



# 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 Compiler . . . . .	3
2.3 Grammar . . . . .	3
2.4 Lexer . . . . .	4
2.5 Parser . . . . .	6
2.6 PLY . . . . .	6
2.7 Python? . . . . .	7
<b>3 Concept</b>	<b>8</b>
3.1 Overview . . . . .	8
3.2 Lexer . . . . .	8
3.3 Parser . . . . .	9
3.4 Generation of the Reduced Grammar . . . . .	9
3.5 Selection of blocked Productions . . . . .	9
3.6 Determination of the remaining reachable Productions . . . . .	9
3.7 Determination of the remaining terminating Productions . . . . .	9
3.8 GUI . . . . .	9
<b>4 Implementation</b>	<b>10</b>
4.1 Lexer . . . . .	10
4.2 Parser . . . . .	11
4.3 GUI . . . . .	11
<b>5 Validation</b>	<b>12</b>

<b>6 Conclusion</b>	<b>13</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

# List of Tables

3.1	Total Ingestion Duration for an <b>SSN!</b> (SSN!) file . . . . .	8
-----	---	---

# Listings

# 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 TODO APLICATION is developed. Based on this, the implementation of the application is featured in chapter 4. 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]

The [EBNF](#) [3] is a `todo` and is used to describe context-free grammars (CFGs).

### 2.2 Compiler

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

Different phases for writing a compiler, phases are processed sequentially

### 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).

A grammar is a list of rules that defines the relationships among tokens [4]. 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 [5].

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. [5]

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  $\xi$ . [5]

### Context-free grammar

## 2.4 Lexer

Lexing or a so called lexical analysis is the division of input into units so called tokens [4]. 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. ??.

A lexer represents an automaton recognizing the specified language TODO 2.4.

A lexer separates the given input in order to divide it into tokens.

Therefore, the 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 automaton 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 automaton 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]

The input can also be separated using the longest input that matches any token is chosen. [1] This convention has higher precedence than any previously defined precedences.

### **Lexer Generator**

Due to the given complexity, a lexer is often generated by a lexer generator and not written manually.

A lexer generator takes a specification of tokens as input and generates the lexer automatically.

### **Regular expression and formal language**

The specification of tokens is usually written using regular expressions. A regular expression represents a set of strings and describes a formal language. A formal language defines a set of words belonging to the language. These words consist of a finite sequence of characters that on the other hand describe the alphabet of the language.

## 2.5 Parser

Parsing describes the process of establishing a relationship among previously generated tokens [4]. The established relationship is described in a syntax tree [1]. The leaves of the syntax tree are the 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. [4]

### Syntax analysis

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.

- 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) [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 Python?

### 2.7.1 Packages

PyQT

# 3 Concept

This chapter outlines the concept and the architecture of the software tool. First, the proposed software architecture is described.

## 3.1 Overview

### 3.1.1 Proposed Architecture

## 3.2 Lexer

-deviations from plain ebnf The [TPTP](#) language is specified in an [EBNF](#), but is modified for its specific purpose???. Therefore there are deviations from standard [EBNF](#) that need to be analysed to specify todo elementary tokens and rules for matching tokens. The standard [EBNF](#) uses only 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.

Table 3.1: Total Ingestion Duration for an **SSN!** file

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

[7]

## 3.3 Parser

### 3.3.1 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 Generation of the Reduced Grammar

## 3.5 Selection of blocked Productions

## 3.6 Determination of the remaining reachable Productions

## 3.7 Determination of the remaining terminating Productions

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

## 3.8 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...

# 6 Conclusion

Outlook

# 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] 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>.
- [4] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O’Reilly Media Inc., 1992. ISBN: 9781565920002.
- [5] Armin Cremers and Seymour Ginsburg. “Context-free grammar forms”. In: *Journal of Computer and System Sciences* 11 (1975), pp. 86–117.
- [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).