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.

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 paper describes a tool 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 users selection which part of the language to maintain. The tool has been successfully tested by extracting CNF from the TPTP language.

Contents

Acronyms

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
TFF	Typed First-order Form
THF	Typed Higher-order Form
TPTP	Thousands of Problems for Theorem Provers

List of Figures

List of Tables

Listings

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 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.
The developed tool 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 corresponing 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. By means of the background information, the third chapter outlines the concept of the developed tool. Based on this, the implementation of the tool is featured in the fourth chapter. The fifth chapter presents an evaluation of the effectiveness? of the tool. Considering the evaluation, the sixth chapter sums up the results of the developed tool, compares the results to the defined goals in first chapter and offers an outlook for possible future research.

2 Background and Theory

This chapter introduces the technologies and background that will be utilised 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 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 Extended Backus-Naur Form (EBNF). [1]

TODO more detailed

2.2 Formal languages

A formal language is a set of words over an alphabet.

An alphabet is a finite, nonempty set of symbols usually represented by $\sum$. An example is the binary alphabet $\sum = \{0, 1\}$. A string is a finite sequence of symbols from some alphabet. For example the string 101 is a string over the binary alphabet $\sum = \{0, 1\}$. A language is set of strings. If $\sum$ is an alphabet, then $L(\sum)$ is a language over $\sum$. [2] - Vocabulary

2.2.1 Finite automata

2.2.2 Regular expression

A regular expression is an algebraic description of a regular/formal language. Regular expressions declare strings that are part of the language. [2] For example the regular expression **10+1*** denotes the language consisting of a single 1 followed by a single 0 or any number of 1's.

2.2.3 Formal grammars

Unlike regular expressions, grammars not only describe a language but also define a structure of the words of a language.
A grammar is a list of rules that defines the relationships among tokens [3]. These rules are also referred to as production rules. Given a start symbol, this symbol can be replaced by other symbols using the production rules. Using a recursive notation, production rules define derivations for words. The derived symbols can then once again be replaced until the derivation is a terminal symbol. Terminal symbols describe symbols that cannot be further derived. The alphabet of the described language is build by the set of terminal symbols. Nonterminal symbols however can be further derived and build merged with the terminal symbols the vocabulary of a grammar. Nonterminal symbols and terminal symbols are disjoint.
- Beispiel

Context-free grammar

Reduced grammars

Grammars are called reduced if each nonterminal symbol is terminating and reachable [4].
Given the set of terminal symbols $\sum$, a nonterminal symbol ξ is called terminating if there are productions $\xi \underrightarrow{*} z$ 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. [4]
Given the set of terminal symbols $\sum$ and the start symbol S, a nonterminal symbol

ξ is called reachable if there are production rules $S \xrightarrow{*} u\xi v$ so that S can be derivated 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. [4]

todo beispiel

2.3 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. [5] An example for a BNF production would be $TPTP\_File ::= <TPTP\_Input> \mid <comment>$. Using this pattern of notation whole grammars can be specified.
The 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.

[6]

2.4 Lexing

Lexing or a so-called lexical analysis is the division of input into units called tokens [3]. Tokens are for example variable names or keywords. The input is a string containing a sequence of characters, the ouput is a sequence of tokens. Afterwards, the ouput can be used for further processing like parsing. 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. [7]

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 automata that tests each token simultaneously. This automata can be build 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. [7]
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. [7]

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 [3]. For doing so, a parser builds a syntax tree out of the generated tokens [7].
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). The output is the generated parser. [3]

During the syntax analysis a parser takes a string of tokens and forms a syntax tree with this construct by finding the matching derivations. The matching derivation can be found by using different approaches for examble random guessing (predictive parsing) or LR parsing. Input: description of grammar [3] Output: parser [3]

-bottom up (LR parsing): parser takes inputs and searches for production where input is on the right side of a production rule and then replaces it by the left side

-top down (predictive parsing): parser takes input and searches for production where input is on the left side of a production rule

2.6 Lex and Yacc

Lex -specify lexer by regular expressions -Lex compiler generates code from input in file lex.yy.c (p.140) Yacc -transforms yacc specification to c file y.tab.c -y.tab.c represens LALR parser (p. 287) [8]

-automated parser generator creates lex and yacc specifications and generated files from TPTP syntax

2.7 Python

todo why python

2.7.1 PLY

Python Lex-Yacc (PLY) [9] is an implementation of lex and yacc in python. [LALR-parsing] consists of lex.py and yacc.py

lex.py tokenizes an input string

2.7.2 PyQt

PyQt is a Python binding for the cross-platform GUI framework Qt [10]. It is licensed under the GNU GPL version 3. - QMainWindow
- QTreeWidget

tkinter

2.7.3 argparse

The python module argparse [11] is a module for creating command-line argurment 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.

3 Concept

This chapter outlines the concept and the architecture of the software tool. First, in section 3.1, the requirements the software tool needs to meet are described. Then, in section 3.2, the components needed are introduced. Then the proposed software architecture is described. After that the concept of each component is developed.

3.1 Requirements

The tool should meet the following requirements:
The tool has a GUI that is the interface between the tool and a user. Hence, the user communicates with the tool via the GUI. The user is able to import a syntax file. After the syntax file is imported, it should is displayed. This includes displaying by the syntax defined productions as well as comments that are associated with these productions. The user can select a new start symbol and can select which productions should be blocked. After the user made his choice, the new sub-syntax is generated and displayed. The tool can also generate a control file listing blocked productions and the start symbol. Furthermore, the tool is able to import a control file and extract a sub-syntax based on this control file instead of extracting a sub-syntax based on a users selection of blocked productions. The new sub-syntax can be exported to .txt format. Also, comments referring to the remaining productions are kept and comments referring to productions that were discarded are not be included in the sub-syntax. The tool also provides a console interface. This interface accepts a TPTP syntax file and a control file and output the sub-syntax described in the control file. It is possible to specify the output path and filename.

3.2 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 should be 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 grammar. That new graph represents the syntax of the extracted sub-language. To make this grammar usable, lastly the syntax has to be output, based on the new graph, in the same format as the original syntax.

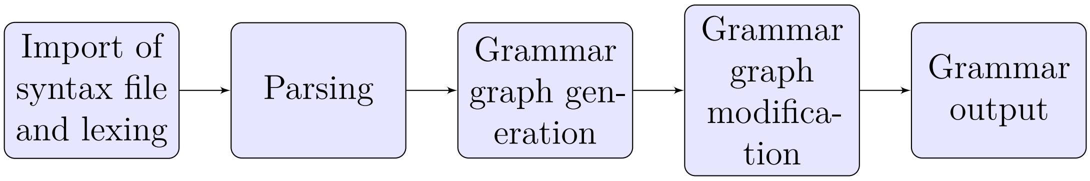

Figure 3.1: Procedure of extracting a sublanguage

3.2.1 Proposed architecture

The architecture of the software tool should take the procedure of extracting a sublanguage (section 3.2) into consideration. From that, five main components can be identified: An import module responsible for importing the TPTP syntax from a file; A lexer for extracting tokens from the language specification; A parser for creating a data structure from the tokens; A graph builder and manipulator; An export module for exporting the graph in a text representation corresponding to the original language specification.
In addition to the components that provide the main functionality a graphical user interface and a console interface for user convenience is desired.

Figure 3.2 contains a high-level UML diagram describing the architecture of the software tool. 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.

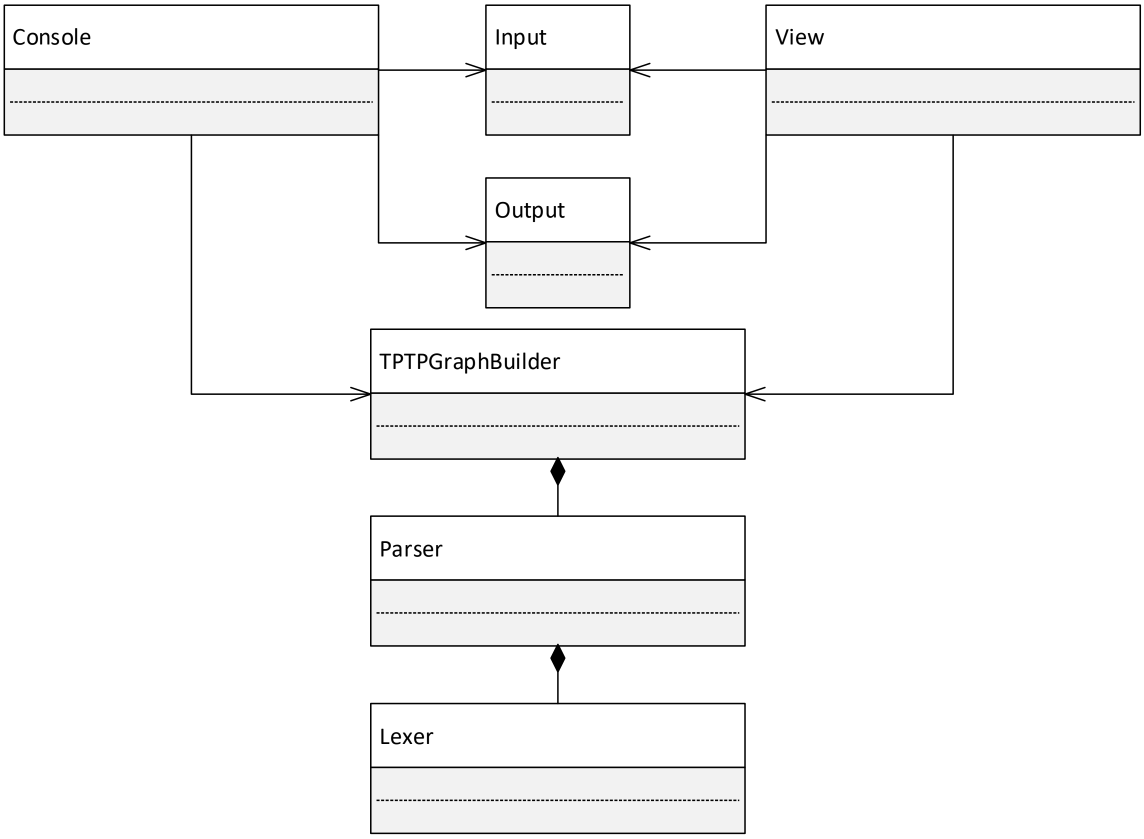

Figure 3.2: UML diagram of the architecture of the software tool

3.2.2 Implementation language

todo why python

3.3 Lexer

The lexer is responsible for extracting tokens from the TPTP language grammar specification file. Using PLY a lexer can be built by specifying tokens as regular expressions.

Therefore the TPTP language grammar specification needs to be analysed in order to find elementary tokens and regular expressions, that precisely describe these tokens.

The standard extended BNF 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. 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 [14]

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

The following paragraph introduces the tokens that are recognized by the lexer. Tokens are written bold.

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 ?is matched if none of the other tokens are matched?.

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 example below features two tokens, a grammar expression and a nonterminal symbol.

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

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 would be ignored in the input stream of the lexer. However, if one of those characters are part of a regular expression they are 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 example below.

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

3.4 Parser

The parser takes the tokens from the lexer as input and creates a data structure that represents the structure of the TPTP syntax.

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. The rules in the data structure have to be numbered, to maintain the correct order for output, after creating the grammar tree in the next step (see section **??**).

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.

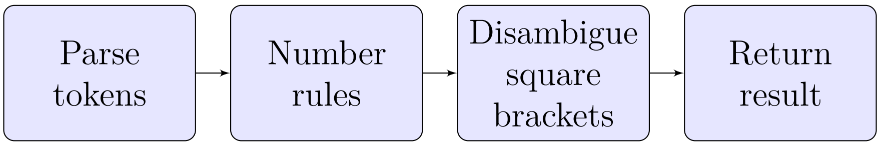

Figure 3.3: Parsing procedure

3.4.1 Data structures and data types

To build the representative data structure, data types that represent the data stored in the TPTP syntax have to be defined. The following section describes the data structure and data types that are used and created by the parser in the parsing process.

Terminal symbol

The terminal symbol data type has one attribute, which is the name of the terminal symbol it represents.

-todo Production Property

Nonterminal symbol

Analogue to the terminal symbol data type, the nonterminal symbol also has its name as an attribute.

todo describe diagram

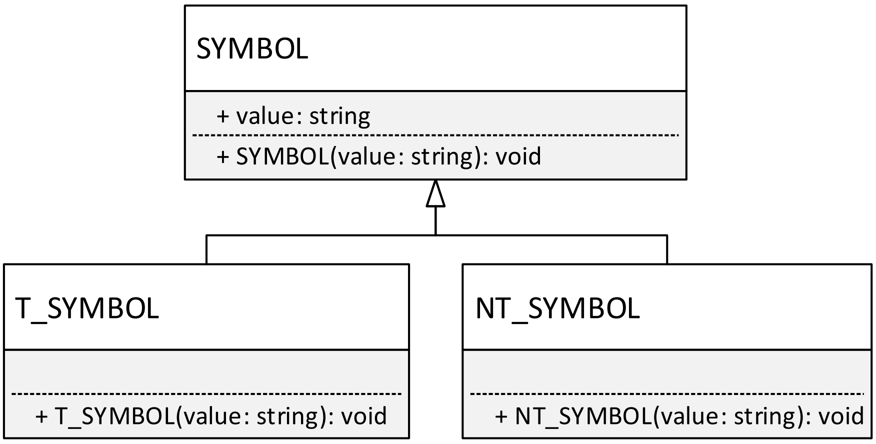

Figure 3.4: Symbol data type UML class diagram

Rules

A rule consists of the nonterminal symbol name which is produced, 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 (see table 3.1) there is a data type. This means that grammar, token, strict and macro rule data types are introduced.
Listing 3.1 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.

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

Listing 3.1: Grammar expression

todo describe diagram

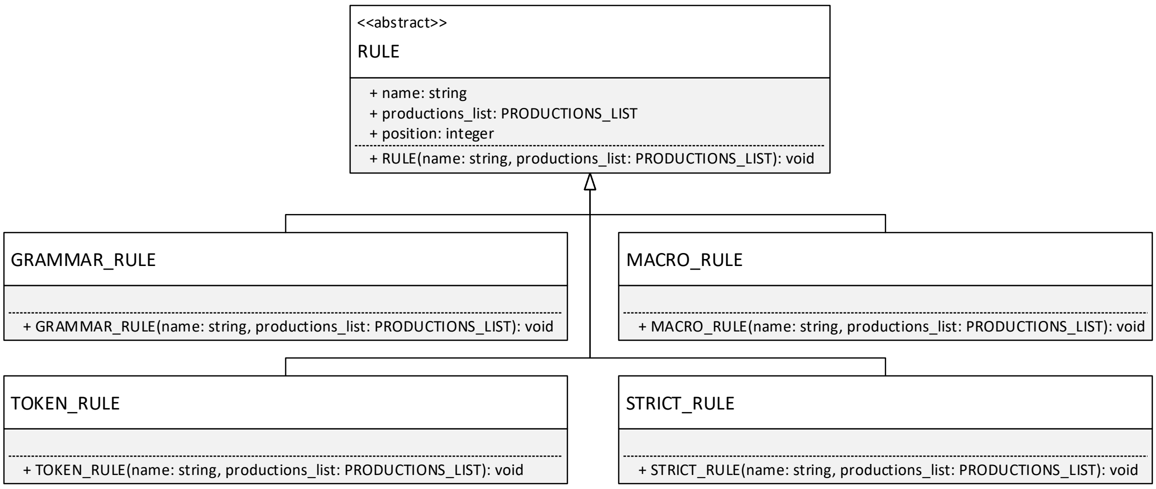

Figure 3.5: Rule data type UML class diagram

Comment block

A comment block is a list of consecutive comment lines.

todo describe diagram

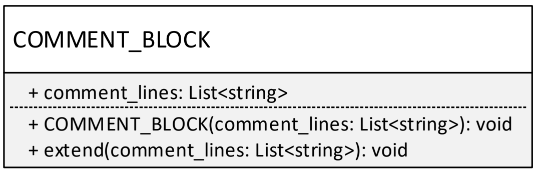

Figure 3.6: Comment block data type UML class diagram

Production element

A production element is either a terminal or nonterminal symbol. Additionally a production symbol has a production property.

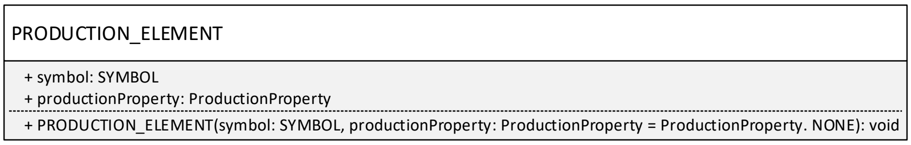

Figure 3.7: Production element data type UML class diagram

Production property

The production property can take one of three values and denotes whether a production is optional, can be repeated any number of times or does not have any special property. In the original TPTP syntax file this was represented by square brackets or the repetition symbol.

todo describe diagram

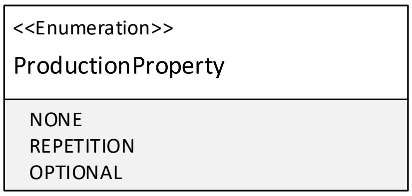

Figure 3.8: Production property data type UML class diagram

Production

A production is one production alternative specified in any expression. It consists of a list of production elements and has a production property. Productions can also be nested. Therefore the list can also contain further productions

-show example

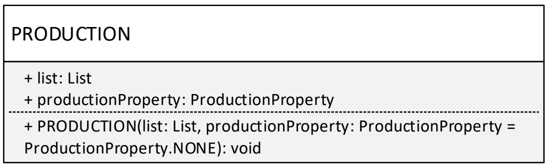

Figure 3.9: Production data type UML class diagram

Productions list

A productions list contains a list of productions where each production is one alternative in the description of an expression.

XOR Productions list

The XOR productions list represents multiple alternatives enclosed by parentheses. It contains a list of the alternate productions.

todo describe diagram

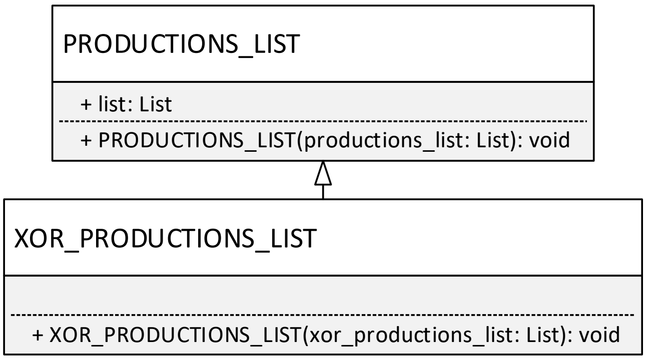

Figure 3.10: Productions list data type UML class diagram

Grammar list

The grammar list is the top level data structure. It contains a list of all elements that were in the TPTP syntax file. This includes any type of rules (grammar, token, strict and macro) and comment blocks.

todo describe diagram

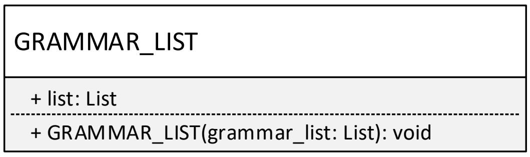

Figure 3.11: Grammar list data type UML class diagram

3.4.2 Production rules

When using the PLY parser generator, production rules have to be defined. The rules describe how the tokens are to be processed.

todo describe production rules

3.4.3 Disambiguation of square brackets

As mentioned before, 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 would then be interpreted as denoting the optional production property. This production property would then be 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 can be iterated, exchanging the production property optional by the square bracket terminal symbols.

todo vor- nachteile

The output of the parser is a list of the rules and the comments from the TPTP syntax file.

3.5 Graph generation

After parsing the TPTP grammar is stored in a grammar list. The grammar list data structure does not allow traversing which is why a new data structure is introduced that allows for modification and traversing. The data structure that is used is a graph representing the grammar.

The nonterminal symbols in the TPTP syntax are represented by a node class that has the following attributes:

- value: name of nonterminal symbol
- productions list: productions list of nonterminal symbol (todo 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, containing lists of all nonterminating symbols per production

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

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. 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.

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

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, the temporary start symbol is introduced before graph generation. This start symbol produces the start symbol that the user specified and is used as a starting point for the graph generation. If the before mentioned nonterminal symbol $< formula\_role >$ 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 nonterminal 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. 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 nonterminal symbol.

Since it is possible for a nonterminal 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 nonterminal symbol that has already been visited wont be visited again independent of circular dependencies.

The following example shows a production rule and the resulting list of children belonging to the node. Each production alternative has its own list of children.

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

3.6 Control file

In the following section a format for specifying the desired start symbol and blocked productions is described. Using a file-based configuration enables the user to store desired configurations and for example a manual selection in the graphical user interface is not necessary. It also helps with 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 nonterminal symbol and production symbol separating each group by a new line. First defining the nonterminal symbol, then the production symbol and after that the index of the alternatives that should be blocked (indexing starts at zero).

Identifying the production symbol is necessary because there may be a nonterminal symbol that has productions with more than one production symbol.

Listing 3.2 contains a sample control file. In this file in the first line $<TPTP\_file>$ is specified as start symbol. The second line means, that the second grammar production alternative of the nonterminal symbol $<TPTP\_input>$ should be disabled. Analogue to that, the first, second, third and fifth grammar production alternative of the nonterminal 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 3.2: Control file

This format is relatively easy to parse and also enables users to specify their desired start symbols and blocked productions without having to use the GUI.

pro: 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.

3.7 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, also the comment should be.
Therefore a mechanism has to be designed for the association of comments to grammar rules.

Listing 3.3 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.

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

Listing 3.3: Comment in the TPTP syntax

todo check if listing is handled correctly

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 sematically. todo maybe proof In listing 3.3 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 3.4. 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
8  %----used like "axiom"s.
9  %----"definition"s are intended to define symbols. They are either universally
10 %----quantified equations, or universally quantified equivalences with an
11 %----atomic lefthand side. They can be treated like "axiom"s.
12 %----"assumption"s can be used like axioms, but must be discharged before a
13 %----derivation is complete.
14 %----"lemma"s and "theorem"s have been proven from the "axiom"s. They can be
15 %----used like "axiom"s in problems, and a problem containing a non-redundant
16 %----"lemma" or theorem" is ill-formed. They can also appear in derivations.
17 %----"theorem"s are more important than "lemma"s from the user perspective.
18 %----"conjecture"s are to be proven from the "axiom"(-like) formulae. A problem
19 %----is solved only when all "conjecture"s are proven.
20 %----"negated_conjecture"s are formed from negation of a "conjecture" (usually
21 %----in a FOF to CNF conversion).
22 %----"plain"s have no specified user semantics.
23 %----"fi_domain", "fi_functors", and "fi_predicates" are used to record the
24 %----domain, interpretation of functors, and interpretation of predicates, for
25 %----a finite interpretation.
26 %----"type" defines the type globally for one symbol; treat as $true.
27 %----"unknown"s have unknown role, and this is an error situation.
28 %----Top of Page---------------------------------------------------------------
29 %----THF formulae.
30 <thf_formula>          ::= <thf_logic_formula> | <thf_atom_typing> |
31                            <thf_subtype> | <thf_sequent>
```

Listing 3.4: Comment lines split by a *Top of Page* line in the TPTP syntax

The flow chart in figure 3.12 shows the process of matching comment blocks, that are consecutive comment lines (see section 3.4.1), 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 and 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. todo why
- If this results in two comment blocks, one *Top of Page* line was present. Then the comment block before the *Top of Page* line is associated with the rule before when possible. If this comment block is at the beginning of the file it is associated with the rule after. The comment block after the *Top of Page* line is associated with the rule after. If it is at the end of the file it is 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. Therefore it is not particularly relevant. 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 3.12.

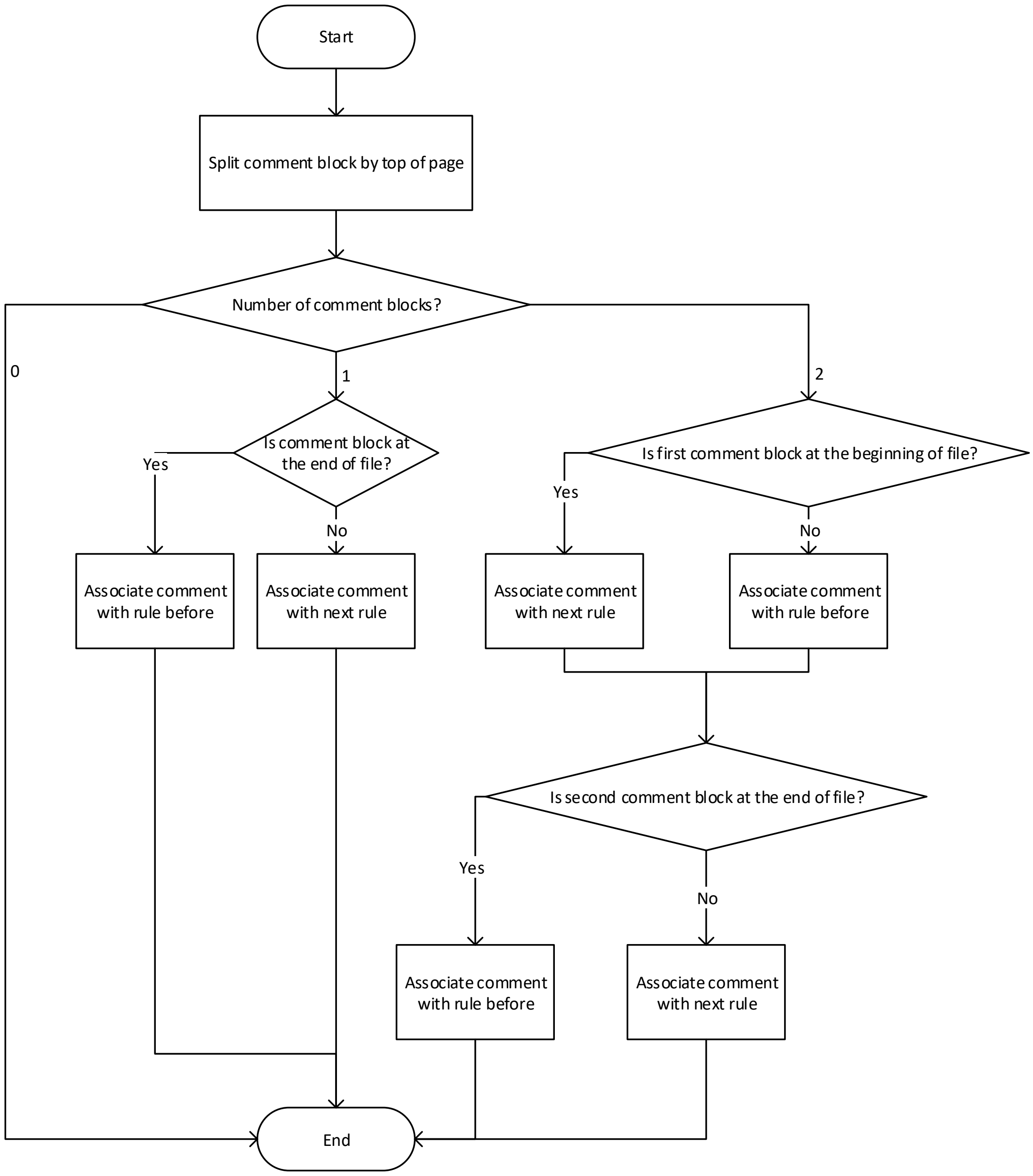

Figure 3.12: Maintaining comments flow chart

todo describe split comment block by top of page

todo explain could use content of comment

3.8 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 3.5) 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 reachable part of the grammar must be computed.
4. Starting from the still reachable part of the grammar, non terminating productions must be removed.

3.8.1 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 3.6. 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 3.8.4) whereas the information on the productions that should be blocked will be needed in the next section.

3.8.2 Removal of blocked productions

In the control file, for each nonterminal symbol, whose 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 adressed (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.

3.8.3 Determination of the remaining terminating symbols

After all reachable symbols have been determined, the last step of extracting a sub-syntax is to determine and remove the non-terminating productions and symbols. Non-terminating productions are productions in which at least one nonterminal symbol is non-terminating.

The terminating symbols can be determined recursively using dynamic programming techniques. The goal is to determine for each nonterminal symbol if it derives a terminating symbol. Only then it is a useful symbol since it is reachable and derives a terminating symbol todo reference Starting at the start symbol, the grammar graph is traversed.

-start with start symbol -traverse through graph and compute terminating symbols - store visited nodes to avoid infinite loops

3.8.4 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 3.5). All nodes that are not part of the new grammar graph are not reachable and can therefore be removed.

3.9 Output generation

Internally in the software tool a syntax is represented by a grammar graph. To make sub-syntaxes represented by a grammar graph usable 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 3.9.2.

3.9.1 Create output from grammar graph

There are three steps necessary in order for

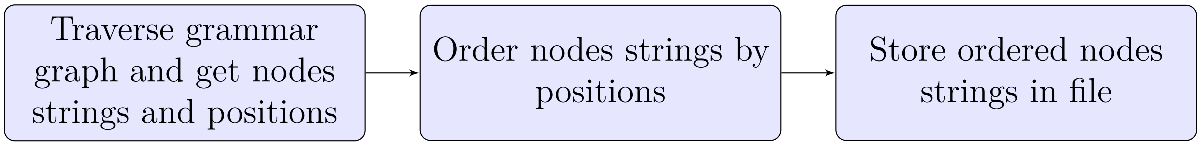

Figure 3.13: Procedure of generating a Syntax string representation from the grammar graph

- iterate over nodes and get string representation of nodes with node position - store node string with position - order node strings by position

- print to file -recursive

- each node represents rule of certain rule type with possibly multiple production alternatives

-recursively build strings

-maintain order of rules

-temporary start symbol pos -1 to not print

3.9.2 Automated parser generator compatibility

The automated parser generator for the TPTP syntax [14] takes a TPTP syntax file as input and creates a lex and yacc file corresponding to the specification in the input syntax.
In the TPTP syntax version 7.3.0 there is a definition of the syntax of comments which is shown in listing 3.5.

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

Listing 3.5: Comment syntax definition in the TPTP syntax

However this part of the syntax is not reachable from any other symbol. 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 ?todo check this?.
There are two ways to include the comment syntax in the output from the tool in order to be accepted by the automated parser generator. Either making the *<comment>* nonterminal symbol reachable by for example by adding it as an alternative to *<TPTP_input>* which can be seen in listing 3.6. Or maintaining the comment syntax separately and adding it to the output syntax even though it is not reachable.

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

Listing 3.6: Making the comment syntax reachable

Both ways are supported by the tool.
If the *<comment>* nonterminal symbol is made reachable in the TPTP syntax, the grammar graph generated by the tool contains the comment syntax and no further action is necessary.
To add the comment syntax to the output if the is *<comment>* nonterminal symbol is not reachable is done by using an external configuration file that contains the syntax from listing 3.5. 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 the tool when outputting a generated sub-syntax.

-Menu option

-comment symbol in lex

-make comment reachableto be tested

3.10 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 library Tkinter PyQt offers checkboxes in treeviews [12] that are used for selecting blocked productions.

todo In this section ...

-Tkinter and PyQt have been evaluated -Tikinter does not offer a kind of table view with checkboxes -which PyQt offers natively The graphical user interface should display the grammar similar to the original language grammar specification file. It should also be possible to make selections in the GUI instead of having to use a control file. -show rules similar to file Selection of a new start symbol and productions that should be possible in the GUI and also with the import of a control file.

The menu of the GUI consists of four submenus.

The *Import* menu that can be seen in figure 3.14 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 [13].

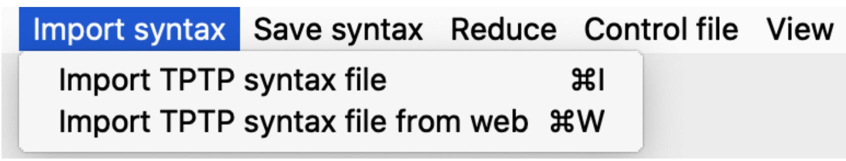

Figure 3.14: Import menu

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 3.15.

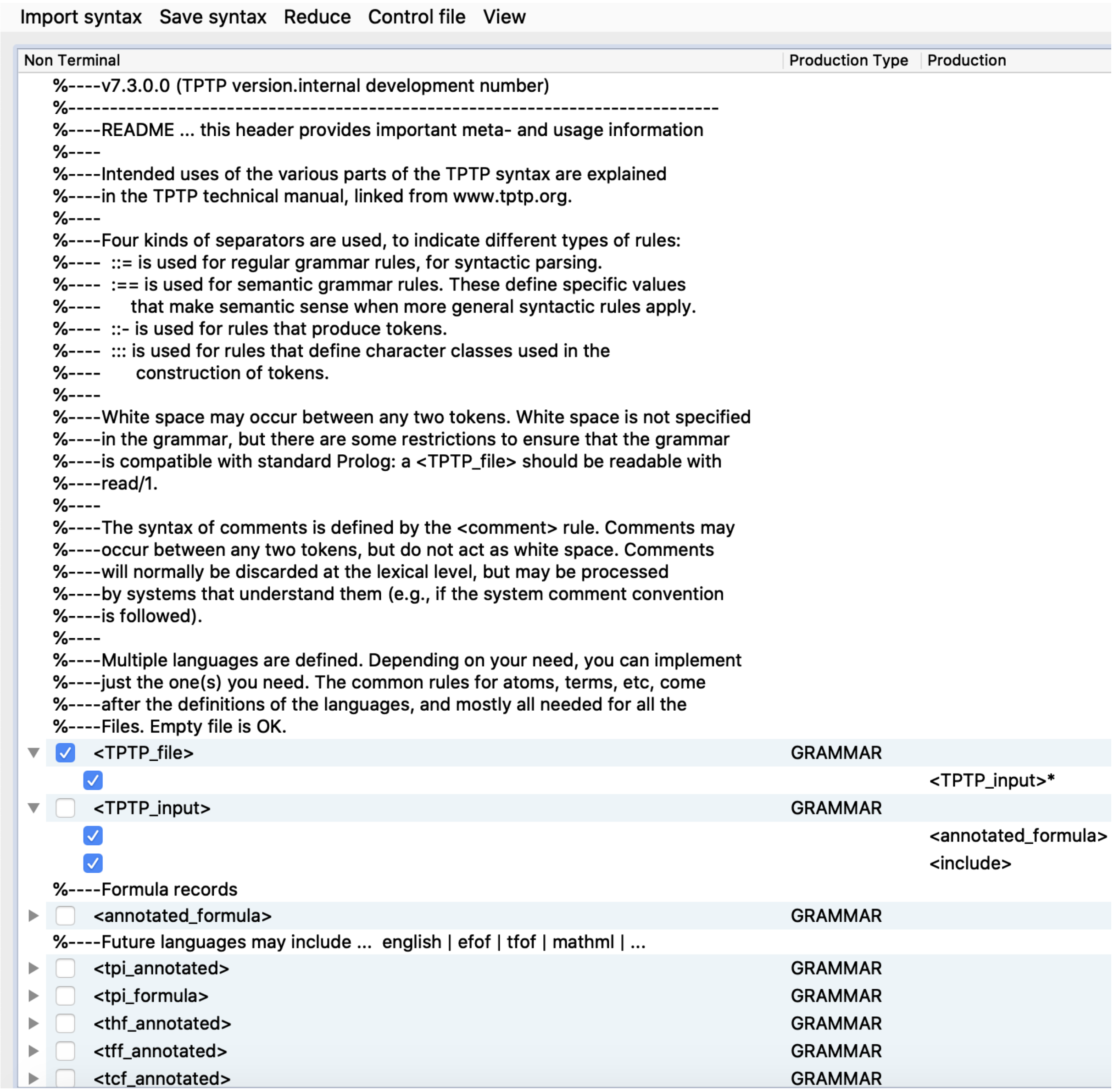

Figure 3.15: GUI

The *View* menu, shown in figure 3.16, provides the possibility to toggle the display of comments to improve readability.

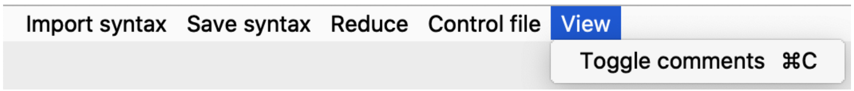

Figure 3.16: View menu

In order to extract a sub grammar 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 grammar 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.

In the *Reduce* menu that can be seen in figure 3.17 the user can reduce the grammar according to his selections. The grammar is reduced and the reduced grammar is displayed afterwards. The rules maintain the same order as in the original TPTP grammar.

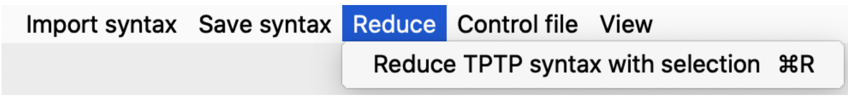

Figure 3.17: Reduce menu

In the *Save* menu that is shown in figure 3.18 the user can reduce the grammar 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 grammar to local storage and does not display it. If the user wishes that the $< comments >$, which is necessary for using the automated parser generator, production is including in the reduced grammar the user has the option to save the reduced grammar with the $< comment >$ production. The *Save* menu also gives the user the option to generate a control file based on his GUI selection. This is particularly helpful if the users is reducing the same grammar multiple times. This way he can use the import control file option later instead of selecting the blocked productions in the GUI tool.

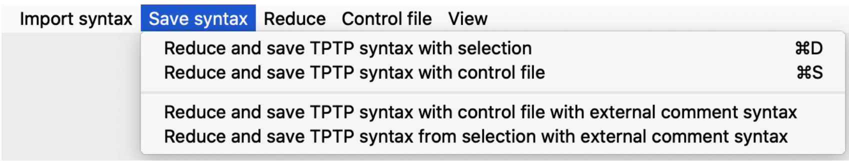

Figure 3.18: Save menu

todo

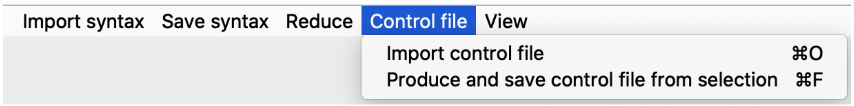

Figure 3.19: Control file menu

3.11 Command-line interface

The goal of the command-line interface is 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. Also basic help information is also accessible over the command-line interface.
Table 3.2 provides an overview about the command-line arguments. The syntax file location and the control file location has to be specified. Specifying an output path and file name is optional, by default the output filename will be *output.txt*. The $-ex$ flag enables additional output of the comment syntax (see 3.9.2. Additionally, the help description can be opend 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 **??**

todo include comment option

-more complex actions like control file generation can more comfortably be done by using gui -gui package not needed

todo count rules

cd /

4 Implementation

4.1 Lexer

As mentioned in chapter 3.3 the implementation of the lexer consists of the definition of tokens in form of regular expression. The following paragraph presents defined tokens and their regular expressions.

Ignored symbols

It is possible to declare symbols that should be ignored. However, if a symbol is declared as ignored but is specially mentioned in another token, then if the sequence of characters represent that token, the ignored symbol is not ignored. In this project, tabs and white spaces are ignored as they do not have any special meaning other than providing clarity. Also, newlines are generally ignored because as can be seen in listing 4.1 there are rules that cover multiple lines.

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

Listing 4.1: Multi line production rule

Apart from the ignored symbols, there are 13 defined tokens:

Expressions

Expressions can either be of the type grammar, token, strict or macro. It is defined as a nonterminal symbol followed by the production symbol itself (::=,:==,:::,...). The nonterminal symbol and the production are merged to a single token and are not identified as two tokens to avoid ambiguity while parsing. If not it would be difficult for the parer to determine whether the nonterminal symbol that describes the rule is the start of a new rule or does still belong to the previous rule because as mentioned rules can cover multiple lines.

Regular expression of grammar expression: $<\backslash w+>[\backslash s]*::=$

$\backslash w+$ matches any alphanumeric and underscore character that can occur more than one time. $[\backslash s]*$ matches an arbitrary amount of white spaces. $::=$ matches the symbol for grammar expression and can be substituted by any other symbol for the other expressions.

Nonterminal symbol

A nonterminal symbol starts with "<" and ends with ">". In between there is any arbitrary sequence of numbers, underscores and small or capital letters which can be represented by $\backslash w+$ in a regular expression.

Regular expression of nonterminal symbol: $<\backslash w+>$

Terminal symbol

Comment

A comment is identified by the lexer as a start of a new line followed by a percentage sign followed by an arbitrary character and ends with a newline. Because the percentage sign is also part of the terminal symbols, it is necessary to check whether the percentage sign is in a newline because the terminal symbol is not because the percentage symbol when used as terminal symbol is embedded in square brackets.

Regular expression of comment: $\hat{}\%.*\$$

Meta-Symbols

Meta-Symbols include open and close parentheses "( )", open and close square brackets "[ ]", asterisks "*"and vertical bars "|".

They are recognized by the symbol itself and have a greater meaning for the parser as they impact the to be build data structures.

Ambiguity

The following example could either be matched as one comment token or as comment, grammar expression, non terminal symbol, terminal symbol, non terminal symbol. This ambiguity is solved because by convention the lexer matches the longest possible token, the sequence of characters is matched as one comment.

```
%——    <formula_role> ::= <user_role>—<source>
```

Listing 4.2: Commented out production rule

4.2 Parser

The parser is taking the tokens from the lexer and matches them to defined production rules.

comment block reimplement equal operator

4.2.1 Data types

Figure 4.1 contains the UML modelling of the data types described in section 3.4.1.

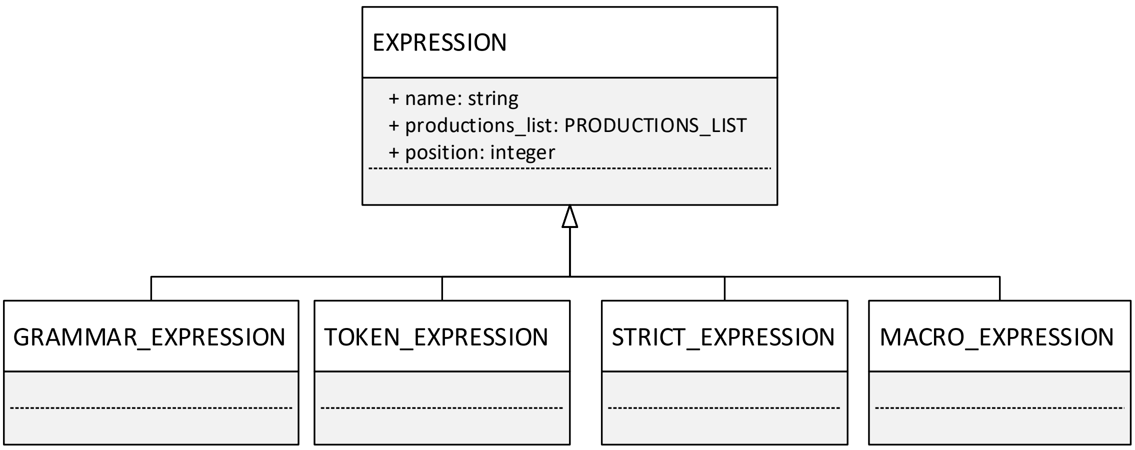

Figure 4.1: UML diagram for expressions

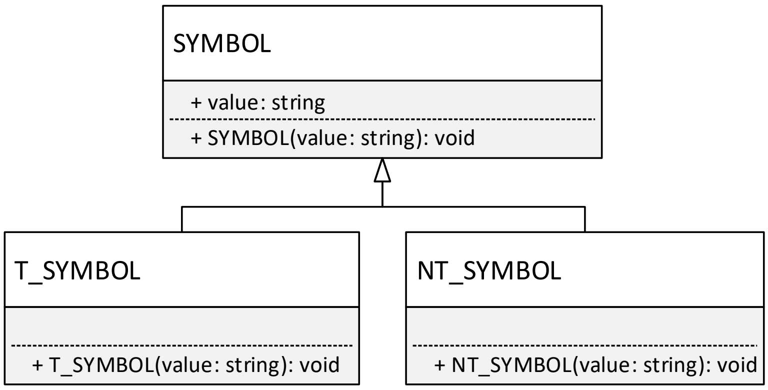

Figure 4.2: Symbols UML diagram

4.2.2 Defined grammar

- Input of parser is formal 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.

$\sum$ is the set of terminal symbols. Terminal symbols of the specified grammar are the tokens generated by the lexer (see 3.3).

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

A grammar list implies a comment block, the four expressions or a grammar list followed by 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 $< null >$::= has nothing on the right 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.

$$
\begin{aligned}
\textit{productions list} &\rightarrow \textit{production} \\
&\mid \textit{productions list}\ \ \textbf{alternative symbol}\ \textit{production}
\end{aligned}
$$

$$
\begin{aligned}
\textit{xor productions list} &\rightarrow \textit{production} \\
&\mid \textit{xor productions list}\ \ \textbf{alternative symbol}\ \textit{production}
\end{aligned}
$$

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

$$
\begin{aligned}
\textit{t symbol production} \rightarrow\ & \textbf{open square bracket}\quad \textbf{t symbol} \\
& \textbf{close square bracket} \\
& \mid \textbf{open square bracket}\quad \textbf{repetition symbol} \\
& \textbf{close square bracket} \\
& \mid \textbf{open square bracket}\quad \textbf{alternative symbol} \\
& \textbf{close square bracket} \\
& \mid \textbf{t symbol}
\end{aligned}
$$

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.

$$
\begin{aligned}
\textit{production element} \rightarrow\ & \textbf{open square bracket} \quad \textbf{nt symbol} \\
& \textbf{close square bracket} \\
& \mid \textbf{nt symbol} \quad \textbf{repetition symbol} \\
& \mid \textit{t symbol production} \ \ \textbf{repetition symbol} \\
& \mid \textbf{open square bracket} \quad \textbf{close square bracket} \\
& \mid \textbf{nt symbol} \\
& \mid \textit{t symbol production}
\end{aligned}
$$

$$
\begin{aligned}
\textit{production} \rightarrow\ & \textit{production element} \mid \textit{production} \ \ \textit{production element} \\
& \mid \textbf{open parenthesis} \ \ \textit{xor} \ \ \textit{productions list} \\
& \textbf{close parenthesis} \\
& \mid \textbf{open parenthesis} \ \ \textit{production} \ \ \textbf{close parenthesis} \\
& \mid \textit{production} \ \ \textbf{open parenthesis} \ \ \textit{production} \\
& \textbf{close parenthesis} \\
& \mid \textit{production} \ \ \textbf{open parenthesis} \ \ \textit{xor productions list} \\
& \textbf{close parenthesis} \\
& \mid \textbf{open parenthesis} \ \ \textit{production} \ \ \textbf{close parenthesis} \\
& \textit{production} \\
& \mid \textbf{open parenthesis} \ \ \textit{xor productions list} \\
& \textbf{close parenthesis} \ \ \textit{production} \\
& \mid \textbf{open parenthesis} \ \ \textit{production} \ \ \textbf{close parenthesis} \\
& \textbf{repetition symbol} \\
& \mid \textit{production} \ \ \textbf{open parenthesis} \ \ \textit{production} \\
& \textbf{close parenthesis} \quad \textbf{repetition symbol}
\end{aligned}
$$

- S: grammar list, start symbol is not mentioned, per convention by PLY first rule found (top level rule)

Listing 4.3 shows the production rule of the nonterminal $< tfxtuple >$ as well as the tokens that have been generated by the lexer.

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

Listing 4.3: Production element

The resulting parse tree can be seen in figure 4.3.

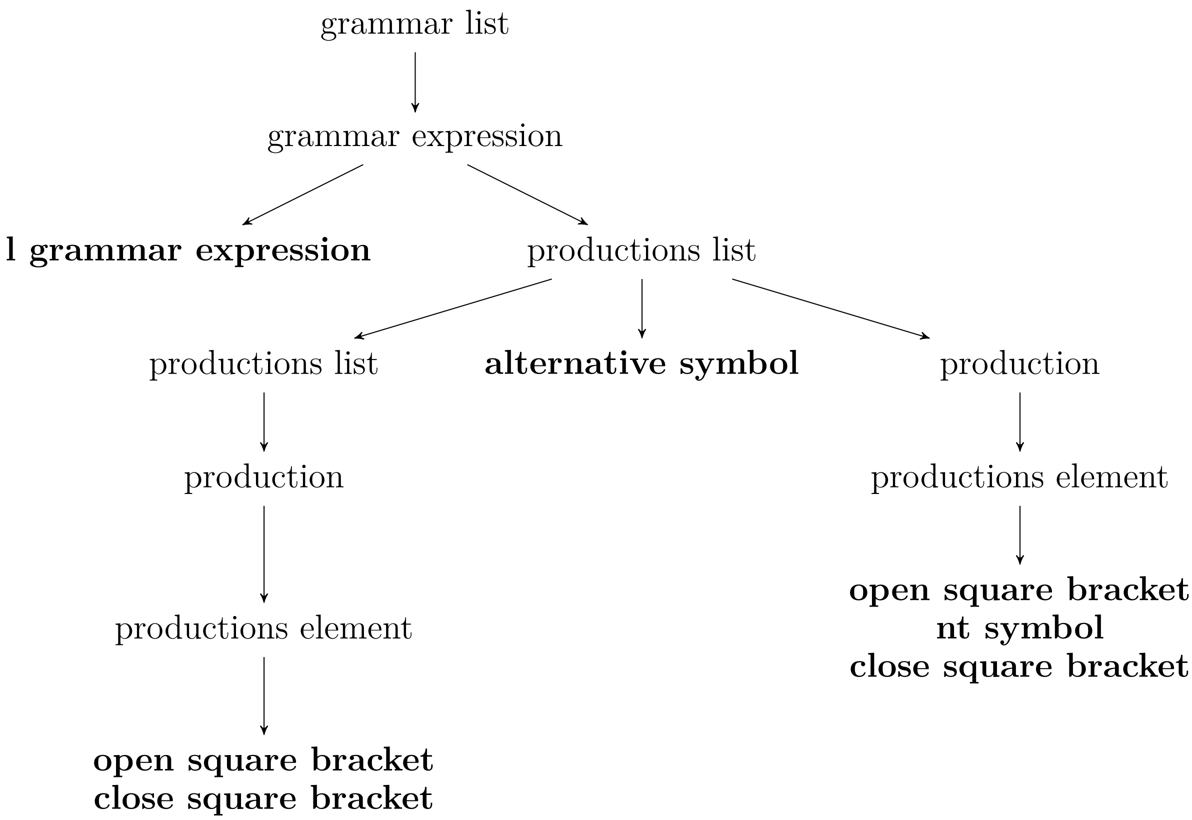

Figure 4.3: Parsing example

4.3 Graph generation

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, every child calls the algorithm resulting in appending their own children to their children's list.

Algorithm 1 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 through 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 who are then appended to the node.

Algorithm 2 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
```

The goal is to identify the nonterminal symbols. Therefore 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 3 Algorithm for appending children to node: searchProductionForNT

Input: node, productionsElement, children

```
for all elements in productionsElementList do
    if element is a production then
        searchProductionForNT(node, element, children)
    else if element is a XOR productions list then
        searchProductionsListForNT
    else if element is a nonterminal symbol then
        find element(s) in node dictionary
        append element(s) to children
    end if
end for
```

4.4 Extraction of a sub-syntax

The following sections describe the algorithms for the extraction of a sub-syntax based on the concept outlined in section 3.8

4.4.1 Removal of blocked productions

Algorithm 4 takes the text of the control file as an input argument (control file format description can be found in section 3.6) 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 of 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 integer. They are the indexes of the productions that should be removed. The elements are ordered in a descending order (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. ?todo verständlich? To delete the productions from the specified node, the node object has to be obtained. 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 4 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
```

4.4.2 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 5 shows the procedure of how to remove nonterminating symbols. First it has to be determined which symbols are terminating and which are nonterminating.

-one way: starting from terminal symbol, find nonterminal symbol that derives terminal symbol, then find nonterminal symbol that only produces terminating symbols -graph data structure can only be traversed top down not bottom up. -therefore start at start node, traverse graph, and find terminating symbols -initialise visited set -store known terminating symbols

A set of terminating nodes and a temporary set of terminating nodes are initialized. In the while loop the recursive algorithm *find_non_terminating_symbols(start_node, temp_terminating, visited)* finds terminating nodes with the start_node, and 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 nonterminating 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 5 Removing non terminating symbols

```
Input: start_node
terminating = new set()
temp_terminating = new set()
while True do
    visited = new set()
    this.find_non_terminating_symbols(start_node, temp_terminating, visited)
    if terminating == temp_terminating then
        break
    else
        terminating = temp_terminating
    end if
end while
visited = new set()
delete_non_terminating_productions(start_node, terminating, visited)
delete_non_terminating_nodes(terminating)
```

Algorithm 6 recursively determines whether a node can be identified as terminating. The input are a set of already known terminating nodes and nodes that already have been visited. If the node has been not been visited before it is added to the visited set. If it has been visited before, the method ends to prevent infinite recursion. If it has not been visited, the *children* list of the node is iterated. (todo children list contains list of nonterminal symbols per production) 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 certainly 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 4.4. If every node in the *children_list* represents a terminating symbol, the node is added to the known terminating nodes.

Algorithm 6 Find non terminating symbols

```
Input: node, terminating_set, visited_set
node_key = Node_Key(node.value, node.rule_type)
if node_key not in visited_set then
    visited_set.add(node_key)
    for all children_list in node.children do
        if len(children_list) == 0 then
            terminating_set.add(node_key)
        else
            terminating_flag = True
            for all child in children_list do
                find_non_terminating_symbols(child, terminating_set, visited_set)
                if not value_in_terminating(child.value, terminating_set) then
                    terminating_set = False
                end if
            end for
            if terminating_flag then
                terminating_set.add(node_key)
            end if
        end if
    end for
end if
```

The flow diagram in figure 4.4 shows the procedure to check whether a nonterminal symbol, described by its name, is in the set that contains information on which nodes are known to be terminating. 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).
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.

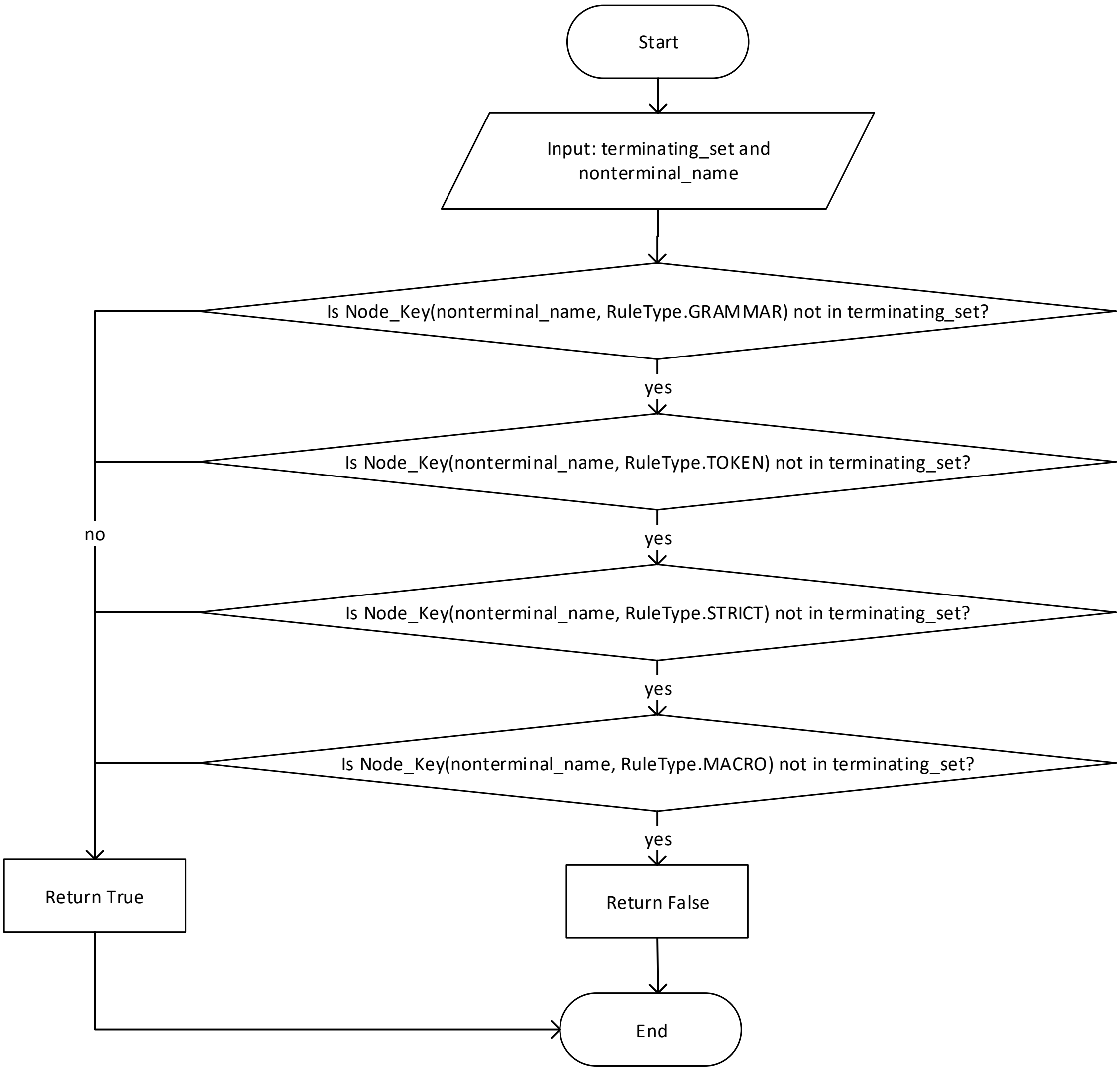

Figure 4.4: Check if node value is in terminating set

After the terminating symbols have been determined, productions that contain a nonterminating symbol have to be removed from the grammar graph. Algorithm 7 describes how productions that include nonterminating symbols are removed from the grammar graph. -Input

If the node has not been visited before it is added to the set of visited nodes. The *children* list is iterated and for each nested list: for each node the algorithm is called recursively. If a node represent a nonterminating symbol the *not_terminating* flag is set to true. If this flag is true after iteration of one element of the *children* list, it means that the production corresponding to the *children* list element contains at least one nonterminating symbol. Then, 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. -The *children* list is iterated from the end to the beginning in -index variable -index to end For each child the algorithm is called recursively. If the child node does not correspond to a terminating symbol, set nonterminating flag to true If the nontermiating flag is true, delete production -> delete children entry and productions list entry. -value in terminating checks if any ?rule type? is terminating

-call for each child in children list -if child not in terminating, set not terminating flag -if not terminating, delete corresponding children list and prouctions list

-todo check children list, ist was wenn symbol mehrere nodes hat?

Algorithm 7 Delete non terminating productions

Input: node, terminating_set, visited_set

```
node_key = Node_Key(node.value, node.rule_type)
if node_key not in visited_set then
    visited_set.add(node_key)
    index = len(node.children) - 1
    for all children_list in reversed(node.children) do
        not_terminating = False
        for all child in children_list do
            delete_non_terminating_productions(child, terminating_set, visited_set)
            if not value_in_terminating(child.value, terminating_set) then
                not_terminating = true
            end if
        end for
        if not_terminating then
            delete node.children[index]
            delete node.productions_list.list[index]
        end if
        index = index - 1
    end for
end if
```

Algorithm 8 removes the nonterminating nodes from the *nodes_dictionary*. It takes a set of *Node_Keys*s as input, that specifies all terminating nodes as input. To remove the nonterminating nodes, in a for-loop, all terminating nodes that are addressed by the *Node_Keys* are stored in a temporary dictionary. This dictionary replaces the original *nodes_dictionary* of the *TPTPGraphBuilder*.

Algorithm 8 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

4.4.3 Determination of the remaining reachable symbols

As mentioned in section 3.8.2 todo check reference the determination of the remaining reachable is done by creating a new grammar graph, with the desired start symbol from the control file. The nodes dictionary is also reduced to contain only the reachable nodes.

4.5 Output generation

4.5.1 Create output from grammar graph

4.5.2 Traverse graph to create string representations of objects

4.5.3 Create string representations of objects

- python has string class `https://docs.python.org/3/library/stdtypes.html#str` - returns string version of the passed object - it returns object.___str ___() when object has the method - following classes implement str method:

COMMENT_BLOCK -lines from lines list with newline

PRODUCTION_ELEMENT

PRODUCTION -order

NTNODE

4.5.4 Maintain original order

- use tuple of position and node string

4.5.5 Automated parser generator compatibility

-append option

4.6 Input/Output

4.7 GUI

The GUI consists of the class View that can be seen in Figure 4.5. 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 are described in the following.

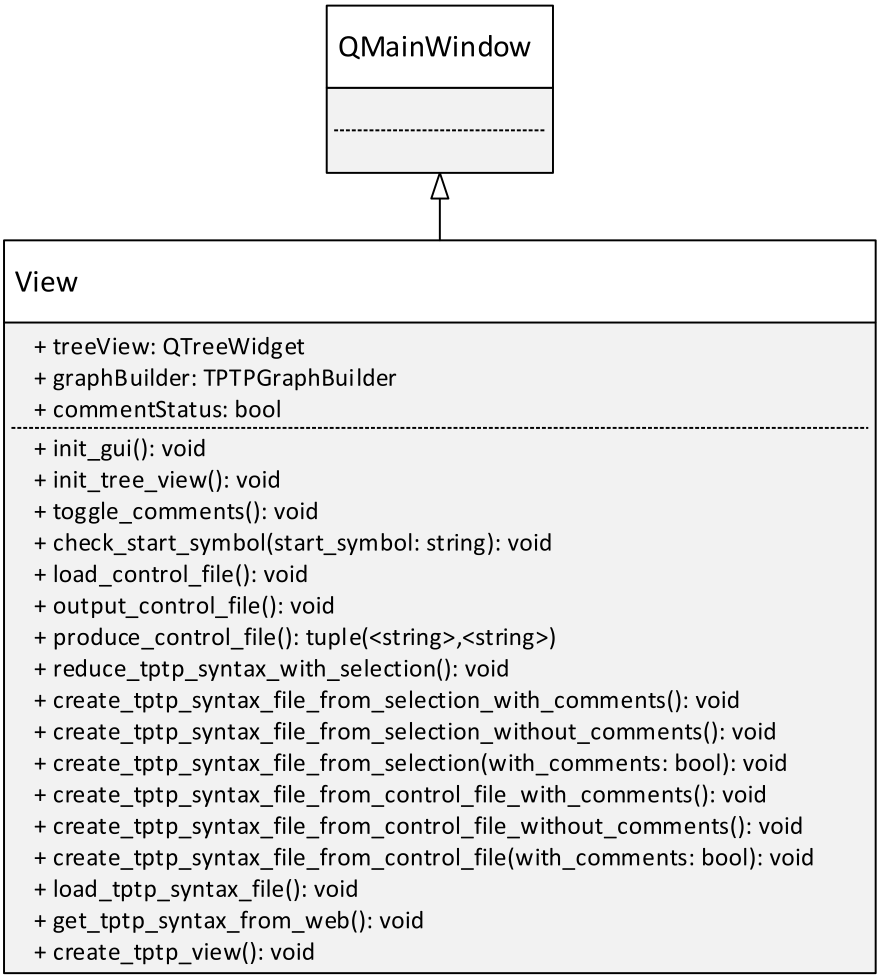

Figure 4.5: View UML class diagram

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 3.10 are added and implemented using QActions. The following listing 4.4 shows the Python code for adding a menu bar and menu actions for example for the "Import syntax" menu bar.

```python
import_menu = menubar.addMenu("Import syntax")
import_menu.addAction(import_tptp_file_action)
import_menu.addAction(import_tptp_file_from_web_action)
```

Listing 4.4: Implementation of menu bar

Besides a name, a short cut for executing the menu action is defined. Also triggers have been defined that triggers functions once menu button is pressed. Listing 4.5 shows how a short cut and a trigger have been defined for the action *ImportTPTPsyntaxfile*.

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

Listing 4.5: 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 grammar is stored in a nodes dictionary whose values (the nodes) are then converted to a list. This list is displayed by transforming it into a tree view. 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 4.6). 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 4.6. The pseudo code in 9 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 4.6: Init Tree View

Algorithm 9 GUI Pseudo Code: init_tree_view

```
for all nodes in nodes_list do
    rule_type = node.rule_type
    item = new QTreeWidgetItem
    Uncheck checkbox of item
    Set background to grey
    for all productions in node.productions_list do
        child_item = new QTreeWidgetItem
        Check checkbox of child item
        Add child_item to item
    end for
    if node.comment_block exists then
        comment_item = new QTreeWidgetItem
        Add comment_item to treeView
    end if
    Add item to treeView
end for
```

Check start symbol

This function checks the checkbox of the defined start symbol. The QTreeView method $findItem(string\ itemName)$ is used that returns all items that match the item name. The found items are than checked.

4.7.1 Display rules

Toggle comments

Algorithm 10 shows the algorithm used for toggling comments. The algorithm should either show or hide the comments based on the current state. 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.

-flag(), bitweises AND

Algorithm 10 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
```

Reduce TPTP syntax with selection

For reducing the TPTP syntax based on the users selection in the GUI a control file is produced (see Produce Control file). If no grammar has been imported or no start symbol or multiple start symbol have been selected an error raises. 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 4.7).

```
control_string, start_symbol = self.produce_control_file()
self.graphBuilder.disable_rules(control_string)
self.init_tree_view()
self.check_start_symbol(start_symbol)
```

Listing 4.7: Reduce TPTP syntax with selection

Import control file

- open control file via QFileDialog
- check if not none - uncheck all left symbols, check all right symbols, separate via has item parents
- read first line in control file, set start symbol accordingly

- for each following line, read nt name and ruletype and search item and uncheck it

Algorithm 11 GUI Algorithm: 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 12 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 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 parents 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 have not he 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 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 12 GUI Algorithm: produce_controlfile

```
Define new tuple Entry("Entry", ["value", "rule_type"])
Create empty dictionary
Create empty list start_symbol
for all items in treeView do
    Get parent of item
    if item has parent and is unchecked then
        Get rule type of parent
        Create entry tuple with name of parent and its rule type
        Get index of child from parent
        if entry tuple is not in dictionary then
            Create new dictionary entry with entry tuple as key and list containing the index of the child as value
        else
            Append index of child to list
        end if
    else if item has no parent and is checked then
        Append item name to start symbol list
    end if
end for
control_string = first entry of start symbol list
Append all entries of dictionary to control_string
```

Import TPTP syntax file

The TPTP syntax file is imported using QFileDialog which is returning the filename. If the filename is not empty, the application opens a dialog window where the user has to specify a desired start symbol. The method *red_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*.

4.8 Command-line interface

With the argparse module a command-line parser object can be created and parameters can be added to that object. In the first line of listing 4.8 the command-line parser object is created with a description. Lines two to four 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*. 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.

todo listing describe last argument

```
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', action='store_true', help="flag - include external comment syntax")
```

Listing 4.8: Argparse command-line parser configuration

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

-options for path to grammar and control file -option for output path

-todo describe seperation of gui and why

5 Validation

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

5.1 Comment association

comment association

5.2 Automated parser generation

A goal of using the software tool is to be able to use an extracted sub-syntax with the automated parser generator for the TPTP syntax [14]. 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 3.9.2). 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.

5.2.1 Building a basic parser

To demonstrate the usability and capability of the tool 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 to use the software tool.
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.

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

Listing 5.2: *annotated_formula* production rule

todo describe annotations 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 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.

5.2.2 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 the tool, for example sub-syntaxes with FOF, or FOF and CNF.

5.3 Syntax size comparison

original number of rules

cnf number of rules

6 Conclusion

6.1 Future Work

It is conceivable to extend the tool to apply it on other languages, e.g. SMT-lib [**BFT-SMTLIB-17**]. Based on their grammar, the lexing and parsing component have to be adapted to parse the given grammar. Remaining components can be used with only minor modifications. The graph generation for example needs to be adapted depending on the amount of production symbols. Also comment splitting has to be adapted based on the specific comment convention.

comment association

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

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

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

[5] 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.

[6] 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.

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

[8] 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] 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.

Online sources

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

[10] 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).

[11] 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).

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

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

Appendix