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
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 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

[**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 $TPTP\_File ::= <TPTP\_Input> \mid <comment>$. 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 ξ is called terminating if there are productions $\xi \underset{\rightarrow}{*} z$ 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. [**Cremers75**]

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

3.1 Overview

The software tool consists of five main components: the lexer, which extracts tokens from a TPTP grammar file. The parser, which analyses the tokens and constructs a data structure representing the TPTP grammar. todo

Figure 3.1 outlines the main activities of the software tool in a

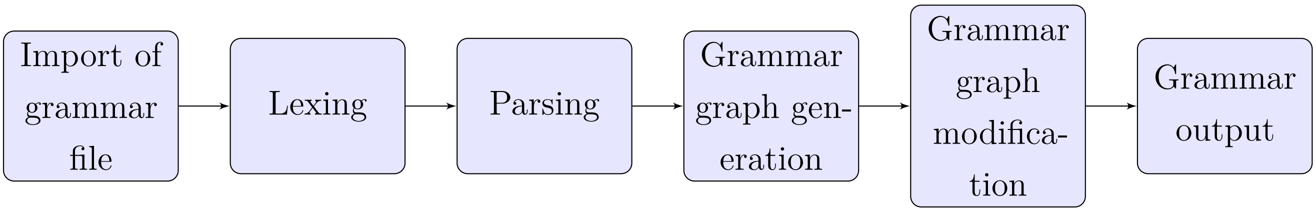

Figure 3.1: General tool workflow

3.2 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.

3.2.1 Proposed Architecture

3.3 Lexer

The TPTP language is specified in a modified EBNF. Therefore there are deviations from standard EBNF (2.2) that need to be analysed to specify elementary tokens in the lexer. The standard EBNF uses only 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: TPTP language production symbols [**VS06**]

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

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. 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 this line.

3.4 Parser

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

3.5 Generation of the Reduced Grammar

3.5.1 Selection of blocked Productions

3.5.2 Determination of the remaining reachable Productions

3.5.3 Determination of the remaining terminating Productions

3.6 Maintainig Comments

In the TPTP language specification there are comments providing supplemental information about the language and its symbols and rules. When genering 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.

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

Listing 3.1: 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)

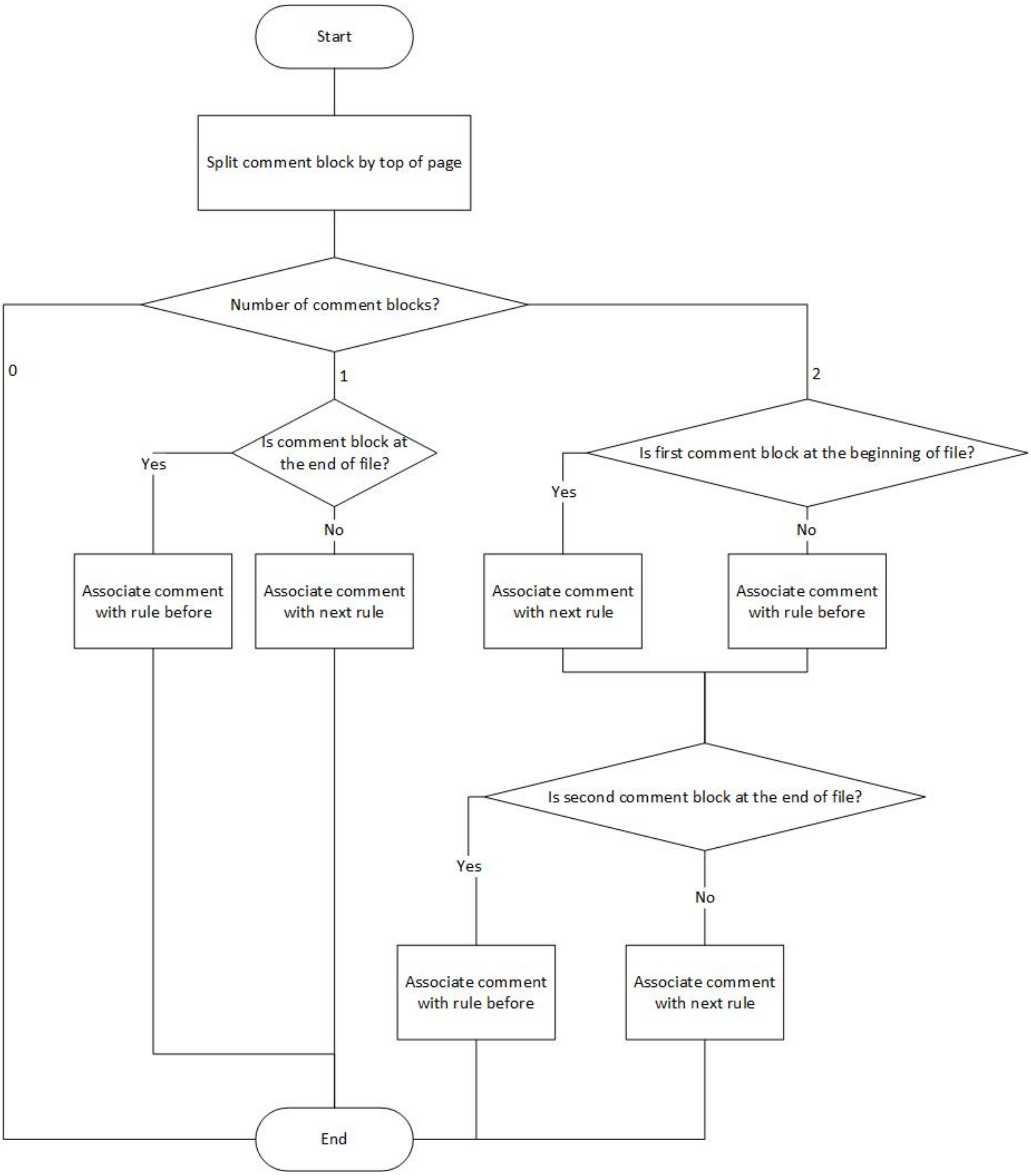
Start
Split comment block by top of page
Number of comment blocks?
0
1
2
Is comment block at the end of file?
Yes
No
Associate comment with rule before
Associate comment with next rule
Is first comment block at the beginning of file?
Yes
No
Associate comment with next rule
Associate comment with rule before
Is second comment block at the end of file?
Yes
No
Associate comment with rule before
Associate comment with next rule
End

Figure 3.2: Maintaining Comments

3.7 Console Interface

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 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 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

6.1 Future Work

Bibliography