Cutting down the TPTP language (and others)

Nahku Saidy[1], Hanna Siegfried[1], Stephan Schulz[1], and Geoff Sutcliffe[2]

[1] DHBW Stuttgart, Germany
nahku.saidy@daimler.com
hanna.siegfried@daimler.com
schulz@eprover.org
[2] University of Miami, FL, USA
geoff@cs.miami.edu

Abstract

Computer languages are likely to grow over time as they get more complex when their functionality is extended. An example of that is the TPTP language for automated theorem proving. Over time various forms of classical logics ranging from Clause Normal Form (CNF) to Typed Higher-order Form (THF) have been added, and that extended the TPTP language. This paper describes *Synplifier*, a tool that automatically extracts sub-languages from the TPTP language. Automatic extraction instead of manually maintaining sub-languages has the advantage of avoiding maintenance overhead as well as unnoticed divergences from the full language. Sub-languages of interest are for example CNF and First-order Form (FOF), and are extracted based on the user's selection. Synplifier has been successfully tested by extracting CNF from the TPTP language.

1 Introduction

Computer languages are designed under competing constraints. On the one hand, languages should be as expressive as possible, allowing users to specify their intent in a simple, compact, and natural way. This calls for a syntax with a large number of specialised features, geared to many possible use cases. On the other hand, increased complexity makes a language more difficult to learn and use, and, in particular, makes it more difficult to implement tools that use the language. One example of this is the TPTP [8] family of languages, which is used nearly universally by Automated Theorem Proving (ATP) systems for classical logics. Originally the TPTP used only Clause Normal Form (CNF) [5], but over the years expanded to include First-Order Form (FOF) [5], Typed First-order Form (TFF, consisting of the monomorphic TF0 [7] and the polymorphic TF1 [2]), through to typed higher-order form (THF, consisting of the monomorphic TH0 [6] and the polymorphic TH1 [4]). There are proposals to further extend the TPTP syntax, to an extended typed first-order form (TFX) [9], and to modal logics [3].

The TPTP syntax uses an extended Backus-Naur Form (BNF), with distinct rules for syntax, semantic constraints, tokens, and character classes. A small tool chain based on Lex/Yacc for building a basic parser from the syntax is available [10]. While the TPTP is a major success story, the resulting syntax has grown large and complex.[1] Understanding the syntax is non-trivial. For new developers who want to implement "only" a first theorem prover for CNF, identifying the relevant parts of the syntax and implementing a parser is a significant barrier to entry.

In this paper, we describe a tool called *Synplifier* ("Syntax Simplifier") that extracts a coherent sub-syntax from a larger syntax (in Backus-Naur form). Synplifier parses the syntax and presents it in a GUI. The user specifies the start symbol(s) of the desired sub-syntax,

[1] http://www.tptp.org/TPTP/SyntaxBNF.html

and Synplifier extracts the parts of the syntax that are necessary for the sub-syntax. The user can disable arbitrary (alternatives of) syntax rules. Synplifier uses heuristics to detect potentially relevant comments in the input syntax file and maintains them. Synplifier has been used successfully on the TPTP 7.3.0 syntax, to extract a sub-syntax for pure CNF. The source code of Synplifier is available on GitHub[2].

2 Architecture and Implementation

In order to decompose the full TPTP syntax into smaller sub-syntaxes we propose to extract desired sub-syntaxes by selecting a start symbol and disabling undesired productions. The TPTP grammar is processed in multiple stages. These stages include (1) parsing the grammar into a structured internal representation and (2) presenting it to the user in order to define undesired productions and the start symbol. To (3) extract the sub-syntax, we use a recursive algorithm that checks which symbols and productions are still reachable and extracts all reachable productions.

Our assumed use cases are all hand-created grammars of relatively small size (dozens to thousands of rules). Also, the algorithms employed do not have high complexity. We have therefore settled on Python 3 as the implementation language. Python is a modern multi-paradigm language with good support and excellent libraries. In particular, there is an existing tool chain for building parsers in Python.

2.1 Overview

Figure 1 outlines the procedure of extracting a sub-syntax of the TPTP language. First, the TPTP syntax file is loaded and broken into to tokens using the lexer. The next phase converts the token sequence into an internal data structure using a parser. This step also also checks the syntax of the input file. From the abstract syntax, we then build a graph representing the imported TPTP syntax.

This graph can be manipulated by disabling certain transitions or selecting a new start symbol in the following phase. This phase also includes computation of the remaining reachable and terminating productions. The resulting reduced graph represents the syntax of the extracted sub-language. To output the reduced grammar in the same format as the TPTP original syntax, the new graph is serialized into a string that can be written to an output file.

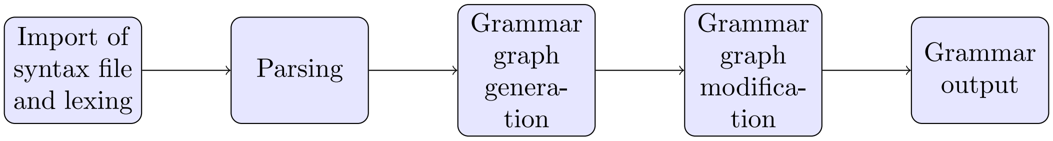


Figure 1: Procedure of extracting a sublanguage

2.2 Lexing

The lexer is responsible for extracting tokens from the TPTP syntax file. We decided to use PLY (Python Lex-Yacc)[3] which is an implementation of lex and yacc in Python. Using PLY a

[2] https://github.com/nahku/Synplifier
[3] https://www.dabeaz.com/ply/

lexer can be generated by specifying tokens using regular expressions.

In order to find elementary tokens the TPTP syntax has been analysed and regular expressions that precisely describe these tokens have been defined. Standard extended BNF uses only one production symbol (`::=`). In the TPTP syntax, the standard production symbol is used for syntactic rules. Additional symbols for semantic (`:==`), lexical (`::-`), and character-macro (`:::`) rules have been added.

The following tokens are recognized by the lexer:

- Following standard BNF, **non-terminal** symbols are enclosed by the < and > symbols. In between there can be any alphanumeric characters and underscores.
- A **terminal symbol** does not have any special notation, and can be any sequence of characters excluding whitespace characters.
- There are four **expression** token types (one for each rule type). Expressions are defined as a non-terminal symbol followed by a production symbol. The non-terminal symbol and the following production symbol are considered to be a single token, and are not identified as two separate tokens. This clearly identifies the start of a new rule and therefore avoids ambiguity while parsing. The example below features two tokens, a syntax expression and a non-terminal symbol.

```
<formula_role>          ::= <lower_word>
```

- A **comment** is defined as the start of a new line, a percentage sign, arbitrary characters and ends with a newline character. The percentage sign when used as terminal symbol is embedded in square brackets and therefore cannot be the first character of a new line.
- Additional tokens are the meta-symbols including **open** ”(” and **close parentheses** ”)”, **open** ”[” and **close square brackets** ”]”, **repetition symbols** ” ∗ ”, and **alternative symbols** ” | ”.

2.3 Parsing

The parser takes the tokens from the lexer as input and creates a data structure that represents the structure of the TPTP syntax.

Figure 2 outlines the responsibilities of the parser component and the sequence of its subfunctions. First, the tokens generated by the lexer need to be parsed and based on that the data structure representing the TPTP syntax is to be created. Secondly, the rules in the data structure have to be numbered to maintain the correct order for output. The third step is the so-called disambiguation of square brackets. In the TPTP syntax, square brackets not necessarily denote that an expression is optional, which is the case in traditional EBNF. In lexical and character-macro rules they denote that an expression is optional and in syntactic and semantic rules square brackets are terminals. Therefore, disambiguation of square brackets is necessary. As a last step, the output of the parser is returned.

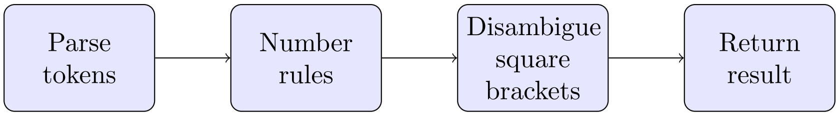


Figure 2: Parsing procedure

2.4 Graph generation

After parsing the TPTP grammar is stored in a grammar list. The grammar list data structure does not allow traversing which is why a new data structure is introduced that allows for modification and traversing. The data structure that is used is a graph representing the grammar.

The non-terminal symbols in the TPTP syntax are represented by a node class that has the following attributes:

- value: name of non-terminal symbol
- productions list: list of non-terminal symbols that have been created by the parser
- rule type: rule type of the non-terminal symbol
- comment block: list of comments belonging to the non-terminal symbol
- position: position of the production in the input file
- children: nested list that contains the node of every non-terminal symbol that is part of the productions list

Before generating the graph, a dictionary is created. The dictionary provides an efficient way accessing the nodes in order to build the grammar graph and also during the next steps of the sub-syntax generation. This dictionary contains nodes constructed from the grammar list output by the parser that contains rules and comments. While constructing the dictionary, comments in the grammar list are associated with nodes using a heuristic that is described in section 2.5. The combination of the nodes value and rule type form the unique key for each node. The dictionary provides an efficient way accessing them in order to build the grammar graph and also during the next step of sub-syntax generation.

As one non-terminal symbol in the TPTP syntax can be mapped to multiple nodes, a new temporary start symbol is introduced. The example below shows the productions for the non-terminal symbol $<formula\_role>$. Since this non-terminal symbol has multiple types of rules one node will be created for each rule type.

```
<formula_role>         ::= <lower_word>
<formula_role>         :== axiom | hypothesis | definition | assumption |
                           lemma | theorem | corollary | conjecture |
                           negated_conjecture | plain | type |
                           fi_domain | fi_functors | fi_predicates | unknown
```

If a non-terminal symbol that has multiple rule types is selected as the desired start symbol multiple nodes would represent that start symbol and therefore it would not be possible to select one node as the starting point of the graph generation. To solve this problem, a temporary start symbol is introduced before generating the graph. The temporary start symbol produces the start symbol that the user specified and is used as a starting point for the graph generation. If the non-terminal symbol $<formula\_role>$ that was mentioned before would be selected as start symbol by the user a temporary start symbol representing the rule

```
<start_symbol>         ::= <formula_role>
```

would be introduced. This ensures that only one node is representing the start symbol, that is used for graph generation.

Starting with the temporary start symbol, the graph is generated recursively. Iterating over each non-terminal symbol in the productions list of the start symbol, the corresponding nodes are identified. These nodes are then appended to the list of children of the start symbol. The identified children may again have children. This process is repeated until a node has no children because there are only terminal symbols in the productions list of a non-terminal symbol.

Since it is possible for a non-terminal symbol to be on the right side as well as on the left side of the same production rule, a node can also be its own child. To avoid revisiting the same node infinitely, it is checked whether a node already has children so that it will not be visited again. This also improves the performance of Synplifier as a non-terminal symbol that has already been visited will not be visited again independent of circular dependencies.

The following example shows a production rule and the resulting list of children belonging to the node. Each production alternative has its own list of children.

```
Production rule:
<disjunction>  ::=  <literal> | <disjunction><vline><literal>
Output:
node.value: <disjunction>
node.ruleType: grammar
node.children: [[<literal>],[<disjunction>,<vline>,<literal>]]
```

2.5 Maintaining comments

In the TPTP syntax there are comments providing supplemental information about the language and its symbols and rules. When generating a reduced grammar maintaining comments is desired. This means that comments from the original language specification should be associated with the rule they belong to semantically and if the rule is still present in the reduced grammar, the comment should be as well.

The following example features a comment in the TPTP syntax. This comment begins with a *Top of Page* line that, in the HTML version of the TPTP syntax, contains a hyperlink which leads to the beginning of the syntax file. The next line contains a relevant comment.

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

The heuristic matching comments to rules takes these *Top of Page* lines into account. When there is a *Top of Page* line in between comment lines it generally also splits comments semantically. Therefore it would be correct to associate the comment line(s) after the *Top of Page* line to the rule after. Also, if there is one *Top of Page* line in between multiple comment lines it is highly probable that the first part of the comment lines before the *Top of Page* line refer to the rule before the comment lines and that the comment lines after the *Top of Page* line refer to the rule after the comment lines. In the following the process of splitting and maintaining comments is summarized. First, the comment block is split into multiple separate comment blocks by using *Top of Page* lines as separators.

- If this results in no comment blocks, the comment block consisted only of one line which is the *Top of Page* line. Then no comment block has to be associated to a rule because *Top of Page* lines are not relevant.
- If this results in one comment block, that means that no *Top of Page* line was present in the comment block and the comment block is associated with the rule after, if the comment

block is not at the end of the file. If it is at the end of the file it is associated with the rule before.

- If this results in two comment blocks, one *Top of Page* line was present. Then the comment block before the *Top of Page* line is associated with the rule before when possible. If this comment block is at the beginning of the file it is associated with the rule after. The comment block after the *Top of Page* line is associated with the rule after. If it is at the end of the file it is associated with the rule before.

The case of three or more comment blocks after splitting the original comment block is not featured above because it does not occur in the TPTP syntax version 7.3.0. Since it might occur in a future version of the TPTP syntax it is handled by merging all comment blocks starting from the second comment block and then following the procedure of two comment blocks as explained above.

2.6 Sub-syntax extraction

Information about which part of the syntax to extract are described in a control file. To extract a sub-syntax from the original grammar graph, four steps must be performed:

1. The control file has to be parsed, in order to get the information on the desired start symbol and which productions to block.
2. The blocked productions specified in the control file must be disabled and therefore the corresponding transitions are removed from the grammar graph.
3. The remaining part of the grammar that is terminating must be computed.
4. From the part of the grammar that is terminating, non-reachable symbols are removed.

2.6.1 Control file format

A format for specifying the desired start symbol and blocked productions is required. Using a file-based configuration enables the user to store desired configurations so that a manual selection in the GUI is not necessary. It is also essential for 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 the specification of all necessary information. This includes the desired start symbol and all production rules that should be blocked. The way this information is provided is as follows:

- Define the desired start symbol in the first line.
- Define blocked productions grouped by the non-terminal symbol and rule type symbol, separating each group by a new line. First defining the non-terminal symbol, then the production symbol and after that the indexes of the alternatives that should be blocked (indexing starts at zero).

Identifying the production symbol is necessary because there are non-terminal symbols that have productions with more than one production symbol. The example below contains a sample control file. In this file, <*TPTP_file*> is specified as start symbol. The second line indicates that the second grammar production alternative of the non-terminal symbol <*TPTP_input*> should be disabled. Furthermore, the first, second, third and fifth grammar production alternative of the non-terminal symbol <*annotated_formula*> are disabled in the third line.

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

This format is easy to parse and also enables users to specify their desired start symbols and blocked productions without having to use the GUI. The control file can also be generated from selections made by the user in the GUI. It can be exported as a textfile. When generating a sub-syntax from the user GUI selection a control file representing that selection is created internally as input for the next processing steps.

2.6.2 Removing blocked productions

In the control file, for each non-terminal symbol, whose productions should be modified, its name, rule type and the indices of the productions that should be blocked are listed. From all nodes that are addressed (by non-terminal symbol name and rule type), the indexed productions are removed. This includes deleting the corresponding element from the productions list and from the children list.

2.6.3 Determination of the remaining terminating symbols

After the desired productions have been deleted from the grammar graph, the next step is to remove the non-terminating symbols from the grammar graph. First it must be determined which symbols are terminating and which are non-terminating. Terminating nodes are found by iterating the nodes in the *children_list* of a node. If the *children_list* is empty, the production consists only of terminal symbols. If that is not the case, the node is terminating if all nodes in the *children_list* represent a non-terminal symbol that is terminating. If a non-terminal symbol has multiple rule types it is represented by multiple nodes. Also, if a non-terminal symbol with multiple rule types is in featured in a production, all nodes representing that non-terminal symbol are in the children list corresponding to this production. However, only one of the nodes needs to be terminating for the non-terminal symbol to be terminating.

After the terminating symbols have been determined, productions that contain a non-terminating symbol are removed from the nodes of the grammar graph. Because some non-terminal symbols might have productions that contain non-terminating symbols, but also other productions that only contain terminating symbols, only the productions containing non-terminating symbols are removed.

2.6.4 Determination of the remaining reachable symbols

After removing the productions that are non-terminating it has to be determined which part of the grammar still remains reachable. This is done by generating a new grammar graph starting from the desired start symbol. When generating the new grammar graph, all parts of the grammar that are reachable from the start symbol are added to the new grammar graph. All nodes that are not part of the new grammar graph are not reachable and can therefore be removed.

2.7 Output generation

Within Synplifier a syntax is represented by a grammar graph. To export sub-syntaxes that have been extracted from a grammar graph they have to be converted to the original form of a TPTP syntax file. There are three steps necessary, displayed in Figure 3, in order to convert a syntax represented by a grammar graph to the original TPTP syntax file form.

The first step is to traverse the grammar graph by iterating all nodes and getting a string representation of the nodes. For creating the string representation, the Python string class is used.[4] The string class creates a string version of the object passed to the constructor. For doing so it calls the *__str__()* of the passed object, if the object provides that method.

In order to maintain the same order that rules have in the TPTP syntax, the position of the nodes is saved alongside the node string. After generating all the strings, in the second step all the node strings are ordered based on the position of the node.

In the third step the ordered list of nodes strings is written to a file and exported. The node that represents the temporary start symbol is not printed as it does not belong to the TPTP syntax.

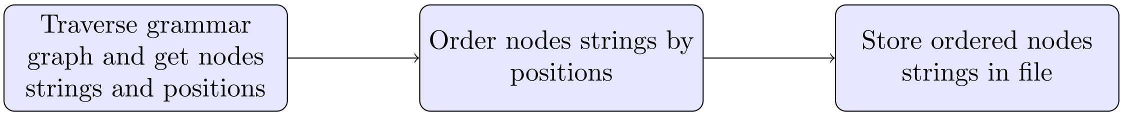

Traverse grammar graph and get nodes strings and positions
Order nodes strings by positions
Store ordered nodes strings in file

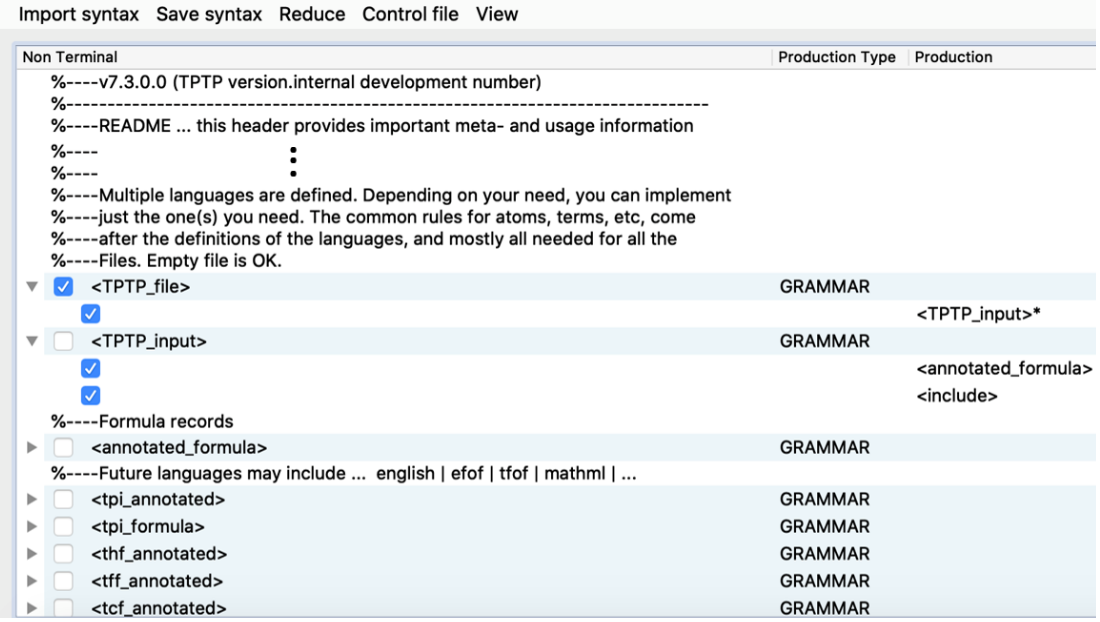
Figure 3: Procedure of generating a string representing the grammar graph

2.8 GUI

The graphical user interface is built using PyQt[5]. An advantage of using PyQT is that it offers a treeview that is used for displaying the grammar. In comparison to the standard Python library Tkinter[6], PyQt offers checkboxes in treeviews that are used for selecting blocked productions. The menu of the GUI consists of five submenus.

The *Import* menu provides the option to import a TPTP syntax file in text file format from local storage, or to import the latest TPTP syntax version from the TPTP website. The TPTP syntax on the TPTP project website is stored in a HTML format. This file is downloaded and converted to plain text using the Beautiful Soup Python library[7]. After selecting a file for import a start symbol needs to be selected. Starting with the start symbol a grammar graph is generated and the corresponding text displayed. Each rule in the TPTP grammar is a top element of the treeview and can be expanded to show the production alternatives. Right of the non-terminals name the rule type is displayed. Comments that have been assigned to a rule are also displayed. An example of the imported TPTP grammar is shown in Figure 4.

[4] https://docs.python.org/3/library/stdtypes.html#str
[5] https://www.riverbankcomputing.com/static/Docs/PyQt5/introduction.html
[6] https://docs.python.org/3/library/tkinter.html
[7] https://pypi.org/project/beautifulsoup4/



Non Terminal	Production Type	Production
%----v7.3.0.0 (TPTP version.internal development number)		
%--		
%----README ... this header provides important meta- and usage information		
%---- ⋮		
%----		
%----Multiple languages are defined. Depending on your need, you can implement		
%----just the one(s) you need. The common rules for atoms, terms, etc, come		
%----after the definitions of the languages, and mostly all needed for all the		
%----Files. Empty file is OK.		
▼ ☑ <TPTP_file>	GRAMMAR	
☑		<TPTP_input>*
▼ □ <TPTP_input>	GRAMMAR	
☑		<annotated_formula>
☑		<include>
%----Formula records		
▶ □ <annotated_formula>	GRAMMAR	
%----Future languages may include ... english | efof | tfof | mathml | ...		
▶ □ <tpi_annotated>	GRAMMAR	
▶ □ <tpi_formula>	GRAMMAR	
▶ □ <thf_annotated>	GRAMMAR	
▶ □ <tff_annotated>	GRAMMAR	
▶ □ <tcf_annotated>	GRAMMAR	

Figure 4: GUI

The *View* menu provides the possibility to toggle the display of comments to improve readability. In order to extract a sub grammar, the user has to choose a new start symbol by checking one of the check boxes left of the non-terminal symbols name. By default, the start symbol that has been selected after importing the grammar is selected. For selecting blocked productions, the user has the choice of on the one hand expanding the rules and unchecking the checkboxes belonging to the productions the user wishes to block. On the other hand, the user can import a control file. If the user imports a control file, the checkboxes are set accordingly.

In the *Reduce* menu the user can reduce the grammar according to his selections. The grammar is reduced and the reduced grammar is displayed afterwards. The rules maintain the same order as in the original TPTP grammar.

In the *Save syntax* menu the user can reduce the grammar based on the GUI selection or an imported control file. The difference between the reduce option and the save menu is that the save menu saves the generated reduced grammar to local storage and does not display it. If the user wants the <*comments*> production that is necessary for using the automated parser generator (see Section 3.1, that production is included in the reduced grammar.

The *Control file* menu gives the user the option to import a control file instead of blocking productions manually. If the user imports a control file the checkboxes are set accordingly. The user also has the option of generating a control file based on his GUI selection. This is particularly helpful if the user uses the same blocked productions multiple times. This way he can use the import control file option later instead of selecting the blocked productions in the GUI tool.

2.9 Command line interface

In addition to the GUI, Synplifier offers a command-line interface to provide the means for convenient automation of sub-syntax extraction. It takes a TPTP syntax file and a control file as input and outputs the resulting sub-syntax. It is implemented using the Python module argparse[8], which provides a framework for creating command-line argument parsers. The table below provides an overview of the command-line arguments. The syntax file location and the control file location have to be specified. Specifying an output path and filename is optional, by default the output filename will be *output.txt*. The $-ex$ flag enables additional output of the comment syntax. The help description can be displayed by using the *-h* option.

Name	*Short form*	*Default*	*Description*
--grammar	-g	None	TPTP syntax file path and filename
--control	-c	None	Control file path and filename
--output	-o	output.txt	Output file path and filename (optional)
--external_comment	-ex	False	Flag to include external comment syntax (optional)

3 Results

To demonstrate the usability and capability of Synplifier a parser is build using the automated parser generator based on an the extracted CNF TPTP sub-syntax. parsing only a sub-syntax of the TPTP syntax. and counting the number of remaining rules is used. The parser is built using an automated parser generator [10]. The automated parser generator can be used with any sub-syntax generated by Synplifier, for example sub-syntaxes containing only FOF or CNF. The following sections exemplify the validation of Synplifier by extracting CNF.

3.1 Automated parser generation

To ensure compatibility with the automated parser generator Synplifier offers the possibility of exporting an extracted sub-syntax and adding the part of the syntax concerning comments (displayed in the following listing), even though it is not reachable in the original syntax. This is implemented by maintaining this part of the syntax in a configuration file. Synplifier creates a grammar graph from this part of the syntax with <*comment*> as start symbol and appends this syntax to the output of a generated sub-syntax. If this part of the syntax would not be present in the output sub-syntax file, the automated parser generation would fail because this syntax part is expected to be present.

```
<comment>             ::- <comment_line>|<comment_block>
<comment_line>        ::- [%]<printable_char>*
<comment_block>       ::: [/][*]<not_star_slash>[*][*]*[/]
<not_star_slash>      ::: ([^*]*[*][*]*[^/*])*[^*]*
<printable_char>      ::: .
```

The automated parser generator is used to check if the output sub-syntax follows the original TPTP syntax format.

[8] https://docs.python.org/3/library/argparse.html

3.2 Extracting and validating CNF

The CNF sub-syntax has been validated by using the automated parser generator. It is validated by testing if the generated parser accepts TPTP problems written in CNF and returns an error when parsing a TPTP problem not written in CNF. In order to generate the parser, the CNF sub-syntax has to be extracted. The extraction is summarised in the following. The listing below contains the control file content that extracts CNF from the TPTP syntax version 7.3.0. The start symbol is <*TPTP_file*>.

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

All productions except the *cnf_annotated* are disabled from the *annotated_formula* grammar rule (line 2 of the listing above). The *annotated_formula* grammar rule can be seen in the following listing.

```
<annotated_formula>    ::= <thf_annotated> | <tff_annotated> |
                           <tcf_annotated> | <fof_annotated> |
                           <cnf_annotated> | <tpi_annotated>
```

Annotations are disabled as well. To extract the CNF sub-syntax based on the control file shown above, either the command-line interface or the GUI can be used. The parser has been generated using the CNF sub-syntax as input and the output Lex/Yacc parser has been modified to count CNF clauses. The generated parser has been successfully tested on all CNF TPTP problems. If the parser accepts the input, it counts the number of CNF clauses. It returns an error if the problem is written using other logics than CNF (100 problems of each logic FOF, TFF, TFA, higher order logic have been tested). The successful testing indicates that the CNF sub-syntax from which the parser has been generated was correctly extracted by Synplifier. The generated parser and the list of non CNF problems that have been used for testing can be found on GitHub[9].

3.3 Syntax size comparison

Based on the extracted CNF and FOF sub-syntax, the size of the sub-syntaxes in comparison to the full TPTP syntax has been analysed. Table 1 contains the number of rules and productions of the original TPTP syntax and extracted sub-syntaxes. A rule is the combination of left hand side non-terminal symbol, rule type and rule alternatives. Since rule alternatives are just a convenient way to display multiple rules with same non-terminal symbol on the left hand side and the same rule type, in addition to the number of rules the number of productions is compared. A production in this case is a rule alternative. The reachable part of the TPTP syntax in the second row of the table contains only the reachable symbols counted from the start symbol <*TPTP_file*>. Rules are counted by counting the nodes in grammar graph and the number of productions is counted by counting the productions list entries of all nodes.

Table 1 implies that the FOF sub-syntax contains only circa 35% of the rules of the full syntax. The CNF sub-syntax contains only circa 30% of the rules of the full syntax. Compared with the total number of reachable productions in the TPTP syntax, FOF contains circa 33% of the total reachable productions and CNF contains circa 28% of the total reachable productions. This shows a significant syntax reduction, which is one goal of using Synplifier.

[9] https://github.com/nahku/TPTP-CNF-Parser

Table 1: TPTP syntax size comparison

Syntax	Rules	Productions
TPTP syntax v. 7.3.0	313	635
Reachable part of TPTP syntax v. 7.3.0	285	595
FOF	108	198
CNF	92	165

4 Future Work

The concept of sub-syntax extraction is not only limited to the TPTP language and can also be applied to any other computer language, e.g., the SMT-LIB language [1]. In order to adopt Synplifier to extract sub-syntaxes from other languages, mainly the lexer and the parser components would have to be modified to parse the new syntax. Other components can be used with only minor modifications because the logic of the sub-syntax extraction would mostly remain the same. Furthermore, comment association would have to be adapted based on the specific comment convention (if comments are present in the syntax description).

It would also be possible to create a tool that accepts plain BNF or EBNF since many computer languages are described in these forms. This would make the developed tool a standard tool since it could then be used with any arbitrary computer languages that is described in BNF/EBNF. As the design of Synplifier is not limited to automated reasoning, mainly the lexing and parsing component have to be modified in order to parse plain BNF/EBNF.

Apart from adopting Synplifier to other languages, it can be investigated if comment association based on the comments content would lead to a significantly better result than using the heuristic approach that we have introduced.

5 Conclusion

This paper has described a tool called Synplifier that automatically extracts sub-syntaxes from the TPTP syntax. Synplifier includes lexing and parsing the TPTP syntax, building a graph that represents the syntax and extracting a sub-syntax based on the graph. The user can interact with Synplifier using a command-line interface or a GUI. The functionality of Synplifier has been demonstrated by extracting the CNF sub-syntax and creating a parser based on that syntax, which has been successfully tested with a number of TPTP problems.

Synplifier reduces the entry barrier for new users of the TPTP language because desired sub-syntaxes of the TPTP syntax can be extracted while maintaining consistency with the original TPTP syntax. In short, Synplifier enables users to consistently, conveniently, and automatically extract sub-languages from the TPTP syntax, turning this one language into a family of languages, coming from a single source, but optimized for particular use cases.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[2] Jasmin Christian Blanchette and Andrei Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. In Maria Paola Bonacina, editor, *Proc. of the 24th CADE, Lake Placid*, volume 7898 of *LNAI*, pages 414–420. Springer, 2013.

[3] T. Gleissner, A. Steen, and C. Benzmüller. Theorem Provers for Every Normal Modal Logic. In T. Eiter and D. Sands, editors, *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 46 in EPiC Series in Computing, pages 14–30. EasyChair Publications, 2017.

[4] Cezary Kaliszyk, Geoff Sutcliffe, and Florian Rabe. TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism. In P. Fontaine, S. Schulz, and J. Urban, editors, *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*, number 1635 in CEUR Workshop Proceedings, pages 41–55, 2016.

[5] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[6] G. Sutcliffe and C. Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[7] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-order Form with Arithmetic. In N. Bjørner and A. Voronkov, editors, *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 7180 in Lecture Notes in Artificial Intelligence, pages 406–419. Springer-Verlag, 2012.

[8] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[9] Geoff Sutcliffe and Evgenii Kotelnikov. TFX: The TPTP extended typed first-order form. In Boris Konev, Josef Urban, and Philipp Rümmer, editors, *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning (PAAR), Oxford, UK*, number 2162 in CEUR Workshop Proceedings, pages 72–87, 2018.

[10] A. Van Gelder and G. Sutcliffe. Extending the TPTP language to higher-order logic with automated parser generation. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd IJCAR, Seattle*, number 4130 in LNAI, pages 156–161. Springer, 2006.