Automating Scientific Paper Screening with Backus-Naur Form (BNF) Grammars

Radu Bucea Manea Tonis[1]

Abstract: This article explores BNF grammars to streamline the automated screening of scientific papers. BNF grammars define the valid structures and sentence formats for a specific language. They act like a rulebook, ensuring consistency and allowing for analysis of elements within the language. In this case, the BNF grammar serves as a blueprint for the proper formatting of scientific papers according to publisher guidelines. The grammar defines the acceptable structure and arrangement of elements like preamble, body, and paragraphs. By leveraging a tool like Java ANTLR v.4 runtime, we can convert the BNF grammar into a custom parser. This parser can then automatically assess submitted scientific papers, verifying if they adhere to the defined formatting rules established by the BNF grammar. This approach offers a way to automate the initial screening process for scientific papers, potentially saving time and improving consistency during the review process.

Keywords: formal grammars; BNF; Template Meta-Programming (TMP)

1. Introduction

Context-Free Grammar (CFG) serves as a formal system for defining the syntax (structure) of languages. Its primary focus lies in the arrangement of symbols to form valid sentences, rather than the meaning, making it particularly useful for delineating programming languages, data formats, and certain aspects of natural languages. CFG

[1] Senior professor, PhD, School of Behavioral and Applied Sciences - Informatics, Danubius International University of Galati, Romania, Address: 3 Galati Blvd., 800654 Galati, Romania, Corresponding author: radumanea@univ-danubius.com.

can be represented as a 4-tuple, consisting of:

$G = (N, \Sigma, P, S)$,

where:

1. N (non-terminal symbols) - Denote abstract elements that can be substituted by other symbols based on production rules.

2. Σ (terminal symbols) - The fundamental building blocks, such as letters, numbers, or punctuation marks.

3. P (production rules) - Specify how non-terminal symbols are transformed into strings of terminals and non-terminals. Each rule consists of a left-hand side (one non-terminal) and a right-hand side (a string of symbols).

4. S (start symbol) - A special non-terminal symbol that serves as the starting point for generating sentences.

As per the findings by Iwashokun & Ade-Ibijola (2024), it is established that S belongs to N, and P constitutes a finite set of formulas in the form of $A \longrightarrow \alpha$, where A belongs to N and α belongs to $(N \cup \Sigma)*$. The terminals are representative of the alphabet, while the non-terminals denote the names of rules. A rule is represented as P::= α, where P denotes the rule's name, and α represents the production. The rule's name is a non-terminal, while the production is a sequence of terminals and non-terminals. In cases where a rule has multiple potential productions, they are segregated as illustrated by Vassor (2022).

S::= a | aS.

The concept of grammar involves a set of rules, with each rule, or production, defining a set of words. In simple terms, a production generates the words specified by the terminal, replacing non-terminals with productions from the corresponding rule. As an example, the rule S mentioned previously generates {a, aa, aaa, ...}. It's important to note that rules are capable of being mutually recursive, as illustrated in Figure 1.

```
<expr> ::= <variable>
         | <abstraction>
         | <application>

<variable> ::= [a-z]+

<abstraction> ::= "\\" <variable> "." <expr>

<application> ::= "(" <expr> <expr> ")"
                | "(" <expr> ")" <expr>
                | <expr> "(" <expr> ")"
```

Figure 1. BNF grammar of λ-calculus syntax, after radubm1/Lambda-calculus-parser (github.com)

As per (Collins, 2019), a context-free grammar (CFG) outlines a set of potential derivations. A string, denoted as $s \in \Sigma*$, belongs to the language specified by the CFG if there exists at least one derivation that produces s. It is important to note that each string in the language produced by the CFG might have multiple derivations, resulting in ambiguity. Some examples of current formalisms include minimalism, lexical functional grammar (LFG), head-driven phrase-structure grammar (HPSG), tree adjoining grammars (TAG), and categorial grammars.

2. Backus-Naur Form (BNF) Grammars

Backus-Naur Form (BNF) serves as a notation system for describing Context-Free Grammars (CFGs). It is not a grammar itself, but rather a method for articulating CFG rules. Both are fundamental tools in computer science and linguistics for expressing syntax. Labeled BNF, also known as LBNF, extends BNF by incorporating labels into grammar rules. In an LBNF grammar, each rule is assigned a label, typically in the form of an identifier. These labels serve as constructors for the abstract syntax tree (AST) that corresponds to the parsed structure[1]. LBNF offers a clearer relationship between the grammar and the AST. The labels explicitly delineate the subtrees that form the larger structure. This can be advantageous for tools that generate parsers or perform semantic analysis on the parsed code. Traditional BNF employs symbols like "< >" and "::=" to define production rules. LBNF retains these elements but introduces labels before the production definition[2]; refer to Figure 2 for further details.

[1] https://bnfc.digitalgrammars.com/.
[2] https://bnfc.readthedocs.io/en/latest/lbnf.html.

```
entrypoints Document, Par;
Progr . Document ::= Pre Abs [Key] Body;
ConstT . Title ::= "Paper Title";
ConstA . Author ::= "(c) by Name";
Preamble . Pre ::= Title "\n" Author;
CnstAK . Abs ::= "Abstract \n Keywords:";
Keywords . Key ::= "\t";
(: []) . [Key] ::= Key;
(:) . [Key] ::= Key ";" [Key];
Plain . Body ::= [Par];
Paragraph . Par ::= "¶";
(: []) . [Par] ::= Par;
(:) . [Par] ::= Par "\n" [Par];
```

Figure 2. LBNF grammar of paper.cf file

A more detailed interpretation was obtained by using the Railroad Diagram Generator (Rademacher, 2024), which is a tool for creating syntax diagrams, also known as railroad diagrams, from context-free grammars specified in EBNF. Please refer to Annex 1. In the LBNF example, “Progr.” serves as the label for the rule. This label becomes a constructor in the AST, representing an expression. The right-hand side defines the structure of a program. Please see Fig. 3 for details:

```
(MkGrammar [(Entryp ["Document", "Par"]), (Rule (LabNoP (Id "Progr")) (IdCat "Document")
[(NTerminal (IdCat "Pre")), (NTerminal (IdCat "Abs")), (NTerminal (ListCat (IdCat "Key"))),
(NTerminal (IdCat "Body"))]), (Rule (LabNoP (Id "ConstT")) (IdCat "Title") [(Terminal "Paper
Title")]), (Rule (LabNoP (Id "ConstA")) (IdCat "Author") [(Terminal "(c) by Name")]), (Rule
(LabNoP (Id "Preamble")) (IdCat "Pre") [(NTerminal (IdCat "Title")), (Terminal "\n"),
(NTerminal (IdCat "Author"))]), (Rule (LabNoP (Id "CnstAK")) (IdCat "Abs") [(Terminal
"Abstract \n Keywords:")]), (Rule (LabNoP (Id "Keywords")) (IdCat "Key") [(Terminal "\t")]),
(Rule (LabNoP ListOne) (ListCat (IdCat "Key")) [(NTerminal (IdCat "Key"))]), (Rule (LabNoP
ListCons) (ListCat (IdCat "Key")) [(NTerminal (IdCat "Key")), (Terminal ";"), (NTerminal
(ListCat (IdCat "Key")))]), (Rule (LabNoP (Id "Plain")) (IdCat "Body") [(NTerminal (ListCat
(IdCat "Par")))]), (Rule (LabNoP (Id "Paragraph")) (IdCat "Par") [(Terminal "¶")]), (Rule
(LabNoP ListOne) (ListCat (IdCat "Par")) [(NTerminal (IdCat "Par"))]), (Rule (LabNoP
ListCons) (ListCat (IdCat "Par")) [(NTerminal (IdCat "Par")), (Terminal "\n"), (NTerminal
(ListCat (IdCat "Par")))])])
```

Figure 3. Abstract Syntax Tree (AST) obtained by validating our grammar

There are two primary approaches to extracting information from textual sources, as described by Iwashokun & Ade-Ibijola (2024):

- The rule-based approach relies on predefined standards or templates to identify specific details in documents by analyzing text font styles. This technique has been applied in previous studies to process various document types, including invoices, legal documents, and CVs.

- Recent developments in big data and text analytics have significantly improved information extraction. The machine-based approach involves using machine

learning (ML), deep learning (DL), or large language models (LLM) instead of formal or rule-based methods.

According to Paslaru (2024), Estonia is developing state-service chatbots called Bürokratt. Unlike other chatbots, Estonian chatbots utilize natural language processing (NLP) to deconstruct requests and identify keywords to understand user intent.

3. Parsing Grammar Using Template Metaprogramming (TMP)

In 1994, at a C++ committee meeting, Erwin Unruh from Siemens presented a program that became famous for its failure to compile. The program, which utilized the Synopsys ARC MetaWare, was created to compute the first prime numbers up to 30 at compile time, demonstrating the capability of template instantiation to carry out mathematical operations during compilation (Grimm, 2021). According to[1], the use of TMP involves both defining and instantiating templates. The template definition outlines the generic form of the produced source code, while instantiation employs this form to generate specific source code. Many template implementations use recursion for flow control. TMP is Turing-complete, enabling us to tackle a wide array of computational problems. It does not involve mutable variables, statements, or loops, resembling functional programming languages such as Lisp or Haskell. There are no runtime costs associated with TMP, and through this sub-language, we can generate code, manipulate types, and perform computations on them or other templates (Andrivet, 2014).

In the study conducted by Whitney & Ibanez (2021), it is mentioned that a meta-function manipulates specific types or values to generate other types or values. It is worth noting that there is no requirement to instantiate the structure. When there are two template classes declared, the Class Template Argument Deduction (CTAD) Rules adhere to a specific ranking system, prioritizing more specialized templates. In our grammar, an abstract base class is created for each category, and a class is extended from the base class for each category constructor. Consequently, multiple files are placed into the subdirectory paper/Absyn. In addition, a Visitor interface and an abstract accept method are generated for each category in the grammar. Finally, an accept method overrides the abstract accept for each category constructor.

[1] https://en.wikipedia.org/wiki/Template_metaprogramming.

The program utilizes the visitor skeleton by default, with Integer as the return type and Object as the "dummy" argument type, following the information from[1].

Stages in parsing the scientific papers by our proposed grammar:

1. Creating the parser files using BNF Converter (bnfc 2.9.5): bnfc --java-antlr -m Paper.cf

2. Building the PaperParser.java with make command, please see Annex 2.

3. Running the parser on article.txt:

Paper Title

(c) by Name

Abstract

Keywords: ; ;

The result is presented below, please see Fig. 4:

```
Parse Successful!

[Abstract Syntax]

(Progr (Preamble ConstT ConstA) CnstAK [Keywords, Keywords, Keywords] (Plain [Paragraph,
Paragraph]))

[Linearized Tree]

Paper Title (c) by Name Abstract Keywords:;;
¶ ¶
```

Figure 4. Parsing article.txt with the scientific grammar parser (PaperParser.java)

To efficiently represent arithmetic expressions, one can leverage nested templates to delegate expression analysis to the template processor. This strategy offers improved speed as the modified expression tree becomes available to the optimizer at the same level as the rest of the code, enabling comprehensive optimization within and across expressions and the surrounding code, as noted in a source[2]. Another viable approach involves developing a domain-specific language (DSL) for the expressions and

[1] https://bnfc.digitalgrammars.com/tutorial/bnfc-tutorial.html.

[2] https://stackoverflow.com/questions/63494/does-anyone-use-template-metaprogramming-in-real-life.

integrating the translated code into the regular program. This alternative also delivers optimization advantages and enhanced readability, albeit requiring the maintenance of a parser.

4. Conclusion

In summary, LBNF offers the following benefits:

- Clearer AST representation: Labels directly correspond to the structure of the AST.

- Improved parser generation: Tools can utilize labels to construct the AST more efficiently.

- Enhanced semantic analysis: Labels can assist in establishing the meaning associated with various parts of the parsed structure.

References

Andrivet, S. (2014). C++11 metaprogramming applied to software obfuscation. *Black Hat Europe 2014 - Amsterdam.* URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Andrivet-C-plus-plus11-Metaprogramming-Applied-To-software-Obfuscation-wp.pdf.

Collins, M. (2019). *Parsing and Context-Free Grammars.* Columbia University. Retrieved from https://www.cs.columbia.edu/~mcollins/cs4705-spring2019/slides/parsing1.pdf, date: July 10, 2024.

Grimm, R. (2021). *Template Metaprogramming – How it All Started.* Retrieved from https://www.modernescpp.com/index.php/template-metaprogramming-a-introduction.

Iwashokun, O. & Ade-Ibijola, A. (2024). Parsing of Research Documents into XML Using Formal Grammars - Applied Computational Intelligence and Soft Computing. *Wiley Online Library*.

Pissarro, B. (2024). *O nouă realitate. Inteligența artificială schimbă profund omenirea / A new reality. Artificial intelligence is profoundly changing humanity*. Retrieved from https://evz.ro/inteligenta-artificiala-solutie-servicii-guvernamentale-bune.html.

Rademacher, G. (2024). *Railroad Diagram Generator - A tool for creating syntax diagrams.* Retrieved from https://rr.red-dove.com/ui.

Vassor, M. (2022). *The formal-grammar package*. Retrieved from https://tug.org/docs/latex/formal-grammar/formal-grammar.pdf.

Whitney, H. & Ibanez, F. (2021). *Template Metaprogramming; CS 106L.* Retrieved from https://web.stanford.edu/class/cs106l/lectures/16_tmp.pdf, date: July 17, 2024.

Source online

www1; The Labelled BNF Grammar Formalism; https://bnfc.digitalgrammars.com/.

www2; LBNF reference; https://bnfc.readthedocs.io/en/latest/lbnf.html.

www3; https://en.wikipedia.org/wiki/Template_metaprogramming.

www4; https://bnfc.digitalgrammars.com/tutorial/bnfc-tutorial.html.

www5; c++ - Does anyone use template metaprogramming in real life? https://stackoverflow.com/questions/63494/does-anyone-use-template-metaprogramming-in-real-life.

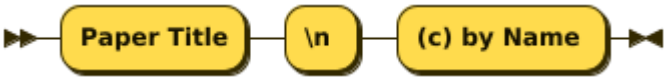
Annex 1

Document:

Document::=Pre ‘Abstract \n Keywords:’ KeyBody

no references

Pre:

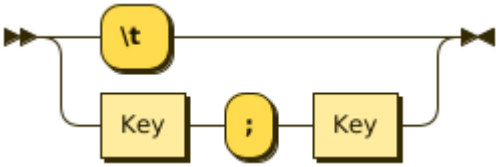


Pre::= “Paper Title” “\n” “(c) by Name”

referenced by:

1. Document

Key:



Key::= ‘\t’

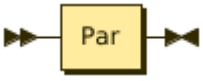
|Key’;’Key

referenced by:

1) Document

2) Key

Body:

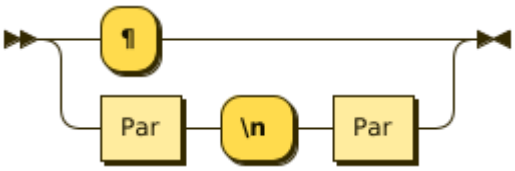


Body::=Par

referenced by:

1) Document

Par:



Par::= '¶'

|Par'\n'Par

referenced by:

2. Body
3. Par

Annex 2

```
package paper;
import paper.*;
import java.io.*;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.atn.*;
import org.antlr.v4.runtime.dfa.*;
import java.util.*;
public class Test
{
PaperLexer l;
PaperParser p;
public Test(String[] args)
{
try
{
```

```
Reader input;
if (args.length == 0) input = new InputStreamReader(System.in);
else input = new FileReader(args[0]);
l = new PaperLexer(new ANTLRInputStream(input));
l.addErrorListener(new BNFCErrorListener());
}
catch(IOException e)
{
System.err.println("Error: File not found: " + args[0]);
System.exit(1);
}
p = new PaperParser(new CommonTokenStream(l));
p.addErrorListener(new BNFCErrorListener());
}
public paper.Absyn.Document parse() throws Exception
{
/* The default parser is the first-defined entry point. */
/* Other options are: */
/* par */
PaperParser.Start_DocumentContext pc = p.start_Document();
paper.Absyn.Document ast = pc.result;
System.out.println();
System.out.println("Parse Successful!");
System.out.println();
System.out.println("[Abstract Syntax]");
System.out.println();
System.out.println(PrettyPrinter.show(ast));
```

```
System.out.println();
System.out.println("[Linearized Tree]");
System.out.println();
System.out.println(PrettyPrinter.print(ast));
return ast;
}
public static void main(String args[]) throws Exception
{
Test t = new Test(args);
try
{
t.parse();
}
catch(TestError e)
{
System.err.println("At line " + e.line + ", column " + e.column + " :");
System.err.println(" " + e.getMessage());
System.exit(1);
}
}
}
```