# 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** | nothing |
| **Student ID; Course** | 8540946, 6430174; TINF17ITA |
| **Advisors** | Prof. Dr. Stephan Schulz and Geoff Sutcliffe |

# Contents

# Acronyms

| | |
|---|---|
| **BNF** | Backus-Naur Form |
| **CNF** | First-Order Clause Normal Dorm |
| **FOF** | Full First-Order Logic |
| **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

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 Dorm (CNF) or Full First-Order Logic (FOF) The grammar of the language is provided in an extended Backus-Naur Form (BNF).
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 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 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

## 2.1 TPTP Language

[2]

## 2.2 Backus-Naur Form (BNF)

## 2.3 Compiler

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

Different phases for writing a compiler, phases are processed sequently

## 2.4 Lexer

Lexing or a so called lexical analysis is the division of input into units so called tokens [3]. 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. [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 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. [1]

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. [1]

## 2.5 Parser

### 2.5.1 Yacc

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

Parsing: establish relationship among tokens [3] Grammar: list of rules that defines the relationships [3]

Input: description of grammar [3] Output: parser [3]

### 2.5.2 PLY

Python implementation of lex and yacc [LALR-parsing] consists of lex.py and yacc.py

lex.py tokenizes an input string

http://www.dabeaz.com/ply/ply.html

### 2.5.3 Nondeterministic Finite Automata

# 3 Concept

## 3.1 Overview

## 3.2 Tokenizer

## 3.3 Parser

### 3.3.1 Data Structure

to store the TPTP Grammar

## 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

# 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 captial 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

## 4.2 Parser

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

## 4.3 GUI

# 5 Validation

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