Cutting Languages Down to Size

Student Project

at the Cooperative State University Baden-Württemberg Stuttgart

by

Nahku Saidy and Hanna Siegfried

08.06.2020

Time of Project	07.10.2019 - 08.06.2020
Student ID; Course	8540946, 6430174; TINF17ITA
Advisors	Prof. Dr. Stephan Schulz and Prof. Geoffrey Sutcliffe, Ph.D.

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 classic logics ranging from first-order clause normal form (CNF) to typed first-order logic (TFF) have been included in and extended the TPTP language.
In this report, we describe a tool that is able to automatically extract sub-languages from the TPTP language. Sub-languages of interest are for example CNF or full first-order logic (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. Therefore, 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 should be maintained and associated with the corresponing rules in the reduced syntax.

1.2 Structure of the Report

The report is structured into six chapters. 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 describes a set of words belonging to a language. These words are built over the alphabet of the language.

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 from 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. While finite automata also describe languages, automata cannot denote languages. 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 among 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 symbols. 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 $\xi \overset{*}{\rightarrow} z$ so that ξ can be derivated to z 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 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 $S \overset{*}{\rightarrow} u\xi v$ 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 symbol containing ξ. [4]

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 $TPTP\_File$ ::= $<TPTP\_Input>$ | $<comment>$.
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 so 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 subset of another token. For solving such conflicts 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 multiple tokens fit the input. [7]
Furthermore, a lexer is separating the input in order to divide it into tokens. Per convention the longest input that matches any token is chosen. [7]

Instead of writing a lexer manually it is often 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. Therefore 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 Python

2.6.1 PLY

Python Lex-Yacc (PLY) [8] is an implementation of lex and yacc in python. [LALR-parsing] consists of lex.py and yacc.py

lex.py tokenizes an input string

2.6.2 PyQt

PyQt is a Python binding for the cross-platform GUI framework Qt [9]. It is licensed under the GNU GPL version 3.

tkinter

2.6.3 argparse

The python module argparse is a module for creating command line interfaces. It provides the means to specify input arguments and [10] automatically creates help and usage messages. It also checks if the given arguments are valid. After specifying 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.

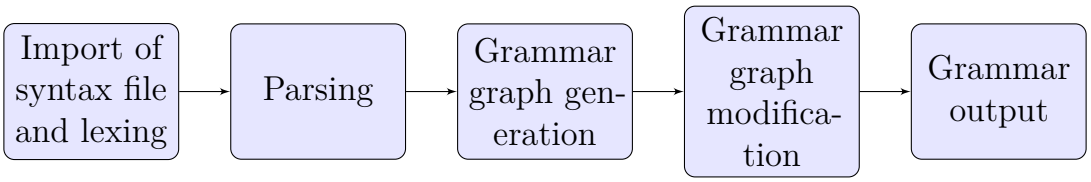

Figure 3.1: Procedure of extracting a sublanguage

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.

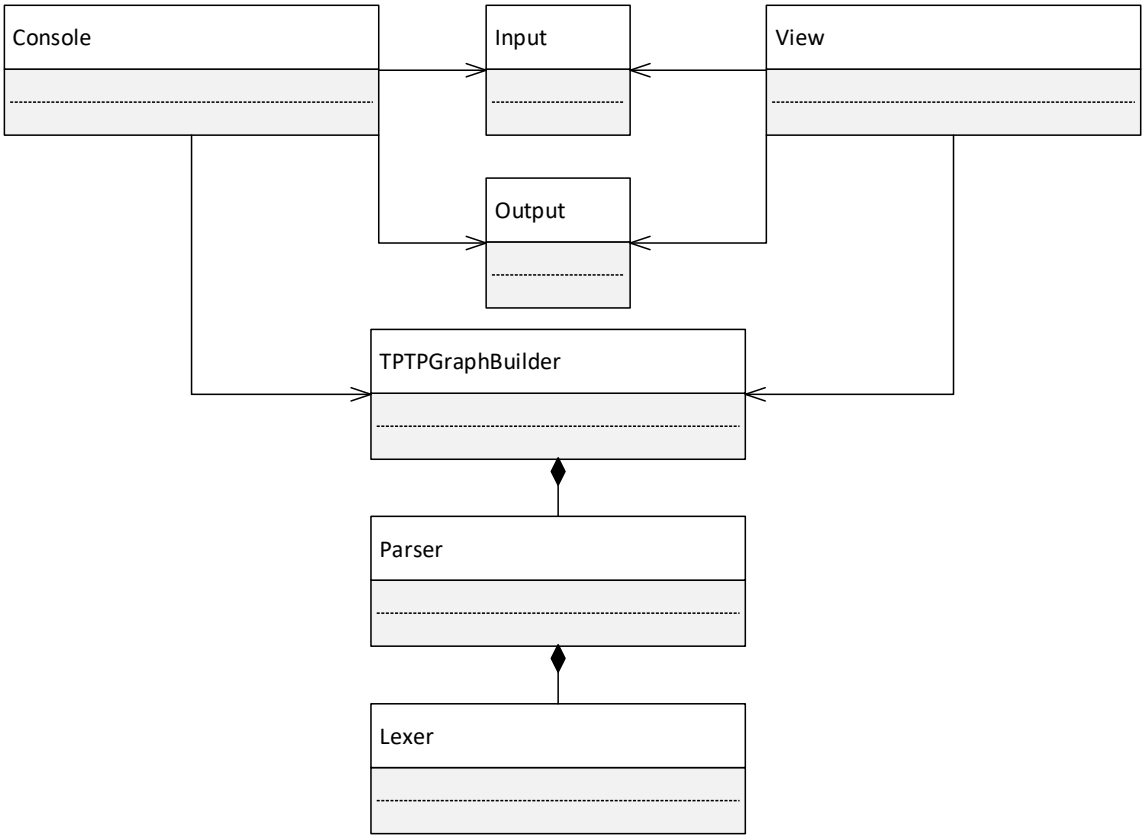

Figure 3.2: UML diagram of the architecture of the software tool

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.

3.3.1 Elementary tokens

The syntax of the TPTP language is specified in a modified EBNF **??**. Therefore there are deviations from standard EBNF (see 2.3) that need to be analysed to specify elementary tokens. The standard EBNF only uses one production symbol (" ::= "). In the TPTP syntax additional production symbols have been added. The following table 3.1 contains the production symbols used in the TPTP syntax, that also have to be recognized by the lexer.

Table 3.1: TPTP language production symbols [11]

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

Another deviation from EBNF is that repetition is not denoted by surrounding curly brackets, but with a trailing $*$ symbol.
Curly brackets have no special meaning in the TPTP syntax and can be treated as terminal symbols.
The meaning of the alternative symbol | is unchanged and also parentheses and square brackets can appear as meta symbols.
Also, there are line comments in the TPTP syntax. A comment starts with the % symbol at the beginning of a line and ends at the end of that line.
Following standard BNF, nonterminal symbols are enclosed by the $<$ and $>$ symbol and terminal symbols are written without any special marking.

3.3.2 Tokens

This section introduces the token, their regular expression are described in chapter 4.1.

It is possible to declare symbols that should be ignored. However, if a symbol is declared as ignored but is specifically 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.

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

Listing 3.1: Example of a multi line production rule

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

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 non terminal 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.

A *nonterminal* symbol starts with <and ends with >. In between there is any arbitrary sequence of numbers, underscores and small or capital letters.

terminal symbol

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 the first character of a newline because the terminal symbol is not because the percentage symbol when used as terminal symbol is embedded in square brackets.

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.

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. In token and macro rules they have the same meaning as in traditional EBNF and in grammar and strict rules square brackets are terminals. Therefore disambiguation of square brackets is necessary.

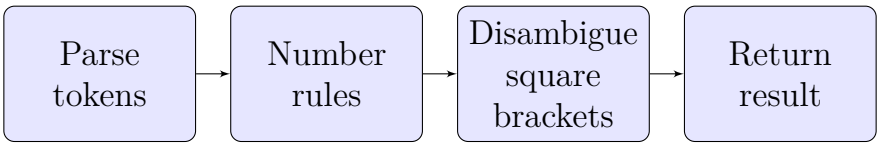

Figure 3.3: Parsing procedure

3.4.1 Data structure 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.

Atomary data types

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.

Composite data types

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 3.2 contains an example of a line in a TPTP syntax file that is represented by the grammar rule data type. The nonterminal symbol name which is produced is $<tff\_formula>$. The production list consists of two productions, as can be seen in the listing.

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

Listing 3.2: Example of a grammar expression

Comment block

A comment block is a list of consecutive comment lines.

Production element

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

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.

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

Production property

The production property represents three options. A production and a production element is either allowed to be repeated multiple times, is optional or does not have any special production property.

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

Multiple alternatives enclosed by parentheses.

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.

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.

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

The TPTP syntax, extracted from the TPTP syntax file, needs to be stored in a data structure that allows for modification and traversing. The data structure that is used is a graph consisting of nodes. Every nonterminal symbol that is on the left side of a production in the TPTP syntax is represented by a node and instantiates a defined class "Node" that has the following attributes:

- value: name of nonterminal symbol
- productions list: productions list of nonterminal symbol (REFERENCE)
- 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: list containing all children of a node TODO A child is ..

Starting with the start symbol, the graph is generated recursively. Iterating over each nonterminal symbol that is on the right side of a production rule, the corresponding node is identified. These nodes are then appended to the list of children of the nonterminal on the left side of the rule. The identified children may again have children. This process is repeated until a node has no children because there are only terminal symbols on the right side of the production rule.

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.

Furthermore, the start symbol can have multiple rule types resulting in two start symbols. As it is not possible in the application to have two start symbols, a new start symbol is generated that implies the two original start symbols.

The following example shows a production rule and the resulting list of children belonging to the node. Each production alternative(REFERENCE) 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

A format for specifying the desired start symbol and blocked productions has to be developed. 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 3.3 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 3.3: Example of a 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 Maintainig 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 of 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 3.4 features an example of a comment in a TPTP syntax file. This comment begins with a *Top of Page* line which, in the HTML version of the TPTP syntax, contains a hyperlink which leads to the beginning of the syntax file. The next line contains a relevant comment.

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

Listing 3.4: Example of a 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 3.4 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 3.5. 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 3.5: Example of comment lines split by a *Top of Page* line in the TPTP syntax

heuristic: comments near the rule they refer to associate comment with r

The flow diagram in figure 3.4 shows the process of matching comments to rules. First comment blocks, that are consecutive comment lines (see section 3.4.1) are split into multiple separate comment blocks by using *Top of Page* lines as separators

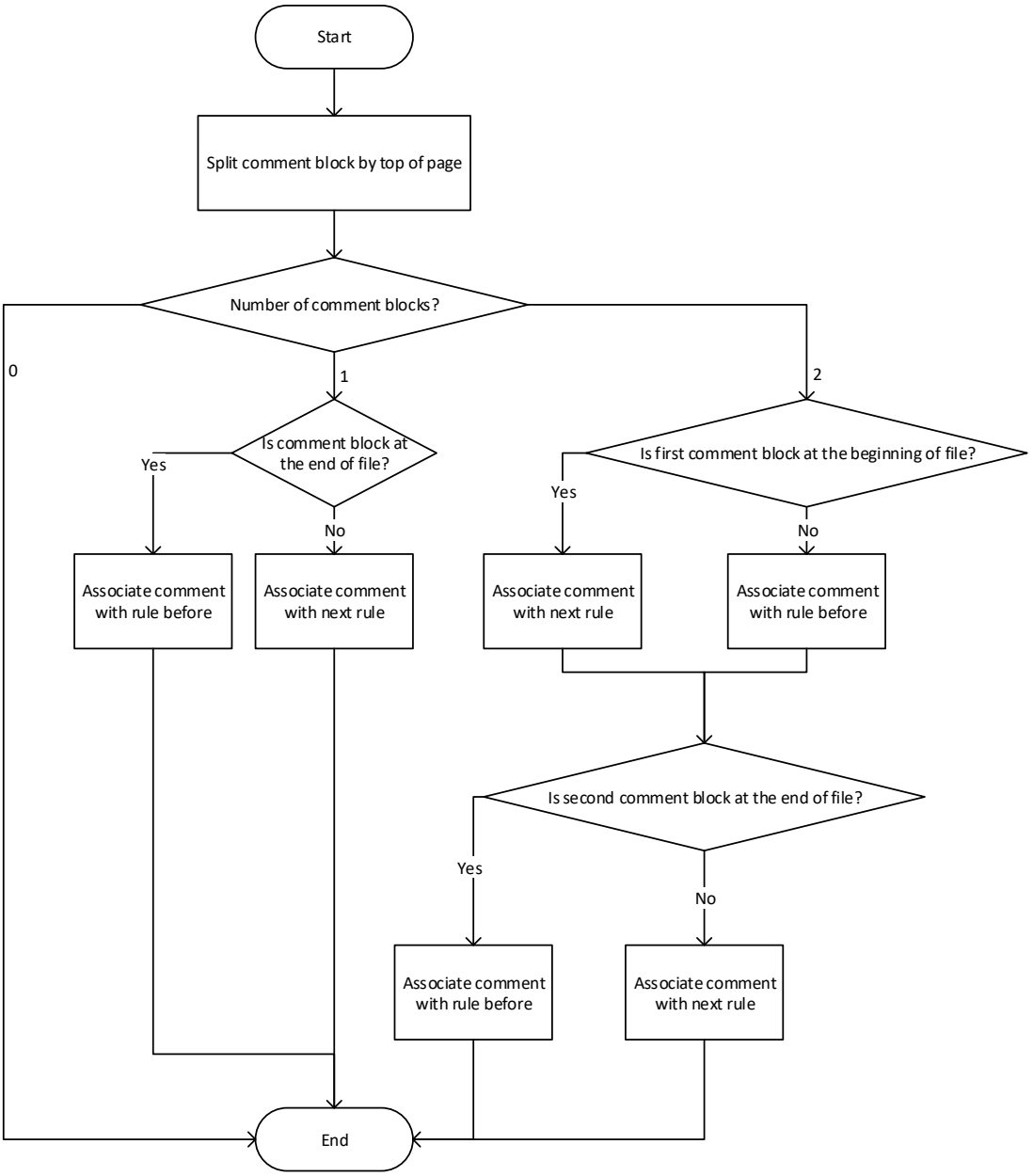

Figure 3.4: Maintaining Comments

3.8 Generation of a sub-syntax

This section covers the concept of how a sub-syntax can be 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 grammar should be extracted is specified in the control file. For that, three steps must be performed:

1. The blocked productions specified in the control file must be disabled and therefore the corresponding transitions must be removed from the grammar graph.
2. The remaining reachable part of the grammar must be computed.
3. Starting from the still reachable part of the grammar, non terminating productions must be removed.

bei tree building temporäres startsymbol nutzen (da mehrere Startsymbole möglich)

3.8.1 Removing of blocked productions

3.8.2 Determination of the remaining reachable productions

dynamic programming

3.8.3 Determination of the remaining terminating productions

3.9 GUI

In this section ... 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.

-show extracted grammar -export exported grammar -include + toggle comments -> algorithm -import/export control file extra: -web import

3.9.1 Menu

3.9.2 Rules display

3.10 Command-line interface

The goal of the command-line interface is to provide means for convenient automation of sub-syntax extraction. It should take a TPTP syntax file and a control file as input and output the resulting sub-syntax. Also basic help information should be accessible over the command-line interface.

-command-line interface allows for automation with scripts for repeated tasks -basic functionality extract sub-syntax by providing base syntax file and control file, -output sub-syntax with input control file -provide basic information with help menu -more complex actions like control file generation can more comfortably be done by using gui

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.

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

Listing 4.1: Example of a 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.

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.

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.

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.

```
1 %----    <formula_role> ::= <user_role>-<source>
```

Listing 4.2: Example of a 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.

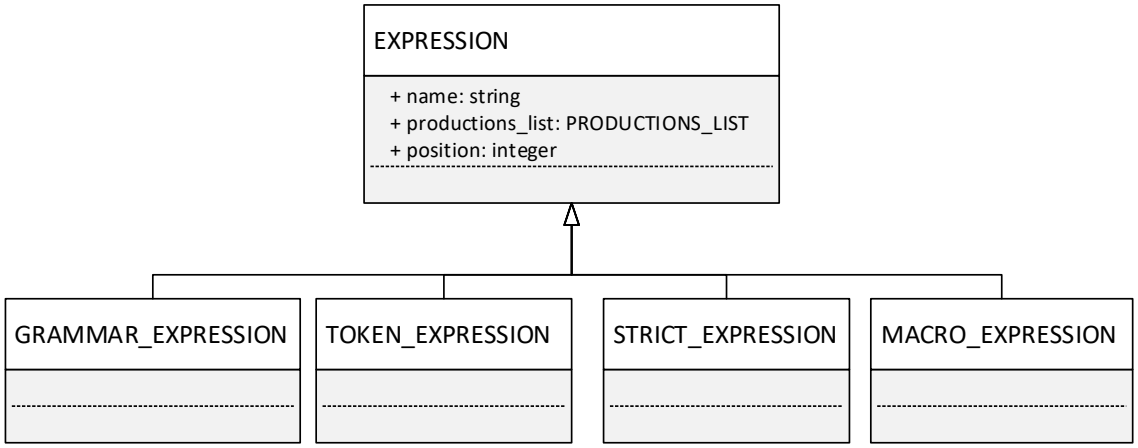

Figure 4.1: UML diagram for expressions

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

```
1: searchProductionsListForNT(node, node.productionsList)
2: if node has children then
3:     for all children do
4:         buildGraphRek(child)
5:     end for
6: 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

```
1: for all elements in productionsList do
2:     children = new empty list
3:     searchProductionForNT(node, element in productionsList, children)
4:     append children to node
5: 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

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

4.3.1 Removing of blocked productions

Algorithm **??**

Algorithm 4 Removing blocked productions

Input: control_string

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

4.3.2 Determination of the remaining reachable productions

Removing non-terminating symbols

Algorithm 5 Removing blocked productions

Input: start_node

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

4.4 GUI

Tkinter and PyQt have been evaluated as a basis for the GUI. -pyqt offers checkboxes in treeviews

4.5 Command-line interface

For the command-line interface (see section 3.10) the python module argparse is used. -argparse Python offers a library for command-line interfaces. -options for path to grammar and control file -option for output path

5 Validation

back to back testing show advantages and useful for tptp users...

comment association

show before after size

5.1 Automated Parser Generation

6 Conclusion

6.1 Future Work

comment association

Bibliography

Publikationen

[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.

[11] 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 Quellen

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

[9] 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).

[10] 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).