
# 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** | first-order clause normal form |
| **EBNF** | extended Backus-Naur form |
| **FOF** | full first-order logic |
| **PLY** | Python Lex-Yacc |
| **TFF** | typed first-order logic |
| **THF** | typed higher-order logic |
| **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 application cases are covered. On the one hand that leads to a more powerful language. However, on the other hand it becomes harder to understand the language and to implement it. Thus, it becomes harder for new users to use the language.
This problem can be addressed by extracting smaller sub-languages for particular use cases. This could be done manually, but using this method is likely to raise errors or divergences from the original grammar.
Therefore, the approach considered in this report is to develop an application that is able to automatically extract sub-languages from a language. A sub-language should be specified by the user using the application.
This report focusses on the Thousands of Problems for Theorem Provers (TPTP) language for automated theorem proving. Sub-languages of interest are for example a grammar just for first-order clause normal form (CNF) or full first-order logic (FOF) The grammar of the language is provided in an extended Backus-Naur form (EBNF). The first step to extract a sub-language is to build a lexer takes the grammar of the TPTP language as input and generates tokens. Subsequently a parser should build a parse tree that represents the grammar rules of the TPTP language. This parse tree should be visually presented to the user and the user can then choose which grammar rules should not be included in the desired sub-language. After the user specified the sub-language, the developed application should extract the sub-language from the TPTP language and present the sub-language 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

This report is structured into six chapters. In the first chapter the research problem and goals of this research are stated. Then in chapter 2 the necessary background information for the following chapters 3 and 4 is provided. In chapter 3 the concept for the software tool is developed. Based on this, the implementation of the application is featured in chapter 4. TODO In chapter 5 the results of the TODO reduced grammar is tested on a problem which is presented in a form corresponding to the reduced grammar. Chapter 6 sums up the results achieved in this research and offers an outlook for possible future research.

# 2 Background and Theory

[**Mogensen.2017**] Compiler: Translate (high-level) programming language into machine language

Different phases for writing a compiler, phases are processed sequently

## 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 grammar is specified in an EBNF. [**Sut17**]

## 2.2 Backus-Naur form (BNF)

The Backus-Naur form (BNF) is a language to describe context-free grammars. In the Backus-Naur form (BNF) nonterminal symbols are distinguished from terminal symbols by being enclosed by angle brackets, e. g. $<TPTP\_File>$ denotes the nonterminal symbol $TPTP\_File$. Productions are described using the " ::= " symbol and alternatives are specified using the "|" symbol. [**BNF.1964**] 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 sorrounded by square brackets.
- repetition is denoted by curly brackets.
- parentheses are used for grouping.
- terminals are enclosed in quotation marks.

[**EBNF.1977**]

Different phases for writing a compiler, phases are processed sequently

## 2.3 Grammar

Unlike regular expressions, grammars not only describe a language but also define a structure among the words of a language(?)(->additionally defines structure on the strings in the language it defines). Lexing or a so called lexical analysis is the division of input into units so called tokens [**LexYacc.1992**]. The input is a string containing a sequence of characters. Often, the lexer is generated by a lexer generator and not written manually. When writing the lexer manually TODO
A lexer generator takes a specification of tokens as input and generates the lexer automatically. The specification is usually written using regular expressions. A regular expression describes a formal language and can be described by a finite automata. A formal language describes a set of words belongig to the language. These words are built over the alphabet of the language. Shorthands are common to simplify a regular expression. For example all alphabetic letters in lower and upper case are combined and represented by [a-zA-Z]. The same principle can be also be applied to represent a set of numbers. However, using not clearly defined intervals e.g. [0-b] is not common as it has different interpretations by different lexer generators and thus can lead to mistakes. [**Mogensen.2017**]

A grammar is a list of rules that defines the relationships among tokens [**LexYacc.1992**]. 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 build 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.

### Reduced Grammar

Grammars are called reduced if each nonterminal symbol is terminating and reachable [**Cremers75**].

Given the set of terminal symbols $\sum$, a nonterminal symbol $\xi$ is called terminating if there are productions $\xi \underset{\rightarrow}{*} z$ so that $\xi$ can be derivated to $z$ and $z \epsilon \sum^*$. In other words, a nonterminal symbol $\xi$ is terminating if there exist production rules so that $\xi$ can be replaced by terminal symbols. [**Cremers75**]

Given the set of terminal symbols $\sum$ and the start symbol $S$, a nonterminal symbol $\xi$ is called reachable if there are production rules $S \underset{\rightarrow}{*} u\xi v$ so that $S$ can be derivated to $u\xi v$ and $u, v \epsilon \sum^*$. In other words, a nonterminal symbol $\xi$ is reachable if there exist production rules so that the start symbol can be replaced by a symbol containing $\xi$. [**Cremers75**]

### Context-free grammar

## 2.4 Lexer

Lexing or a so called lexical analysis is the division of input into units so called tokens [**LexYacc.1992**]. 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 e.g. **??**. 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. [**Mogensen.2017**]

A simple approach to build a lexer to to build an automata for each token definition and then test to which automata the input correspondends. However, this would be slow as all automatas need to be passed through in the worst case. Therefore, it is convenient to build 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.

It is possible, that final states overlap as a consequence of one token being a subset of another token. For solving such conflicts a precendence of tokens can be declared.

Usually the token that is being definded the earliest has a higher precende and thus will be chosen if multiple tokens fit the input. [**Mogensen.2017**]

Another task of the lexer is seperating the input in order to divide it into tokens. Per convention the longest input that matches any token is chosen. [**Mogensen.2017**]

## 2.5 Parser

Building a syntax tree out of the generated tokens [**Mogensen.2017**]

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 grammar. The output is the generated parser. [**LexYacc.1992**]

### Syntax analysis

Parsing: establish relationship among tokens [**LexYacc.1992**] Grammar: list of rules that defines the relationships [**LexYacc.1992**]

In this phase a parser will take a string of tokens and form 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 [**LexYacc.1992**] Output: parser [**LexYacc.1992**]

-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 PLY

Python Lex-Yacc (PLY) [**PLY**] 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 Python?

# 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.

why python

## 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 should be able to import a grammar file. After the grammar file is imported, the grammar file should be displayed. This includes displaying by the grammar defined productions as well as comments that are associated with grammar productions. The user can select a new start symbol and can select which productions should be blocked. Productions can either be blocked as a whole or partly if the production is a disjunction. After the user made his choice, the new reduced grammar should be generated and displayed. The tool should also generate a control file listing blocked productions and the start symbol. Furthermore, the tool should be able to import a control file and reduce a given grammar based on this control file instead of reducing a grammar based on a users selection of blocked productions. The new reduced grammar should be exported to .txt format. Also, comments referring to the remaining productions should be kept and comments referring to productions that fell away should not be included in the new grammar.

## 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 language grammar specification file and extract the tokens 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 grammar file was correct. Then, a graph representing the imported TPTP grammar 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 grammar of the extracted language. To make this grammar usable, lastly the language specification has to printed, based on the new graph, in the same format as the original language specification.

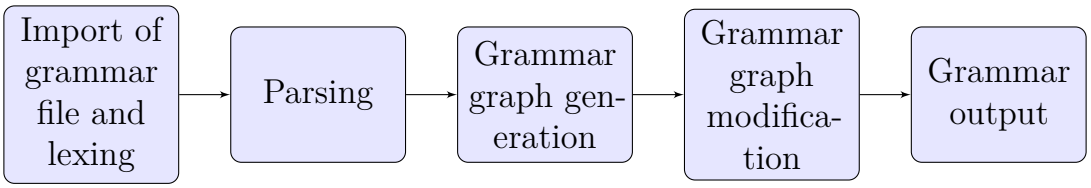


Figure 3.1: Procedure of extracting a sublanguage

### 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 language specification 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.

todo architecture diagram

## 3.3 Lexer

The lexer is responsible for extracting the tokens from the TPTP language grammar specification file. todo why lexer with ply, what does ply help PLY offers a lexer generator for python (see section 2.6). 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

todo check if bnf or ebnf

The grammar of the TPTP language is specified in a modified EBNF todo source. Therefore there are deviations from standard EBNF (see 2.2) that need to be analysed to specify elementary tokens. The standard EBNF only uses one production symbol (" ::= "). In the TPTP language additional production symbols have been added. The following table 3.1 contains the production symbols used in the TPTP language, that also have to be recognized by the lexer.

Table 3.1: TPTP language production symbols [**VS06**]

| **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 laguage grammar specification 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 language specification. 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.

todo regular expressions

todo explain token nt symbol + production rule bc. of parser

### 3.3.2 Regular Expressions

## 3.4 Parser

The goal that the parser component should fulfil is to take the tokens from the lexer as input and create a data structure that represents the structure of the TPTP language grammar specification sign. To create the parser the PLY parser generator is used.

-rules have to be defined

In the TPTP language specification 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 semantic rules square brackets are terminals.

### 3.4.1 Data Structure and Data Types

To build the representative data structure, data types that represent the data stored in the TPTP language grammar specification have to be defined. The following section describes the data structure and data types that are being 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. -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

### Expressions

An expression consists of the nonterminal symbol name which is produced, a production list and a position. The position denotes at which position in the TPTP language grammar specification the grammar expression was listed. This information is needed to maintain the original order of the expressions when printing the reduced grammar specification (todo reference).
For each production symbol (see table 3.1) there is a data type. This means that grammar, token, strict and macro expression data types are introduced.
Listing 3.1 contains an example of a line in a TPTP language grammar specification that is represented by the grammar expression data type.

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

Listing 3.1: Example of a grammar expression

The nonterminal symbol name which is produced is $<tff\_formula>$. The production list consists of two productions, as can be seen in the listing.

### Comment Block

A comment block is a list of consecutive comment lines.

### Production Element

A production element is. -terminal symbol -nonterminal symbol -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 language grammar specification 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. -show example

### Production Property

-none -repetition -optional

### 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 parantheses.

#### Grammar List

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

### 3.4.2 Production Rules

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

## 3.5 Graph Generation

The TPTP grammar, extracted from the TPTP grammar file, needs to be stored in a data structure that allows for modification and traversing. The data structure that is used is a graph representing nodes and weighted transitions between nodes. A node is ?? The node that 'controlls' the transition is the nonterminal symbol on the left side in the TPTP grammar. The receiving? end is on the right side of the production rule. In conclusion, if there is a transition between two nodes, the 'controlling' end can be represented by the 'receiving' node. In the example below the nonterminal symbol $< alpha\_numeric >$ can be reduced to the nonterminal symbol $< lower\_alpha >$ that again can be reduced to the terminal symbol $[a - z]$. Every node that is a nonterminal symbol has a least one controlling and one receiving transition. The nodes that are terminal symbol are 'leaves'. The graph represented in figure 3.2 has the same appearance as a tree data structure. However, a graph cannot be represented by a tree because circular dependencies exist and those cannot be represented by a tree data structure. An example is given in listing 3.2.

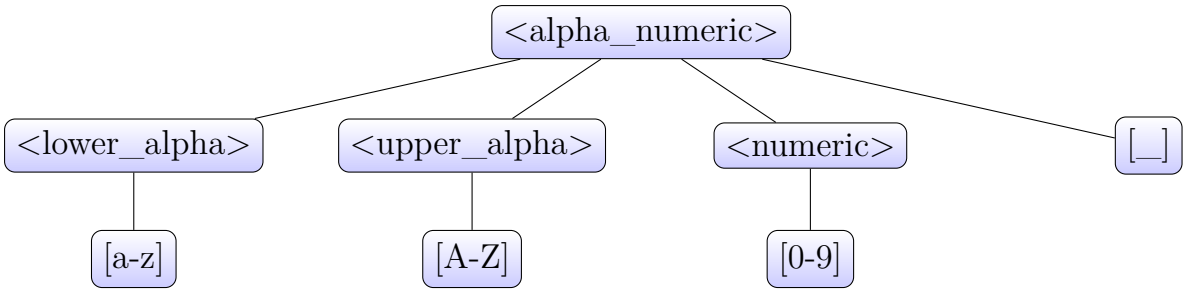


Figure 3.2: Example of parse tree

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

Listing 3.2: Example of a circular dependency

Another challenge is that the start symbol can have multiple rule types resulting in two start symbols. As it is not possible to have two start symbols, a new start symbol is generated that implies the two original start symbol. In the following example the original start should be $< formula\_role >$. Because $< formula\_role >$ has the rule type *Grammar* as well as the rule type *Strict*, a new start symbol $< start\_symbol >$ is generated that implies $< formula\_role >$.

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

Listing 3.3: Example of replacing the start symbol

### Generating the graph

The graph is generated recursive. Starting from the start symbol every nonterminal symbol that is on the right side of a production rule is a so called children of the nonterminal on the left side of the rule. These children again have children. This process repeats itself until all children are a terminal symbols.

## 3.6 Generation of the reduced Grammar

### 3.6.1 Selection of blocked Productions

### 3.6.2 Determination of the remaining reachable Productions

### 3.6.3 Determination of the remaining terminating Productions

## 3.7 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 3.4 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.

todo further describe control file, is it understood, which production is disabled? maybe graphical help??

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

Listing 3.4: 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 language grammar specification, the control file may have to be updated.

## 3.8 Maintainig Comments

In the TPTP language specification 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 3.5 features an example of a comment in the TPTP laguage specification.

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

Listing 3.5: Example of a comment in the TPTP language specification

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

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

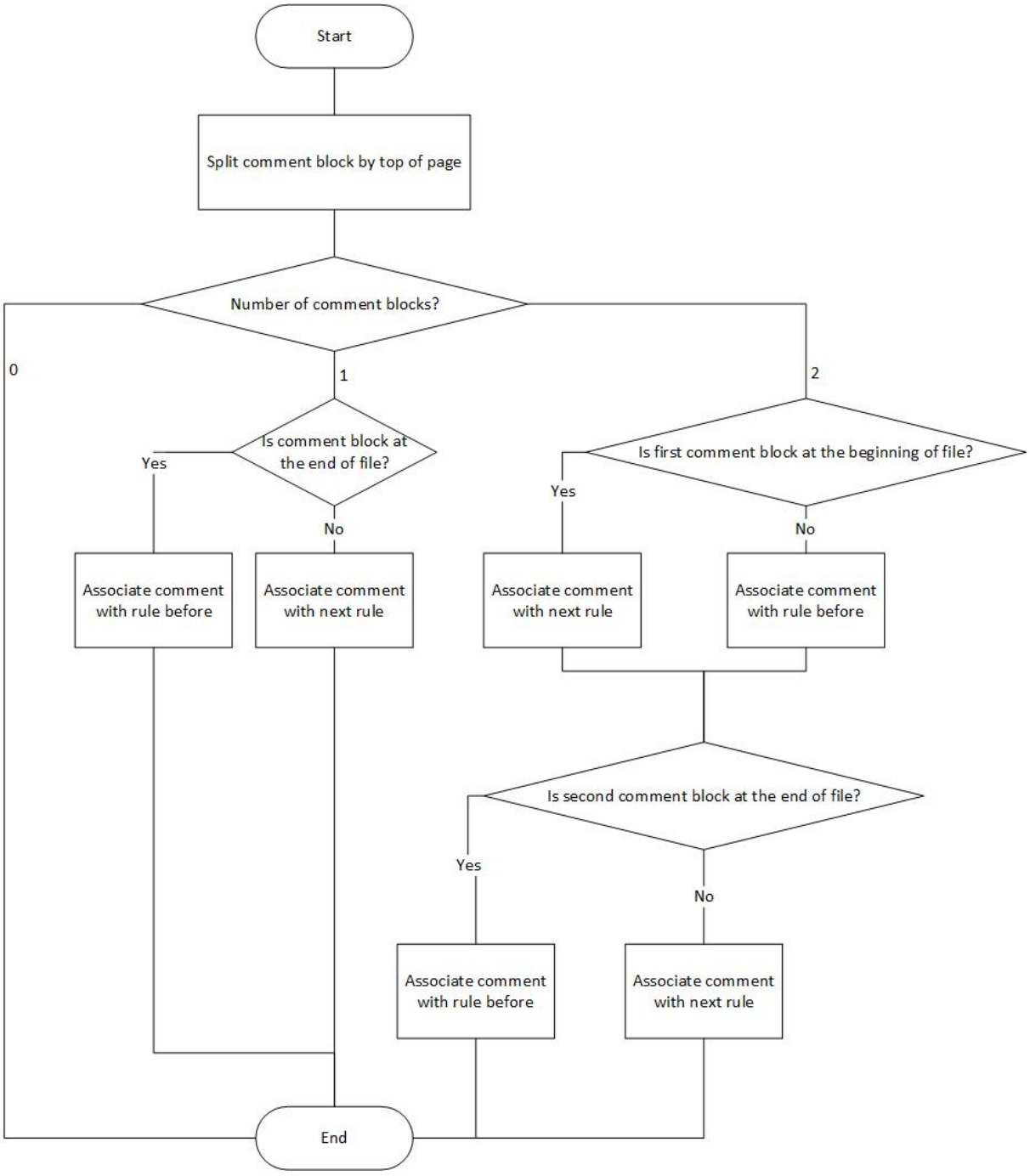


Figure 3.3: Maintaining Comments

## 3.9 Console Interface

-output reduced grammar with input grammar control file

## 3.10 GUI

-show rules similar to file -select start symbol + blocked productions -show extracted grammar -export exported grammar -include + toggle comments -import/export control file extra: -web import

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 certain symbol as symbol 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 don not have any special meaning other than providing clarity. Also, newlines are generally ignored because as can be seen below there are rules that cover multiple lines. If newlines wouldn't be ignored it would be difficult to ??

```
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 grammar, token, strict or macro expression. REFERENCE A non terminal symbol followed by the symbol itself (::=,:==,:::,...). The non terminal symbol and the symbol build a single token and are not identified as two tokens to avoid further ambiguity while parsing. Otherwise 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 the previous rule because as mentioned the 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.

### Non terminal symbol

A non terminal 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.

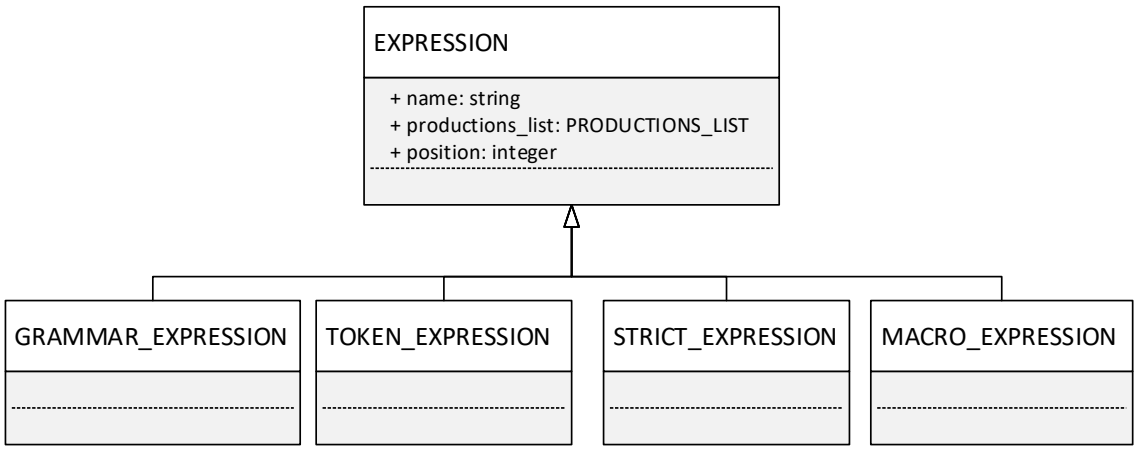


Figure 4.1: UML diagram for expressions

## 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
```

## 4.4 GUI

# 5 Validation

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

comment association

# 6 Conclusion

## 6.1 Future Work

comment association

# Bibliography