



# 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**

**Student ID; Course**

**Advisors**

nothing

8540946, 6430174; TINF17ITA

Prof. Dr. Stephan Schulz and Geoff Sutcliffe

# Contents

<b>Acronyms</b>	<b>I</b>
<b>List of Figures</b>	<b>II</b>
<b>List of Tables</b>	<b>III</b>
<b>Listings</b>	<b>IV</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement and Goals . . . . .	1
1.2 Structure of the Report . . . . .	2
<b>2 Background and Theory</b>	<b>3</b>
2.1 TPTP Language . . . . .	3
2.2 Backus-Naur form (BNF) . . . . .	3
2.3 Grammar . . . . .	4
2.4 Lexer . . . . .	4
2.5 Lexer . . . . .	5
2.6 Parser . . . . .	6
2.7 PLY . . . . .	7
2.8 Python? . . . . .	7
<b>3 Concept</b>	<b>8</b>
3.1 Requirements . . . . .	8
3.2 todo maybe rename Overview . . . . .	8
3.3 Lexer . . . . .	9
3.4 Parser . . . . .	10
3.5 todo Graph Generation . . . . .	11
3.6 Generation of the reduced Grammar . . . . .	11
3.7 Control File . . . . .	11
3.8 Maintainig Comments . . . . .	12
3.9 Output of reduced Grammar . . . . .	13
3.10 Console Interface . . . . .	13
3.11 GUI . . . . .	13
<b>4 Implementation</b>	<b>14</b>
4.1 Lexer . . . . .	14

4.2	Parser . . . . .	15
4.3	GUI . . . . .	15
<b>5</b>	<b>Validation</b>	<b>16</b>
<b>6</b>	<b>Conclusion</b>	<b>17</b>
	<b>Bibliography</b>	<b>i</b>

# Acronyms

<b>ATP</b>	automated theorem proving
<b>BNF</b>	Backus-Naur form
<b>CFG</b>	context-free grammar
<b>CNF</b>	first-order clause normal form
<b>EBNF</b>	extended Backus-Naur form
<b>FOF</b>	full first-order logic
<b>PLY</b>	Python Lex-Yacc
<b>TFF</b>	typed first-order logic
<b>THF</b>	typed higher-order logic
<b>TPTP</b>	Thousands of Problems for Theorem Provers

# List of Figures

3.1 Procedure of extracting a sublanguage . . . . .	8
---	---

# List of Tables

3.1	Thousands of Problems for Theorem Provers ( <a href="#">TPTP</a> ) language production symbols <a href="#">[7]</a> . . . . .	10
-----	--	----

# Listings

3.1	Example of a control file content . . . . .	12
3.2	Example of a comment in the <a href="#">TPTP</a> language specification . . . . .	13

# 1 Introduction

## 1.1 Problem Statement and Goals

Formal 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 dividing languages into smaller sub-languages that cover everything relevant to the specific use case. 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 divide the **TPTP** language in smaller sub-languages is to build a parser that parses the grammar of the **TPTP** language. The 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 corresponding 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

[1] Compiler: Translate (high-level) programming language into machine language

Different phases for writing a compiler, phases are processed sequentially

### 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](#). [2]

### 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.  $\langle TPTP\_File \rangle$  denotes the nonterminal symbol  $TPTP\_File$ . Productions are described using the "  $::=$  " symbol and alternatives are specified using the "|" symbol. [3] An example for a [BNF](#) production would be  $TPTP\_File ::= \langle TPTP\_Input \rangle \mid \langle comment \rangle$ . 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.

[4]

Different phases for writing a compiler, phases are processed sequentially

## 2.3 Grammar

## 2.4 Lexer

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 [5]. 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 belonging 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. [1]

A grammar is a list of rules that defines the relationships among tokens [5]. 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  $\Sigma$ , a nonterminal symbol  $\xi$  is called terminating if there are productions  $\xi \xrightarrow{*} z$  so that  $\xi$  can be derivated to  $z$  and  $z \in \Sigma^*$ . 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  $\Sigma$  and the start symbol  $S$ , a nonterminal symbol  $\xi$  is called reachable if there are production rules  $S \xrightarrow{*} u\xi v$  so that  $S$  can be derivated to  $u\xi v$  and  $u, v \in \Sigma^*$ . 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

## 2.5 Lexer

Lexing or a so called lexical analysis is the division of input into units so called tokens [5]. Tokens are for example variable names or keywords. The input is a string containing a sequence of characters, the output is a sequence of tokens. Afterwards, the output can be used for further processing e.g. 2.6.1. 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. [1]

A simple approach to build a lexer is to build an automata for each token definition and then test to which automata the input corresponds. 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 precedence of tokens can be declared.

Usually the token that is being defined the earliest has a higher precedence and thus will be chosen if multiple tokens fit the input. [1]

Another task of the lexer is separating the input in order to divide it into tokens. Per convention the longest input that matches any token is chosen. [1]

## 2.6 Parser

### 2.6.1 Yacc

Building a syntax tree out of the generated tokens [1]

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. [5]

#### Syntax analysis

Parsing: establish relationship among tokens [5] Grammar: list of rules that defines the relationships [5]

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 example random guessing (predictive parsing) or LR parsing. Input: description of grammar [5] Output: parser [5]

-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

## 2.6.2 PLY

## 2.7 PLY

Python Lex-Yacc ([PLY](#)) [6] is an implementation of lex and yacc in python. [LALR-parsing] consists of lex.py and yacc.py

lex.py tokenizes an input string

### 2.7.1 Nondeterministic Finite Automata

## 2.8 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

## 3.2 todo maybe rename 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 be printed, based on the new graph, in the same format as the original language specification.

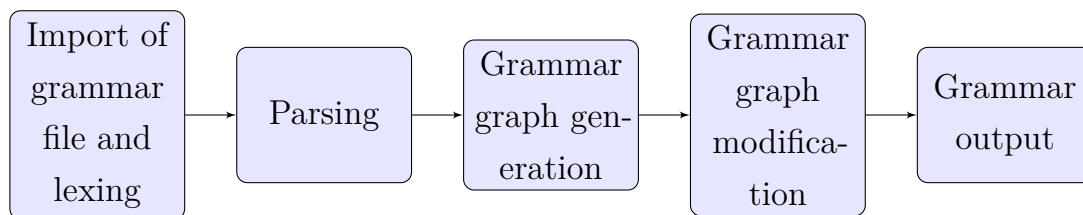


Figure 3.1: Procedure of extracting a sublanguage

### 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 2.7). 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 [7]

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** language 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

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

The [TPTP](#) grammar, extracted from the [TPTP](#) grammar file, needs to be stored in a data structure that allows for modification. A graph representing possible transitions within the

### 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 todo Graph Generation

-number rules -challenge that one start symbol can have multiple rule types -uml  
NTNode

## 3.6 Generation of the reduced Grammar

## 3.7 Control File

A format for specifying the desired start symbol and blocked productions has to be developed. Using a file-based format 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 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.

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

Listing 3.1 contains a sample control file. In this file `<TPTP_File>` is specified as start symbol.

todo further describe control file

```

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

```

Listing 3.1: Example of a control file

### 3.7.1 Selection of blocked Productions

### 3.7.2 Determination of the remaining reachable Productions

### 3.7.3 Determination of the remaining terminating Productions

## 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 3.2 features an example of a comment in the [TPTP](#) language specification.

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

Listing 3.2: 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)

## 3.9 Output of reduced Grammar

## 3.10 Console Interface

## 3.11 GUI

cd /

# 4 Implementation

## 4.1 Lexer

-Definition of tokens

-Tabs and newlines ignored

-Newline would be helpful to identify comments because a comment is a newline followed by the percentage sign, as well as new rules if each rule would be represented in one line. However, there are rules that cover multiple lines. That is the main reason newlines are ignored.

-A comment is identified by the lexer as a percentage sign followed by an arbitrary character excluding "]". This is followed by any arbitrary character. A comment can not only be identified by a percentage sign as the percentage sign is also part of the terminal symbols. However, the percentage symbol when used as terminal symbol is embedded in square brackets.

Tokens: LGRAMMAR/TOKEN/STRICT/MACRO EXPRESSION:

Any arbitrary symbol that is the name of the rule followed by the symbol itself  
(:==,:::,...)

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.

T SYMBOL:

COMMENT:

OPEN SQUARE BRACKET/CLOSE SQUARE BRACKET, OPEN/CLOSE PARENTHESIS, ALTERNATIVE SYMBOL, REPETITION SYMBOL:

-is recognized and represented by the symbol itself

test

## 4.2 Parser

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

## 4.3 GUI

# 5 Validation

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

comment association

# 6 Conclusion

## Outlook

comment association



# Bibliography

## Publikationen

- [1] Torben Aegidius Mogensen. *Introduction to Compiler Design*. Springer, 2017. ISBN: 9783319669656.
- [2] 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.
- [3] 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](https://doi.org/10.1145/355588.365140). URL: <https://doi.org/10.1145/355588.365140>.
- [4] 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](https://doi.org/10.1145/359863.359883). URL: <https://doi.org/10.1145/359863.359883>.
- [5] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O’Reilly Media Inc., 1992. ISBN: 9781565920002.
- [7] A. Van Gelder and G. Sutcliffe. “Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation”. In: *Proceedings of the 3rd International Joint Conference on Automated Reasoning*. Ed. by U. Furbach and N. Shankar. Lecture Notes in Artificial Intelligence 4130. Springer-Verlag, 2006, pp. 156–161.

## Online Quellen

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