar are stored in a nodes dictionary whose values (the nodes) are then converted to a list. This list is displayed by transforming it into a PyQT TreeView. A tree view can be used for displaying nested lists and allowing a user to navigate through them. The tree view consists of all nodes whose leaves are the productions in their (nested) productions list. The nodes can be expanded to see their productions which are then displayed indented and underneath the node.

A new tree view is defined using QTreeWidget with the header labels "Nonterminal", "Production Type" and "Production" (line 1 and 2 in listing 4.14). All nodes of the nodes dictionary of the GraphBuilder instance are extracted and sorted based on their position in the original TPTP syntax file (line 3 and 4 in listing 4.14. The pseudo code in 18 shows the code used for filling the created treeView. For every node a new QTreeWidgetItem is generated. The item consists of the nodes (node.value) and its rule type. Also a new checkbox is defined that is displayed next to the item whose initial value is unchecked. The background colour to set to grey. For every production in the nodes production list, a new QTreeWidgetItem and checkbox are created as well. However, the default value of the checkbox is set to unchecked. The generated item is then added as a child to the node item. A potential comment block is added to the treeView and the item is added to the treeView as well.

```
1 self.treeView = QTreeWidget()
2 self.treeView.setHeaderLabels(['Non Terminal', 'Production Type', 'Production'])
3 nodes_list = list(self.graphBuilder.nodes_dictionary.values())
4 nodes_list.sort(key=lambda x: x.position)
```

Listing 4.14: Init Tree View

Algorithm 18 GUI Pseudo Code: init_tree_view

```
1: for all nodes in nodes_list do
2:     rule_type = node.rule_type
3:     item = new QTreeWidgetItem
4:     Uncheck checkbox of item
5:     Set background to grey
6:     for all productions in node.productions_list do
7:         child_item = new QTreeWidgetItem
8:         Check checkbox of child item
9:         Add child_item to item
10:    end for
11:    if node.comment_block exists then
12:        comment_item = new QTreeWidgetItem
13:        Add comment_item to treeView
14:    end if
15:    Add item to treeView
16: end for
```

4.8.2 Import menu

Import TPTP syntax file

The TPTP syntax file is imported using a QFileDialog which is returning the filename of the imported file selected by the user. If the filename is not empty, the application opens a dialog window where the user has to specify a desired start symbol. The method *read_text_from_file* returns the text of the file belonging to the filename. With the textfile and the start symbol, the method *create_tptp_view* is called.

Import TPTP syntax from the TPTP website

The TPTP syntax file is extracted from the TPTP website using the Input method *import_tptp_syntax_from_web* (see chapter REF). This method returns a textfile. The application opens a dialog window where the user has to specify a desired start symbol. With the textfile and the start symbol, the method *create_tptp_view* is called.

create_tptp_view
The method creates a new instance of the $TPTPGraphBuilder$, calls $graphBuilder.run$, *init_tree_view* and *check_start_symbol*.

Check start symbol

This function checks the checkbox of the defined start symbol. The QTreeView method $findItem(string\ itemName)$ is used to find the start symbol in the treeview and return all items that match the item name. The found items are then checked.

4.8.3 View menu

Toggle comments

Algorithm 19 shows the algorithm used for toggling comments. The algorithm should either show or hide the comments based on the current status. The View class attribute *commentStatus* indicates whether comments are displayed (true) or not (false). For toggling comments, a new status is set that is the opposite of *commentStatus*. While looping through every item in $treeView$, comment items can be found by checking whether the item is checkable because by default a comment is not checkable by the user. A QTreeWidgetItem has flags including a flag "ItemIsUserCheckable" that is used for determining whether the item is checkable. The QTreeWidgetItem function $setHidden(bool\ hide)$ is applied on every comment with the value of the new status. CommentStatus is set to the new status.

Algorithm 19 GUI Algorithm: toggle_comments

```
1: newStatus = not commentStatus
2: for all items in treeView do
3:     if item is not checkable then
4:         item.setHidden(newStatus)
5:     end if
6: end for
7: commentStatus = newStatus
```

4.8.4 Reduce menu

Reduce TPTP syntax with selection

The method reduce TPTP syntax with selection reduces the syntax and displays the resulting syntax in the GUI. For reducing the TPTP syntax based on the users selection in the GUI, a control file is produced. If no grammar has been imported, no start symbol or multiple start symbol have been selected an error is raised. The error is displayed using a QMessageBox. With the produced control file, the Graphbuilder disables blocked productions, treeView is initialized and the start symbol is checked (see listing 4.15).

```
1 control_string, start_symbol = self.produce_control_file()
2 self.graphBuilder.disable_rules(control_string)
3 self.init_tree_view()
4 self.check_start_symbol(start_symbol)
```

Listing 4.15: Reduce TPTP syntax with selection

4.8.5 Save syntax menu

Create TPTP syntax file from selection

Figure 4.16 shows a flow diagram of the process of reducing and saving a TPTP syntax file from a user's selection. The input of this method is a boolean value *with_comments* that describes if the syntax should be saved with or without comments for the automatic parser generator (see chapter 5.1.2). First, the filename is import using a *QFileDialog*. If the import is successful, meaning a filename has

been selected, a control file based on user's selection is produced. If the import is not successful, the method terminates.
In comparison to *Reduce TPTP syntax with selection* the reduced syntax is not displayed in the GUI but only saved to local storage and the GUI displays still the full syntax. If the user wishes to display the reduced syntax in the GUI and save it, he can use the button *Reduce TPTP syntax with selection* and then use the button *Reduce and save TPTP syntax with selection*. Because the full syntax is still displayed, the graphBuilder object storing the syntax cannot be modified. Therefore, the old graphBuilder is copied to a new graphBuilder object. Because the old object should not change, the new object should have copies of all objects that are part of the graphBuilder rather than references. To copy the graphBuilder object which is a compound object it is necessary to use *copy.deepcopy* instead of *copy.copy*. Deepcopy creates a new object and inserts copies of all objects that are part of to be copied object. Copy however, creates a new object and inserts references to the to be copied objects.
The new graphBuilder is initialized with the start symbol that has been provided by the generated control file and rules that include blocked productions are disabled. During the initialization the new start symbol $< start\_symbol >$ has been added (see chapter 3.6 for explanation). This start symbol is searched in the reduced syntax. If it can be found, the syntax is saved with or without comments based on the input parameter. However, if the start symbol cannot be found in the syntax no part of the grammar is terminating. In this case the saved file is empty. To save the grammar the method $Output.save\_ordered\_rules\_from\_graph(\_with\_comments)$ is used.

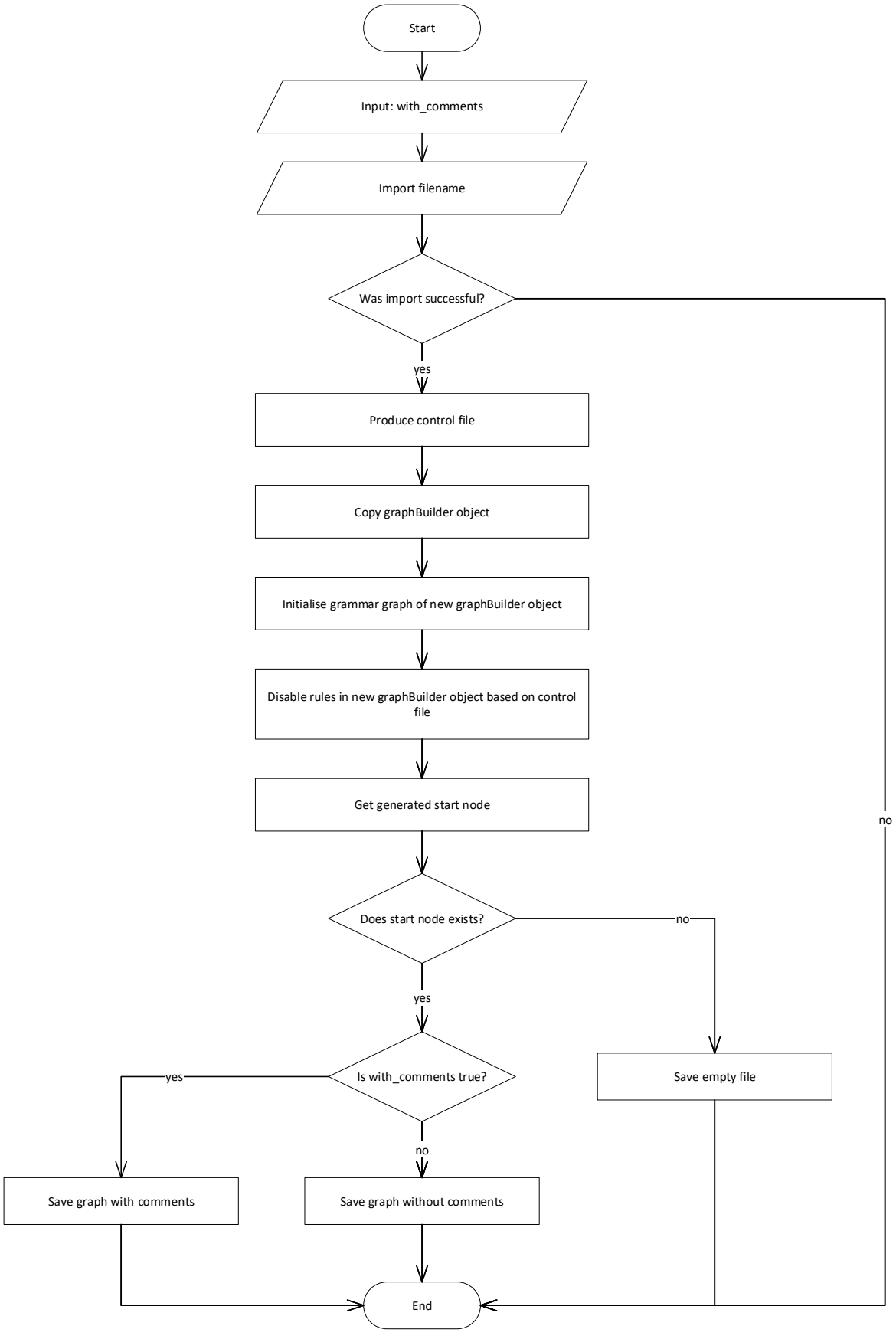

Figure 4.16: Reduce and save TPTP syntax

In the GUI there are two options of reducing and saving the TPTP syntax namely *Reduce and save TPTP syntax with selection* and *Reduce and save TPTP syntax from selection with external comments*. These two options both call the function described above and in figure 4.16. However, one calls it and passes *with_comments* as true and the other as false. Because each button in the GUI needs it own designated function to call, the functions $create\_tptp\_syntaxfile\_from\_selection\_with\_comments$ and $create\_tptp\_syntaxfile\_from\_selection\_without\_comments$ are introduced. They simply set the parameter *with_comments* accordingly and call the function *Create TPTP syntax file from selection*.

Create TPTP syntax file from control file

The previous section explained how a TPTP syntax is reduced and saved based on a user's selection in the GUI. The described function in this section also reduces and saves a TPTP syntax but this time based on an imported control file. The function is similar to the previous function *Create TPTP syntax file from selection*. Figure 4.17 shows the same flow diagram as figure 4.16 but highlights applied changes in red. Changes are that besides importing a filename, a name for the control file has to be imported as well. Instead of producing a control file based on the users selection, the specified control file is read and the grammar graph is initialised with the start symbol that is specified in the first line of the control file.

Also similar to the previous section, the functions $create\_tptp\_syntax\_from\_control\_file\_with\_comments$ and $create\_tptp\_syntax\_from\_control\_file\_without\_comments$ are introduced that set the parameter *with_comments* accordingly and call the function *Create TPTP syntax file from control file*.

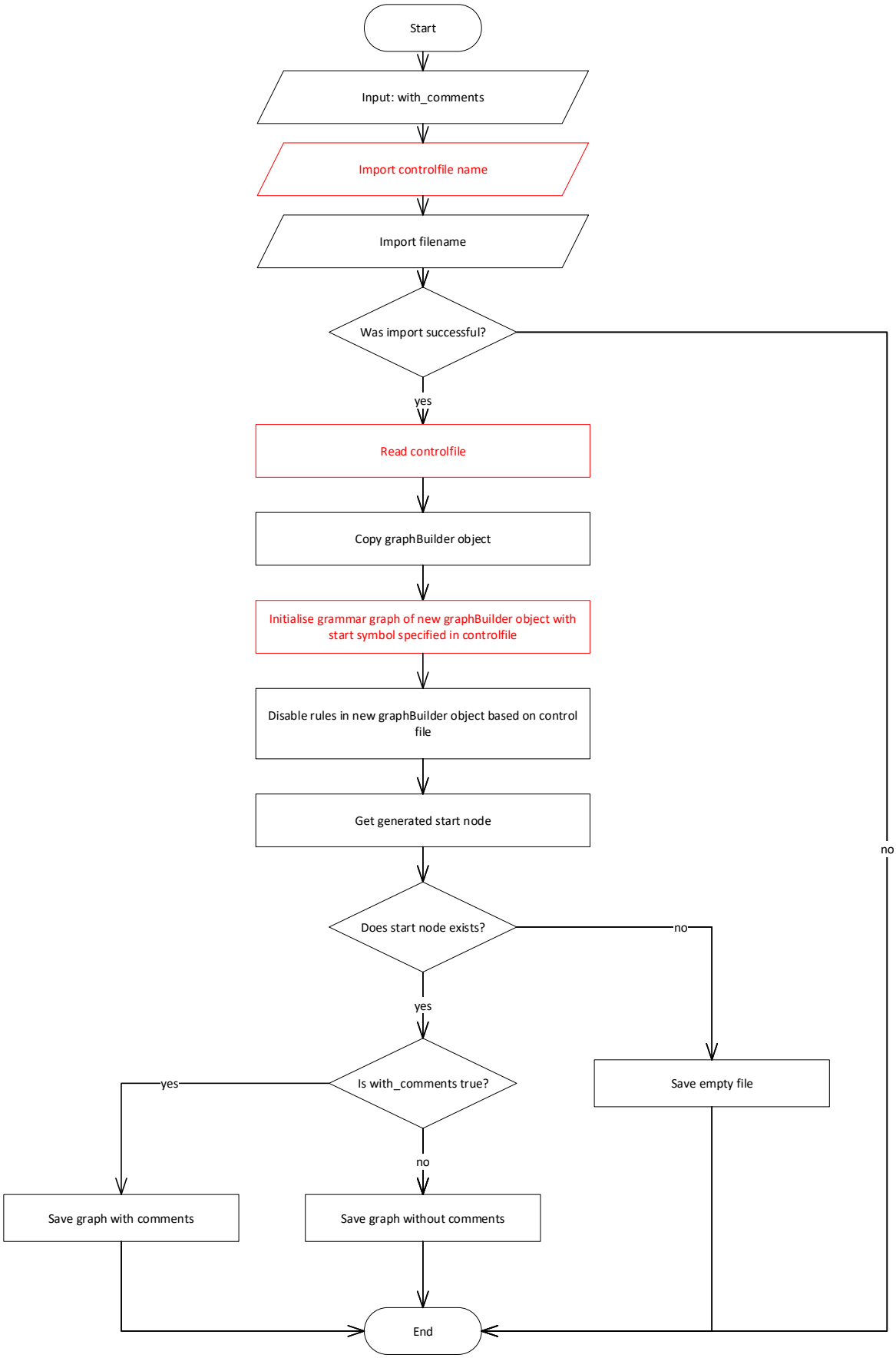

Figure 4.17: Reduce and save TPTP syntax from control file

4.8.6 Control file menu

Import control file

If a control file is imported, checkboxes have to be set accordingly. The pseudo code in 20 shows the process. The name of the control file is read in using a QFileDialog. The file itself is imported using the method $Input.read\_text\_from\_file$ (see chapter 4.6). If the file exists and if a tree view exists, meaning a grammar has been imported, all nodes are unchecked and all leaves/productions are checked. The different items representing nodes and leaves can be distinguished by looking for whether an item has parents and if not if it is checkable. The checkboxes are set this way because the default mode is that all nodes are unchecked and all productions are checked. After setting the checkboxes to the default mode, the first line of the control string is read and the start symbol is set accordingly. If there are multiple start symbols, every start symbol is checked. Iterating every line of the control string, the *nt_name* which is the first element of a line is saved and the *rule_type* which is the second element as well. The item that belongs to *nt_name* is then searched and every production whose index is listed in the control file is unchecked.

Algorithm 20 GUI Pseudo Code: load_controlfile

```
Import Control File
if Control File exists then
    for all items in tree view do
        if parent of item is None then
            if item is not a comment then
                Uncheck item
            end if
        else
            Check item
        end if
    end for
    Get first line of control file
    Search item that represents start symbol
    Check item
    for all lines in control file do
        Split line by comma
        nt_name = first element of split
        rule_type = second element of split
        Search nt_name node in tree view
        if  parent is None and rule_type matches rule_type then
            for all indices in line do
                if child exists then
                    Uncheck child
                end if
            end for
        end if
    end for
end if
```

Produce control file

Algorithm 21 shows how the control file used for reducing the TPTP syntax is produced. The index of productions that have been blocked by the user in the GUI are identified and stored in a dictionary until further processing. The key of the dictionary is a tuple made of a value and a rule type that has to be defined alongside the dictionary. Besides the blocked productions, the start symbol is featured in the control file as well, therefore an empty list for listing possibly multiple start symbols is created. Multiple start symbols can occur when the nonterminal symbol has multiple rule types or if the users makes an invalid input. The dictionary is

used for collecting all blocked productions. Looping through every node of the tree view it is checked, if the item has parents and if its checkbox is unchecked. If so, the rule type of the parent is identified and a new tuple is defined. The index of the production in the parent's productions list is identified as well. If the tuple is not already in the dictionary, a new dictionary entry is created with the tuple as key and the index of the item as value. If the tuple is already in the dictionary, the index of the item is added to the existing indices.
However, if the item is checked and has no parents, the name of the item is appended to the list of start symbols.
If there are multiple start symbols in the list that do not have the same nonterminal value or no start symbol at all, the user made an invalid input and an error is raised. The final control string is made up of first entry of the start symbol list. The first entry is sufficient because the control file only specifies the start symbols name and not the rule type. If there are multiple start symbols, they have the same nonterminal name. Each in a new line, the nonterminal symbol name, rule type and indices of blocked productions are appended to the control string. Within the line, the name, rule type and indices are separated by a comma. Moreover, an error is risen if there is no imported grammar, for example the user presses the Reduce grammar button in the GUI without importing a grammar.

Algorithm 21 GUI Algorithm: produce_controlfile

```
1: Define new tuple Entry("Entry", ["value", "rule_type"])
2: Create empty dictionary
3: Create empty list start_symbol
4: for all items in treeView do
5:     Get parent of item
6:     if item has parent and is unchecked then
7:         Get rule type of parent
8:         Create entry tuple with name of parent and its rule type
9:         Get index of child from parent
10:        if entry tuple is not in dictionary then
11:            Create new dictionary entry with entry tuple as key and list containing
    the index of the child as value
12:        else
13:            Append index of child to list
14:        end if
15:    else if item has no parent and is checked then
16:        Append item name to start symbol list
17:    end if
18: end for
19: control_string = first entry of start symbol list
20: Append all entries of dictionary to control_string
```

Output control file

Output control file saves a control file to local storage. First, a control file is produced based using the method *Produce control file*. A filename under which the control file is stored is input using a QFileDialog window. The control file is then stored using $Output.save\_text\_to\_file$.

4.8.7 Exception handling

Exceptions are used to handle wrong input from the user. There are three exceptions that are raised whenever the user makes a wrong input:

- Multiple start symbols
- No start symbol

- No imported grammar

The exception *multiple start symbols* is raised if the user selects multiple start symbols in the GUI and presses any Reduce button. On the contrary the exception *no start symbol* is raised if the user selects no start symbol in the GUI and presses any Reduce button. The exception *no imported grammar* is raised if the user starts the application and does not import a grammar but presses a button that requires a grammar for example in the Reduce, View or Control File menu.

4.9 Command-line interface

With the argparse module a command-line parser object can be created and parameters can be added to that object following the concept describet. In the first line of listing 4.16 the command-line parser object is created including a description. Lines two to five contain the specification of the accepted arguments. In addition to the name and short form of the name, the type, a help message, and whether the parameter is optional or not can be specified. Default values for arguments can also be specified. If no default value is specified and if the argument is not passed it will have the value *None*. The *action* =′ $store_t rue$′ in line 5 means, that if this argument flag is present in the command-line call, the value of this argument will be set to $True$. If the flag is not present it's value will be set to $False$. Argparse automatically checks the given conditions, for example if a required argument is not given and displays an error message if that is the case.

```
1 self.argument_parser = argparse.ArgumentParser(description='Extract sub-syntax using
      TPTP syntax file and a control file')
2 self.argument_parser.add_argument('-g', '--grammar', metavar='', type=str, required=
      True, help='path of the TPTP syntax file')
3 self.argument_parser.add_argument('-c', '--control', metavar='', type=str, required=
      True, help='path of the control file')
4 self.argument_parser.add_argument('-o', '--output', metavar='', type=str, required=
      False, help='optional output file name (default output.txt)', default= "output.
      txt")
5 self.argument_parser.add_argument('-ec',  '--external_comment', action='store_true',
      help="flag - include external comment syntax")
```

Listing 4.16: Argparse command-line parser configuration

Argparse will also automatically create the help output by using the descriptions provided when configuring the argument parser.

5 Validation

show advantages and useful for tptp users... show size before after

This chapter reflects on the sub-syntaxes generated by Synplifier. It demonstrates that sub-syntaxes are compatible with the TPTP syntax format and compares the sizes of several sub-syntaxes of interest.

5.1 Automated parser generation

A goal of using Synplifier is to be able to use an extracted sub-syntax with the automated parser generator for the TPTP syntax [7]. To ensure compatibility it is possible to export an extracted sub-syntax and adding the part of the syntax concerning comments, even though it is not reachable in the original syntax (see section 3.10.2). If this part of the syntax would not be part of the output sub-syntax file automated parser generation would result in an error because this syntax part is expected to be present.
Also, the automated parser generator is used to check if the output sub-syntax follows the original TPTP syntax format.

5.1.1 Building a basic parser

To demonstrate the usability and capability of Synplifier a parser parsing only CNF, that counts the number of CNF clauses, is used. The creation of the parser can be divided in the following steps:

1. Extract the CNF sub-syntax from the original TPTP syntax using Synplifier.
2. Generate lex and yacc file based on the sub-syntax using the automated parser generator.
3. Modify the generated yacc parser to count CNF clauses.

CNF sub-syntax extraction

The following listing 5.1 contains the control file content, that extracts CNF from the TPTP syntax version 7.3.0.0. The start symbol is *TPTP_file*.

```
1 <TPTP_file>
2 <annotated_formula>,::=,0,1,2,3,5
3 <annotations>,::=,0
```

Listing 5.1: Control file to extract CNF

All productions except the *cnf_annotated* are disabled from the *annotated_formula* grammar rule (line 2 of listing 5.1). The *annotated_formula* grammar rule can be seen in the following listing 5.2. Annotations are also disabled (line 3 in listing 5.1).

```
1 <annotated_formula>     ::= <thf_annotated> | <tff_annotated> | <
      tcf_annotated> |
2                             <fof_annotated> | <cnf_annotated> | <
                                  tpi_annotated>
```

Listing 5.2: *annotated_formula* production rule

To extract the CNF sub-syntax using the control file content from listing 5.1 either the command-line interface or the GUI can be used.

Lex and yacc file generation

The generated sub-syntax is input to the automated parser generator. The automated parser generates a lex and yacc file, and also the corresponding c files from that.

CNF clause counter implementation

To count the CNF clauses the counter *cnf_counter* has been added to the parser. Also a main function is added, that can be seen in listing 5.3. Using this function, either a file can be passed to the parser, or the input can be provided via the command-line. After parsing is complete, the total number of CNF clauses is output to the console.

```
int main(int argc, char **argv ){
   ++argv, --argc;  /* skip over program name */
    if(argc>0){
        yyin = fopen(argv[0],"r");
    }
    else{
        yyin = stdin;
    }
    yyparse();
    printf("Total count of cnf clauses: %d\n", cnf_counter);
}
```

Listing 5.3: CNF parser main-function

The incrementation of the *cnf_counter* is done in the yacc rule-action, that has been created by the automated parser generator, for the *cnf_annotated* symbol.

5.1.2 Testing the generated parser

The generated parser has been successfully tested on TPTP problems. It returns an error if other logics than CNF are present and correctly counts the number of CNF clauses.

The automated parser generator can be used with any sub-syntax generated by Synplifier, for example sub-syntaxes with FOF, or FOF and CNF.

5.2 Syntax size comparison

Based on the extracted CNF and FOF sub-syntax, the size of the sub-syntaxes in comparison to the full TPTP syntax is analyzed. Table 5.1 contains the number of rule statements and rules of the original TPTP syntax and extracted sub-syntaxes. A rule statement in this case means the combination of left hand side non-terminal symbol, rule type and rule alternatives. Since rule alternatives are just a convenient way to display multiple rules with same non-terminal symbol on the left hand side and the same rule type, in addition to the number of rule statements the number of rules is compared.
The reachable part of the TPTP syntax in the second row of the table contains only the reachable symbols counted from the start symbol <*TPTP_file*>. Rule

statements are counted by counting the nodes in grammar graph and the number of rules is counted by counting the productions list entries of all nodes.
Table 5.1 implies that the FOF sub-syntax contains only 35 % of the rule statements of the full syntax. The CNF sub-syntax contains only circa 30 % of the rule statements of the full syntax. Compared with the total number of reachable rules in the TPTP syntax, FOF contains circa 33 % of the total reachable rules and CNF contains circa 28 % of the total reachable rules. This shows a significant syntax reduction which was one goal of the tool usage.

Table 5.1: TPTP syntax size comparison

Syntax	**Rules**	**Productions**
TPTP syntax v. 7.3.0	313	635
Reachable part of TPTP syntax v. 7.3.0	285	595
FOF	108	198
CNF	92	165

6 Conclusion

This paper described Synplifier, a software tool that automatically extracts sub-syntaxes from the TPTP syntax. The tool includes lexing and parsing the TPTP syntax, building a grammar graph that represents the syntax and extracting a sub-syntax based on the graph. The user can interact with the tool using a command-line interface or a GUI. Both the GUI and the command-line interface use a developed control-file format that specifies blocked productions and the start symbol. For the GUI a concept to display the grammar in a clear and structured way using a tree structure has been developed and implemented. The functionality of the tool has been demonstrated by extracting the CNF sub-syntax and creating a parser based on that syntax, which has been successfully tested with a number of TPTP problems.

All requirements, outlined in section 3.2, have been fulfilled. In addition, features like importing the original TPTP syntax from the TPTP website, toggling comments in the GUI and counting rules as well as rule productions have been included to increase the user-friendliness of Synplifier.

The tool reduces the entry barrier for new users to use the TPTP language because desired sub-syntaxes of the TPTP syntax can be extracted while maintaining consistency with the original TPTP syntax.

In short, the developed software tool enables users to consistently, conveniently, and automatically extract sub-languages from the TPTP syntax, turning this one language into a family of languages, coming from a single source, but optimized for particular use cases.

6.1 Future Work

The concept of sub-syntax extraction is not only limited to the TPTP language and can also be applied to any other computer language for example the SMT-lib [26].

In order to adapt Synplifier to extract sub-syntaxes from other languages, mainly the lexer and the parser components would have to be modified to parse the new syntax. Other components can be used with only minor modifications because the logic of the sub-syntax extraction would mostly remain the same. Furthermore, comment association would have to be adapted based on the specific comment convention (if comments are present in the syntax description).

It is also be possible to create a tool that accepts plain BNF or EBNF since many computer languages are described in these forms. This would make the developed tool a standard tool since it could then be used with any arbitrary computer languages that is described in BNF/EBNF.

Apart from adopting the tool to other languages, it can be investigated if comment association based on the comments content would lead to a significantly better result than using the heuristic approach that we have introduced.

Bibliography

Publications

[1] G. Sutcliffe. "The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0". In: *Journal of Automated Reasoning* 59.4 (2017), pp. 483–502.

[2] G. Sutcliffe. "The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 337–362.

[3] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. "The TPTP Typed First-order Form with Arithmetic". In: *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by N. Bjørner and A. Voronkov. Lecture Notes in Artificial Intelligence 7180. Springer-Verlag, 2012, pp. 406–419.

[4] Jasmin Christian Blanchette and Andrei Paskevich. "TFF1: The TPTP typed first-order form with rank-1 polymorphism". In: *Proc. of the 24th CADE, Lake Placid*. Ed. by Maria Paola Bonacina. Vol. 7898. LNAI. Springer, 2013, pp. 414–420.

[5] G. Sutcliffe and C. Benzmüller. "Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure". In: *Journal of Formalized Reasoning* 3.1 (2010), pp. 1–27.

[6] Cezary Kaliszyk, Geoff Sutcliffe, and Florian Rabe. "TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism". In: *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*. Ed. by P. Fontaine, S. Schulz, and J. Urban. CEUR Workshop Proceedings 1635. 2016, pp. 41–55. URL: http://ceur-ws.org/Vol-1635/.

[7] A. Van Gelder and G. Sutcliffe. "Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation". In: *Proceedings of the 3rd International Joint Conference on Automated Reasoning*. Ed. by U. Furbach and N. Shankar. Lecture Notes in Artificial Intelligence 4130. Springer-Verlag, 2006, pp. 156–161.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation 3rd Edition*. Pearson Education, Inc., 2007. ISBN: 0321455363.

[9] Armin Cremers and Seymour Ginsburg. "Context-free grammar forms". In: *Journal of Computer and System Sciences* 11 (1975), pp. 86–117.

[10] Donald E. Knuth. "Backus Normal Form vs. Backus Naur Form". In: *Commun. ACM* 7.12 (Dec. 1964), pp. 735–736. ISSN: 0001-0782. DOI: 10.1145/355588.365140. URL: https://doi.org/10.1145/355588.365140.

[11] Niklaus Wirth. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?" In: *Commun. ACM* 20.11 (Nov. 1977), pp. 822–823. ISSN: 0001-0782. DOI: 10.1145/359863.359883. URL: https://doi.org/10.1145/359863.359883.

[12] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly Media Inc., 1992. ISBN: 9781565920002.

[13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, & Tools, Second Edition*. Ed. by Michal Hirsch, Matt Goldstein, Katherine Harutunian, and Jefferey Holocumb. Second. Addison-Wesley, 2007.

[14] Torben Aegidius Mogensen. *Introduction to Compiler Design*. Springer, 2017. ISBN: 9783319669656.

[22] Scott Bradner. *A practical test for univariate and multivariate normality*. Tech. rep. Internet Engineering Task Force, Mar. 1997. URL: https://tools.ietf.org/html/rfc2119.

[26] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2017.

Online sources

[15] M. E. Lesk and E. Schmidt. *Lex - A Lexical Analyzer Generator*. URL: http://dinosaur.compilertools.net/lex/index.html (visited on 05/16/2020).

[16] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. URL: http://dinosaur.compilertools.net/yacc/index.html (visited on 05/16/2020).

[17] David Beazley. *PLY (Python Lex-Yacc)*. URL: https://www.dabeaz.com/ply/ (visited on 01/26/2020).

[18] Riverbank Computing Limited. *Introduction - PyQt v5.14.0 Reference Guide*. URL: https://www.riverbankcomputing.com/static/Docs/PyQt5/introduction.html (visited on 03/09/2020).

[19] Python Software Foundation. *argparse - Parser for command-line options, arguments and sub-commands - Python 3.8.2 documentation*. URL: https://docs.python.org/3/library/argparse.html (visited on 03/09/2020).

[20] Leonard Richardson. *beautifulsoup4 4.9.0*. URL: https://pypi.org/project/beautifulsoup4/ (visited on 04/08/2020).

[21] Leonard Richardson. *Beautiful Soup Documentation*. URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (visited on 05/10/2020).

[23] Python Software Foundation. *The Python Standard Library: Built-in Types*. URL: https://docs.python.org/3/library/stdtypes.html#str (visited on 05/10/2020).

[24] Python Software Foundation. *tkinter — Python interface to Tcl/Tk*. URL: https://docs.python.org/3/library/tkinter.html (visited on 04/08/2020).

[25] Michael Foord. *HOWTO Fetch Internet Resources Using The urllib Package*. URL: https://docs.python.org/3/howto/urllib2.html (visited on 04/18/2020).