CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Developing a Back-end Compatibility Check for the BNF Converter

And Implementing a Blazingly-fast BNFC Back-end in Rust

Master's Thesis in Computer Science and Engineering

Jonathan Widén
Leopold Wigbratt

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2025

Master's Thesis 2025

Developing a Back-end Compatibility Check for the BNF Converter

And Implementing a Blazingly-fast BNFC Back-end in Rust

Jonathan Widén
Leopold Wigbratt

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2025

Developing a Back-end Compatibility Check for the BNF Converter
And Implementing a Blazingly-fast BNFC Back-end in Rust
Jonathan Widén
Leopold Wigbratt

© Jonathan Widén, Leopold Wigbratt, 2025

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31-772 10 00

Typeset in Typst
Gothenburg, Sweden, 2025

Developing a Back-end Compatibility Check for the BNF Converter
And Implementing a Blazingly-fast BNFC Back-end in Rust
Jonathan Widén
Leopold Wigbratt
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

There exist various architectural models for structuring large-scale software projects. A common pattern employed in various systems involves decomposing the application into a user-facing front-end component, responsible for input validation, and several back-end components, which perform subsequent processing and generate output. In such a system, each back-end may target a distinct output format. Communication between the front-end and the back-ends is typically conducted by means of a well-defined interface.

It is often desired that the front-end be responsible for all input validation, so as to promote encapsulation and enforce uniform communication to the user. However, difficulties may arise when the back-ends impose different requirements on the input that they process.

This thesis examines this problem in the context of the BNF Converter – a compiler construction tool that transforms a common grammar specification into software components used for implementing a compiler for the language of said grammar. The BNF Converter adheres to the aforementioned architectural model and supports multiple back-ends that target various output formats and are characterized by distinct capabilities and requirements with respect to the input.

In order to address these discrepancies, a so-called *back-end compatibility check* was developed. This check enables the front-end to validate whether a given input conforms to the requirements imposed by a particular back-end. The methodology used for this purpose is expected to generalize to other systems exhibiting a similar structure. Additionally, a new back-end was implemented, both to evaluate the usefulness of the interface and to contribute to the development of the BNF Converter itself.

Keywords: BNF, Programming Languages, Lexing, Parsing, Parser Generators

Acknowledgements

The authors would like to express their deepest gratitude to their supervisor, Andreas Abel, for invaluable guidance and orientation within the area of the thesis in general and BNFC in particular. His expertise, feedback, and willingness to discuss multiple possible directions, led to a great final result. The authors would also like to thank Carl-Johan Seger for acting as an examiner, and showing interest in the work.

Moreover, the authors would like to thank their friends and families, Hannah Sundvall in particular, for their continued support throughout the project.

Jonathan Widén and Leopold Wigbratt, Gothenburg, 2025-06-05

Contents

List of Figures

List of Grammars

List of Listings

List of Tables

1

Introduction

Large software projects can be structured in accordance with various architectures. *Monolithic* applications are common in practice, but may lack the flexibility that is required for certain projects [1]. Instead, by means of *encapsulation*, a codebase can be divided into separate components, which interact by means of a well-defined *interface*.

An instance of an architecture of this kind is that of an application consisting of a user-facing *front-end* component and multiple opaque *back-end* components. The responsibility of the former is that of handling the input to the program supplied by the user and conveying it to a designated back-end for further processing. This architecture lends itself to software that is intended to be extensible and to which future additions, especially with regard to back-end components, are desired. It is thus found in software serving various purposes, amongst which compilers are prime examples [2]; the GNU Compiler Collection [3] and the *BNF Converter*[1] both employ this technique.

This thesis investigates how a mechanism, known henceforth as a *back-end compatibility check*, tasked with validating the input of an application with regard to its compatibility with a user-selected back-end, can most proficiently be devised. In doing so, a back-end compatibility check and an interface by which the check is conducted have been implemented for the BNF Converter.

1.1 Research questions

Q1. The first research question explored in this thesis is: How to design a back-end compatibility check, and the interface which it is controlled by? Can this be done whilst keeping the benefits of encapsulation, namely, that features may be added to both the front-end and back-end without breaking the other parts?

This question shall be explored in this thesis in the context of the BNF Converter, for which the implementation of such a check has been implemented. The BNF Converter is a compiler construction tool that transforms a general grammar format into programming-language-specific ones, each controlled by a separate back-end, that is to be used as part of a compiler.

[1]The BNF Converter has previously exhibited a monolithic structure, but its codebase has been recently rewritten utilizing the front-end and back-end model, in an effort to further its extensibility and maintainability.

In order to explore the utility and limitations of the back-end compatibility check, a new back-end for the BNF Converter has been developed, targeting the Rust programming language. Not only shall this new back-end be used to evaluate the new interface, it shall also be the subject of a second research question:

Q2. How can an efficient abstract syntax tree that is to be generated from a general grammar format be implemented for the Rust programming language?

1.2 Objectives

A number of objectives have been defined for this project.

> **O1.** *A back-end compatibility check shall be developed for the BNF Converter.*

Several properties by which this check should be characterized have been identified, namely those that follow.

P1. The source code relating to the execution of the back-end compatibility check shall be located in the code of the front-end; no further validation shall be performed by the back-ends.
P2. The behaviour of the back-end compatibility check shall be controlled by a designated back-end through the use of an *interface*.
P3. The back-end compatibility check shall be sufficiently expressive that
  (a) no further validation would be required for any input successfully passing the check and
  (b) no back-end specific knowledge, besides that specified through the interface, be required in order for the check to be performed in the front-end.
P4. The interface through which the back-end compatibility check is conducted shall enable a consistent and uniform means of communication to the user.

> **O2.** *The codebase of BNFC shall be restructured, so as to further its encapsulation and to limit its coupling.*

This shall primarily be accomplished through the separation of the front-end codebase from that of the back-ends.

> **O3.** *A back-end for BNFC targeting the Rust programming language shall be developed.*

This back-end shall be developed in close association with the back-end compatibility check, and it shall utilize the interface of this check. Furthermore, special care shall be given to development of an abstract syntax representation in Rust, as stipulated by **Q2**.

1.3 Structure of the thesis

Chapter 2 *Background* provides the reader with adequate preliminaries on compilation pipelines in general and parsing and lexing in particular, so as to render the remaining content of the thesis more comprehensible. The chapter continues with both previous and recent developments of BNFC, the subject of research, and on LBNF – the input format of BNFC.

Chapter 3 *The Interface* relates the process of defining and designing a back-end compatibility check and its accompanying interface, in addition to motivations for the design choices taken. Moreover, this chapter includes the research conducted in order to identify how the back-end compatibility check and interface were to be constituted. This chapter contains the result of objectives **O1** and **O2**.

Chapter 4 *The Rust Back-end* presents the new back-end, which was developed as a means to evaluate the interface and aid in its development. This chapter focuses mainly on the back-end in isolation, in regards to what code it does produce. It also explores different implementations of abstract syntax trees, which is the result of objective **O3**.

Chapter 5 *Evaluation* assesses the back-end compatibility check in relation to objective **O1**, containing discussions on each property **P1** through **P4**. The work dictated by **O2** is also evaluated. Furthermore, the Rust back-end presented in Chapter 4 *The Rust Back-end* is evaluated, both in terms of its utilization of the back-end compatibility check and the new interface, as well as in how the back-end itself performs and the features it implements, compared to other back-ends. The latter will be done in relation to objective **O3**.

Chapter 6 *Conclusions* summarizes the insights gained from the findings in Chapter 5 *Evaluation*, and contains discussion about topics of future work.

2

Background

This chapter covers fundamental topics for this thesis: formal grammar, anatomy of compilers, parsing, and the BNF Converter.

2.1 Grammars

A *formal language* is a, possibly infinite, set of sequences of symbols drawn from an *alphabet*; a *formal grammar* consists of a set of rules that govern which sequences of symbols of the alphabet constitute the *sentences* of the language that the grammar defines. For the purpose of illustration, a grammar could be defined for a language consisting of all strings of digits terminating with the digit 3 or all strings that are palindromes.

Formally, a grammar G is denoted by a four-tuple $G = (N, T, P, S)$, where N is a finite set of *non-terminal symbols*, T is a finite set of *terminal symbols*, P is a finite set of *production rules*, and $S \in N$ is the *starting symbol*. The set of terminal symbols, or alphabet, T is disjoint from N. Grammars can be classified according to the type of production rules that they contain. One such commonly-used classification is the one of Chomsky [4], which defines a hierarchy of grammars, as seen in Figure 1.

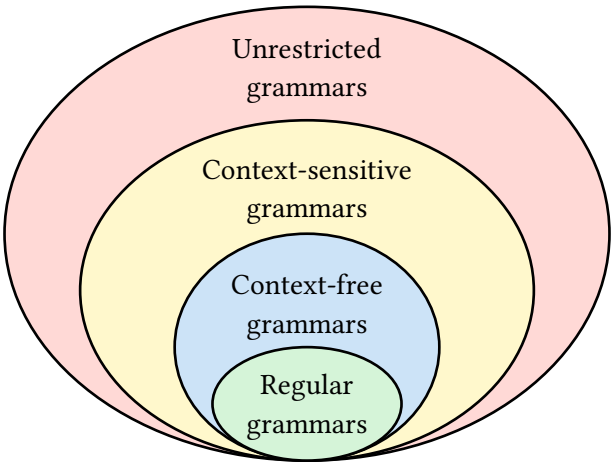


Figure 1 : The Chomsky hierarchy of formal grammars

In this hierarchy, the regular grammars are the most restricted, and thus, the least expressive. Recognizing sentences of this grammar can be done very efficiently. Regular grammars are equivalent to regular expressions, which are very commonly used in searching for patterns in text by command-line utilities, such as `grep`, as well as in text based programs, such as text editors. Context-free grammars are less restricted, and hence more expressive, than regular grammars, and a certain subset of these grammars can be efficiently recognized. Context-sensitive grammars are very powerful but generally hard to recognize; recognizing a string to be generated by an unrestricted grammar is equivalent to running any arbitrary program on a Turing machine [5].

For the purpose of defining a grammar for a computer language, whilst the class of the grammar employed must be sufficiently expressive in order to express the desired sentences, an excessively complex grammar is to be avoided on account of difficulties that may arise when utilizing such grammars in computer programs. Thus, most computer languages, such as programming languages, are defined in terms of context-free grammars, for the reason being that it is the least complex class of grammars that is sufficient for the task.

2.1.1 Context-free grammars

A context-free grammar is formally defined by a four-tuple $G = (N, T, P, S)$ where the production rules P are relations between a non-terminal and a sequence of non-terminal and terminal symbols: $N \times (N \cup T)^*$. The production rules shown in Grammar 1 denote palindromes over the alphabet that consists of the letters a and b. The left-hand side of a production rule may only consist of a single non-terminal, whilst the right-hand side may consist of any sequence of terminal or non-terminal symbols, including the empty sequence, denoted ε.

$$
\begin{aligned}
S &\to \varepsilon \\
S &\to aSa \\
S &\to bSb
\end{aligned}
$$

Grammar 1 : A context-free grammar recognizing palindromes in the alphabet $T = \{a, b\}$

All sentences recognized by a grammar must be formed by a valid *derivation*, a sequence of applications of production rules, in which the right-hand side of a rule is substituted for its left-hand side, beginning with the starting symbol S. One such application may be regarded as a binary relation, denoted by $\to$. The derivation of the sentence $abba$ can be represented by three applications of the production rules in Grammar 1, namely: $S \to aSa \to abSba \to abba$.

$$
\begin{array}{lll}
D \to 0 & N \to D & M \to P \\
D \to 1 & N \to DN & M \to P * M \\
\cdots & P \to N & A \to M \\
D \to 8 & P \to (A) & A \to M + A \\
D \to 9 &
\end{array}
$$

Grammar 2 : A context-free grammar recognizing arithmetic expressions

A sequential application of production rules, wherein, at each derivation step, the right-most non-terminal is substituted is called a *right-most derivation*; a derivation formed by left-most substitutions is considered a *left-most derivation*.

The production rules for a context-free grammar recognizing additive and multiplicative expressions is given in Grammar 2. The non-terminal D represents a single digit, N one or more digits, P a number or parenthesized expression, A an addition, and M a term. The sentence $4 * (3 + 15)$ may be derived from the starting symbol A, of which the left-most and right-most derivations are demonstrated in Figure 2. Whilst these two methods of derivation yield equivalent sentences, both methods exhibit different properties that may be exploited by algorithms devised to recognize sentences of a certain grammar.

By *leftmost derivation*:

$$
\begin{aligned}
A \Rightarrow\ & M \Rightarrow\ P * M \Rightarrow\ N * M \\
\Rightarrow\ & D * M \Rightarrow\ 4 * M \Rightarrow\ 4 * P \Rightarrow\ 4 * (A) \\
\Rightarrow\ & 4 * (M + A) \Rightarrow\ 4 * (P + A) \Rightarrow\ 4 * (N + A) \Rightarrow 4 * (D + A) \\
\Rightarrow\ & 4 * (3 + A) \Rightarrow\ 4 * (3 + M) \Rightarrow\ 4 * (3 + P) \Rightarrow\ 4 * (3 + N) \\
\Rightarrow\ & 4 * (3 + DN) \Rightarrow 4 * (3 + 1N) \Rightarrow 4 * (3 + 1D) \Rightarrow\ 4 * (3 + 15)
\end{aligned}
$$

By *rightmost derivation*:

$$
\begin{aligned}
A \Rightarrow\ & M \Rightarrow\ P * M \Rightarrow\ P * P \\
\Rightarrow\ & P * (A) \Rightarrow\ P * (M + A) \Rightarrow\ P * (M + M) \Rightarrow\ P * (M + P) \\
\Rightarrow\ & P * (M + N) \Rightarrow P * (M + DN) \Rightarrow P * (M + DD) \Rightarrow P * (M + D5) \\
\Rightarrow\ & P * (M + 15) \Rightarrow\ P * (P + 15) \Rightarrow\ P * (N + 15) \Rightarrow\ P * (D + 15) \\
\Rightarrow\ & P * (3 + 15) \Rightarrow\ N * (3 + 15) \Rightarrow\ D * (3 + 15) \Rightarrow\ 4 * (3 + 15)
\end{aligned}
$$

Figure 2 : Left-most and right-most derivations of the sentence $4 * (3 + 15)$ by means of the production rules of Grammar 2

2.1.2 Backus-Naur form

Backus-Naur form (BNF) is a textual format used to specify context-free grammars and is commonly employed for defining computer languages. A grammar in the format of BNF equivalent to that of Grammar 2 is demonstrated by Grammar 3.

```
<digit>  ::= "0" | "1" | "2" | "3" | "4" |
             "5" | "6" | "7" | "8" | "9"
<number> ::= <digit> <number>      | <digit>
<par>    ::= "(" <add> ")"         | <number>
<mul>    ::= <mul> "*" <par>       | <par>
<add>    ::= <add> "+" <mul>       | <mul>
```

Grammar 3 : The production rules of Grammar 2 represented in BNF

The popularity of BNF has led to various extensions, such as Extended BNF (EBNF), Augmented BNF (ABNF), Translational BNF (TBNF), and Labelled BNF (LBNF) [6]. The Labelled BNF format, which is of particular interest for this thesis, is described in further detail in Section 2.4.

2.2 Compilers and interpreters

A compiler, in its most basic sense, is a computer program that translates a program specification from one format to another. More specifically and more commonly, the term is used to denote a program that translates source code, often in the format of a high-level programming language, into an executable format. The output format is typically that of *machine code*, which is to be executed directly by the processor, or *byte code*, a format recognized by a *virtual machine*. The output of a compiler for the C programming language [7] is an example of the former, and the Java compiler [8], which produces byte code to be run on the Java Virtual Machine [9], is an instance of the latter.

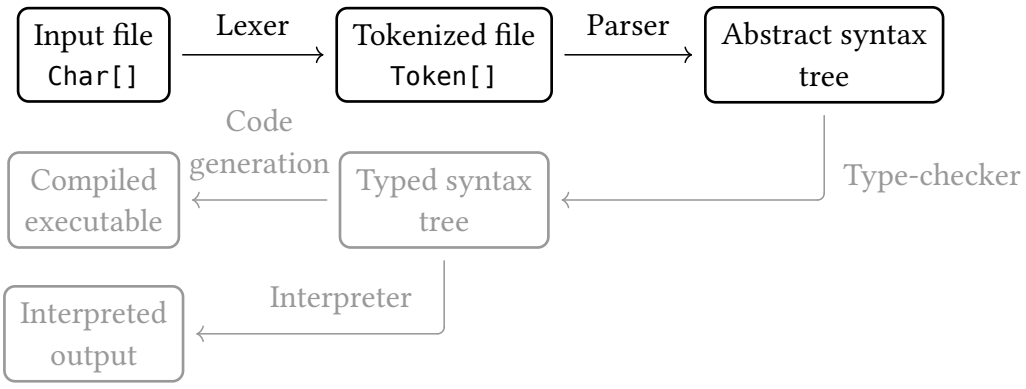


Figure 3 : The anatomy of a compiler and an interpreter

The general structure of a compiler and the stages of which it consists is demonstrated in Figure 3. The input, usually in the form of text based source code, is acted upon by a *lexer*, a

component that produces a sequence of *tokens*. The tokens are subsequently operated upon by a *parser*, which produces a structural representation of the input format. The behaviour and structure of these two components, the lexer and the parser, which validate the input in terms of its syntactic structure, are governed by the grammar of the source language. Then, depending on the source language for which the compiler is designed, it may be passed through a *type checker*, which validates the output of the parser in terms of the semantics of the source language, such as that of a type system. Finally, in the ultimate stage, namely that of *code generation*, the obtained structural representation is converted into the target format.

An *interpreter* is a program that is closely related to a compiler. Rather than converting the input into a low-level format that is to be executed at a later point in time, an interpreter executes the program directly. As seen in Figure 3, the structure of an interpreter is nevertheless closely linked to that of a compiler.

2.2.1 Lexing

A *lexer* is an algorithm that converts a sequence of symbols into a sequence of *tokens*. A token, which may consist of several distinct symbols, such as characters in a textual input, typically represent the smallest meaning-bearing unit of a language. As such, it may be more expediently employed as the input of a parser, for the reason being that the parser need not consider abstract symbols that carry no significance on their own, but larger units conveying meaning partly independent of their context. When a sequence of tokens is employed as the input format of a parser, a token constitute a terminal symbol in the alphabet of the language that the parser operates on. For the purpose of example, a sequence of characters, acting as the input of a lexer, and a sequence of tokens, its output, are illustrated in Figure 4.

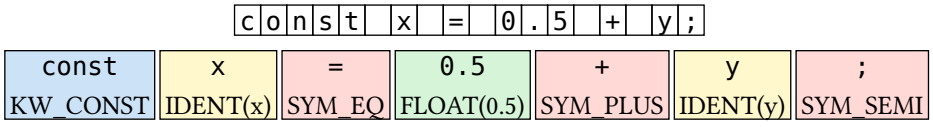


Figure 4 : A sequence of characters and its tokenized representation

The purpose of the lexer is to process the input such that the parser becomes smaller and faster, by handling the work that can be done by regular grammars. This reduces the work for the less efficient context-free grammar-based parser. The lexer often handles more than pure tokenization, such as removing insignificant whitespace and code comments, and converting numeric literals from text to values.

Note that in the derivation rules of Grammar 2, as well as the grammar Grammar 3, parsing numbers takes up a significant part of the grammar, even though it is the least complicated part. If this was done already before reaching the grammar, this would not be a problem. In addition, it effectively allows the parser to see further into the input: instead of seeing just one digit of a number literal, it could already know if it is an integer literal or a decimal-number literal.

2.2.2 Parsing

Parsing denotes a process in which information arranged in a linear form is analysed according to some grammar [10], which is also known as *syntactic analysis*. The term may equally denote such processes in various fields, e.g. computer science and traditional grammar. In the latter, it denotes the division of a phrase into its constituent parts of speech and the analysis of their syntactic relations; Latin *pars (ōrātiōnis)* 'part (of speech)' is the ultimate etymon of the term [11].

In the field of computer science, a *parser* is a program, or component thereof, that, applied to a linear input form, produces, as its output, a structural representation of the input reflecting the structure defined by some grammar. The input of a parser is generally a sequence of symbols or characters, and its output is a hierarchical data structure, most commonly a kind of tree. A tree structure employed for this purpose can directly correspond to the grammar, in which case it is a *concrete syntax tree*, also known as a *parse tree*. It may also exclude parts of the grammar that do not contribute to the structure of the tree, e.g. means for encoding operator precedence and terminal symbols representing punctuation or grouping, the meaning of which is often retained by the structure of the tree, in which case it is known as an *abstract syntax tree* (AST) [12].

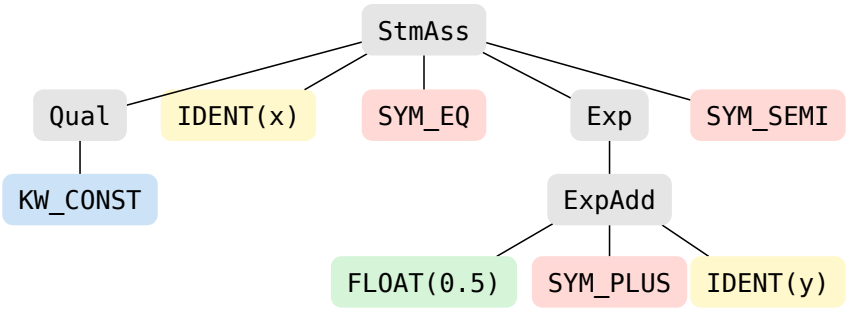


Figure 5 : A parse tree for an assignment statement corresponding to the sequence of tokens in Figure 4

The sequence of tokens illustrated in Figure 4 could constitute an assignment statement in some programming language. A parse tree for a statement of this kind, with respect to some hypothetical grammar, is demonstrated in Figure 5. Likewise, an equivalent abstract syntax tree is shown in Figure 6. It it to be noted that the tokens `SYM_EQ`, `SYM_PLUS`, and `SYM_SEMI`, which are terminal symbols in the context of this grammar, are not present in the AST, inasmuch as they provide no additional information beyond that which can be inferred from the non-terminal symbols and their respective production rules. The terminal symbols `IDENT(x)`, `FLOAT(0.5)`, and `IDENT(y)`, on the other hand, contain complementary data, which cannot be inferred from the remaining structure. Thus, they may not be excluded from the abstract syntax tree; they bear information that is semantically important. This type of symbol is known as an *annotated terminal* and is part of what can be described as an extension to context-free grammars [13]. The terminal symbol `KW_CONST` is also retained from the parse tree in order to disambiguate from which production rule the non-terminal symbol `Qual` is derived,

provided that there are more than one production rule for the non-terminal in question, e.g. `Qual` → `KW_CONST` | `KW_VAR` | `Type`.

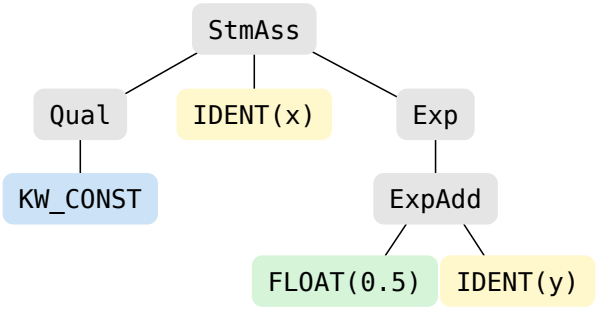


Figure 6 : An abstract syntax tree equivalent to the parse tree in Figure 5

2.2.3 LL(k) and LR(k) grammars

Donald Knuth [14] introduced the concept of LR(k) grammars, a subset of the context-free grammars for which he devised a parsing algorithm exhibiting linear time complexity that processes its input left-to-right and produces a *right-most* derivation. Moreover, the constant k denotes a *right-bound*, the maximum number of symbols following those that have been parsed with which the algorithm must deterministically decide upon the next course of action. This is more commonly referred to as a *lookahead*. This algorithm is of particular importance to the application of parsing in computer science, for the reason being that the most common input form to a parser, i.e. strings of text, is inherently structured left-to-right[2].

In contrast to these, there also exist LL(k) grammars, the algorithms for which parse their input left-to-right, but produce *left-most* derivations. The LL(1) grammars are properly contained within the LR(0) grammars[3], thus rendering the LR grammars inherently more expressive and interesting [15].

In practice, the parsing algorithm for LR(k) grammars devised by Knuth is implemented by means of representing state transitions with lookup tables. Thus, at each stage in the parsing algorithm, the subsequent action to be performed can be determined in constant-time by consulting the lookup table with the current state and the right-bound, i.e. the subsequent k symbols of the input. Nevertheless, these tables are typically very large for many grammars in practice [16], such as those of programming languages, and the number of states grows exponentially with respect to the right-bound k. As a consequence of this, right-bounds greater than $k = 1$ are seldom employed, and variants of the LR(k) parsing algorithm have been introduced in order to address the concern of large state tables.

[2]Typical text formats include source code written in a programming language and other types of structured data formats. These are almost universally arranged left-to-right, as is the case of writing utilizing the Latin script in general.

[3]More accurately, it is only the case for the LL(k) grammars containing no production rule of which the right-hand side is the empty string.

In 1969, DeRemer introduced the LALR(1) parsing algorithm [17]. It is based upon the LR(k) algorithm devised by Knuth, more specifically that of LR(1), but it reduces the number of states through the merger of equivalent states. The number of states produced by the LALR(1) algorithm is thus the same as that of the LR(0) algorithm, however with a larger lookup table. From this it follows that the set of languages recognized by the LALR(1) algorithm is a subset of those recognized by an LR(1) parser, whilst requiring the number of states of an equivalent LR(0) parser. Nonetheless, through the process of transforming the table of states from the LR(1) algorithm to the one of LALR(1), information with regard to the lookaheads is lost when states are merged. Thus, for certain grammars, the information retained in the LALR(1) state table is not sufficient to unambiguously parse sentences in the language of said grammar; not all LR(1) languages can be parsed by the LALR(1) algorithm.

2.3 Parser generators

As has been discussed above, efficient implementations of parsers for LR languages typically employ large lookup tables for most practical grammars. As an alternative to writing one by hand, a parser may be obtained by means of a *parser generator*. It is a software component that, when applied to a grammar specification, produces a parser for the corresponding grammar. Depending on what parser generator is used, the generated parser may be of different types. There are parser generators that generate full LR(1) parsers, but the large size of such parsers may render them somewhat impractical to be applied in practice; it is thus common for parser generators to generate other kinds of parsers, such as SLR(1), LALR(1), or other restricted LR parsers.

Most parser generators are designed to target one specific programming language; the generated parser is typically in the form of source code in a certain programming language. The parser may then be integrated into a larger codebase, such as that of a compiler or any other software requiring parsing capabilities.

There exist numerous parser generators[4], two of which are Yacc [18] and Happy [19]. Many parser generators require a grammar specification of a specific format. Two example grammar specifications, one for Yacc and the other for Happy, can be seen in Grammar 4. A grammar specified in the format of a particular parser generator may thus require a substantial amount of additional effort, in order to convert it to the specification format of another.

Since parsers are often used in conjunction with lexers, parser generators are often used in conjunction with *lexer generators*, which generate lexers from a lexer specification. Note, however, that writing a lexer by hand is less complicated than writing a parser.

[4]There are 78 deterministic parser generators for context-free languages listed in the Wikipedia article on the ‘Comparison of parser generators’.

```
%{
#define YYSTYPE double
void yyerror(const char* str) {
  fprintf(stderr, "error:%s\n", str);
}
int main(int argc, char* argv[]) {
  yyparse(); return 0;
}
%}
%token NUMBER ADD SUB MUL DIV LB RB RET
%%
S: S E RET { printf("%lf\n", $2); } | ;
E: E ADD T { $$=$1+$3; }
 | E SUB T { $$=$1-$3; }
 | T { $$=$1; } ;
T: T MUL F { $$=$1*$3; }
 | T DIV F { $$=$1/$3; }
 | F { $$=$1; } ;
F: LB E RB { $$=$2; }
 | NUMBER { $$=$1; } ;
%%
```

(a): A Yacc specification for a calculator [20][5]

```
{
module BNFC.Par where
import Prelude
import qualified BNFC.Abs
import BNFC.Lex
}
%name pListString_internal ListString
%monad { Err } { (>>=) } { return }
%tokentype {Token}
%token ',' { PT _ (TS _ 5) }
%%
ListString :: { (BNFC.Abs.BNFC'Position,
[String]) }
ListString
  : String { (fst $1, (:[]) (snd $1)) } |
String ',' ListString { (fst $1, (:) (snd
$1) (snd $3)) }
```

(b): An extract from the Happy parser specification used by BNFC

Grammar 4 : Two parser generator specifications

2.4 Labelled BNF

Context-free grammars can be defined in *Labelled BNF* (LBNF) [21], a format that extends the functionality of BNF. In its most elementary notion, LBNF is a means by which context-free grammars can be specified with syntax analogous to BNF, with the additional requirement that individual production rules must be named with a label. A simple LBNF specification for terms in the lambda calculus is demonstrated by Grammar 5, where `Var`, `Abs`, `App`, and `_` are labels for production rules of the non-terminal symbol `Term`. Furthermore, terminal symbols can be defined as strings of characters enclosed by double quotation marks: `"λ"`, `"."`, `"("`, and `")"`.

```
Var. Term ::= Ident              ;
Abs. Term ::= "λ" Ident "." Term ;
App. Term ::= Term Term          ;
_.   Term ::= "(" Term ")"       ;
```

Grammar 5 : An LBNF specification for terms in the lambda calculus

[5]The project is licensed under the MIT License.

2.4.1 Categories

The left-hand side of a production rule consists of a label and a non-terminal. Each non-terminal formally belongs to a *category*. A category may be constituted by a single non-terminal, e.g. `Term` in the example above. Multiple non-terminal symbols may also be logically grouped into a common category. That is the case in Grammar 6, where the non-terminals `Exp1`, `Exp2`, and `Exp3` are *indexed* variants of the proper category `Exp`. Indexed variants may be used to specify order of operations by equipping the production rules of each precedence level with a distinct index, as seen here, where the conventional order for arithmetic operations is specified. This is possible on account of the indexed variants being distinct non-terminal symbols in the grammar.

```
EInt. Exp3 ::= Integer       ;
EMul. Exp2 ::= Exp2 "*" Exp3 ;
EAdd. Exp1 ::= Exp1 "+" Exp2 ;
_.    Exp3 ::= "(" Exp ")"   ;
_.    Exp2 ::= Exp3          ;
_.    Exp1 ::= Exp2          ;
_.    Exp  ::= Exp1          ;
```

Grammar 6 : An LBNF specification defining addition and multiplication of integers with conventional order of operations

However, in order that a non-terminal symbol may refer to any of the constituent indexed variants of a category, or that a non-terminal of a lower index may be used with a production rule belonging to one of a higher index, additional production rules must be defined to accommodate such behaviour. This can be accomplished by employing *coercion* rules with the special rule label _ (underscore).

Sequences of non-terminal symbols, of arbitrary length, can be represented in LBNF as *list categories*. A non-terminal of a list category is denoted by square brackets, e.g. `[Exp]` for a sequence of the non-terminal `Exp`. Production rules for list categories can recursively be defined using the special rule labels `[]`, `(:[])`, and `(:)`, the first two of which denote the base cases, viz. empty and singleton sequences, and the third denotes the (right-associative) recursive case. In Grammar 7, the list category `[Exp]` is defined as a comma separated sequence, the elements of which are constituted by the non-terminal `Exp`. The names of these rules are derived from the equivalent operations on the list data structure in Haskell.

```
[].    [Exp] ::=               ;
(:[]). [Exp] ::= Exp           ;
(:).   [Exp] ::= Exp "," [Exp] ;
```

Grammar 7 : An LBNF specification for comma delimited sequences of expressions, of length zero or greater

LBNF provides other means to define grammars than through production rules. Amongst such facilities are special directives known as *pragmas*.

2.4.2 Pragmas

The `coercions` pragma is expediently employed when indexed variants of a category have been defined and coercions between them are desired. Rather than by explicit definition of coercion rules, this result may be obtained by means of this pragma. Thus, in place of the coercion rules employed in Grammar 6, the equivalent grammar can be obtained through the pragma invocation `coercions Exp 3`, in which `Exp` denotes the proper category and `3` the highest index level that is to be coerced.

For the purpose of facilitating the definition of sequences, as opposed to using the special rule labels for list categories described above, the `separator` and `terminator` pragmas may be used. For a category and a string of characters, the pragmas define sequences, the elements of which consist of the former and are separated by the latter. The `terminator` pragma requires the existence of a final delimiter, but the `separator` pragma does not. Thus, the definition of the list category `[Exp]` in Grammar 7 may equally be defined by the pragma invocation `separator Exp ","`.

Usage of the `separator` and `coercions` pragmas in addition to equivalent production rules, respectively demonstrated in Grammar 7 and Grammar 6, is illustrated by Figure 7.

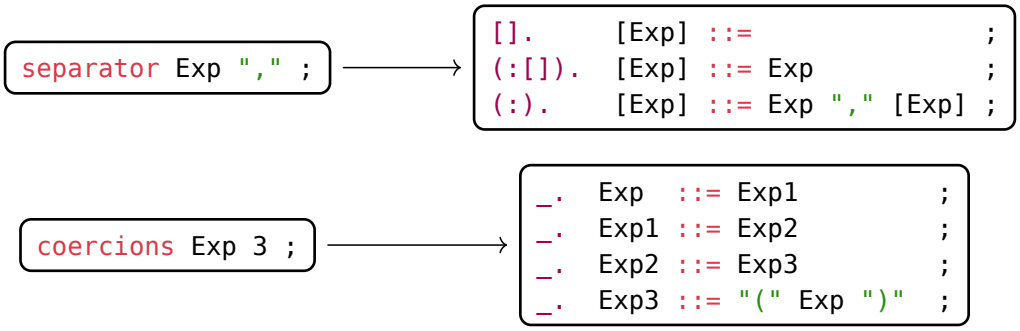


Figure 7 : Invocations of the `separator` and `coercions` pragmas with their respective equivalent production rules

2.4.3 Tokens

LBNF provides facilities for creating user-defined tokens by means of regular expressions. For the purpose of illustration, a token for hexadecimal integer literals, as those found in C and Haskell, e.g. `0xFF` for decimal `255`, can be defined as demonstrated in Grammar 8. Furthermore, LBNF provides a set of pre-defined tokens, namely `Integer`, `Double`, `Char`, `String`, and `Ident`. Each token, whether pre-defined or explicitly specified by a token definition, is regarded by LBNF as its own *token category*, which can not be re-defined, but may be used as any other category.

```
token HexInt ('0' 'x' (digit | ["abcdefABCDEF"])+) ;
```

Grammar 8 : A definition for a token of hexadecimal integer literals, e.g. literals of the form `0xff00a`

2.4.4 Functions

In addition to attaching a label to a rule, which starts with an uppercase letter, LBNF allows for invoking functions, which are defined in LBNF itself. Functions are distinguished from normal labels by starting with lowercase letters. By invoking a function rather than using a label for a rule, the function is applied before the resulting parse tree is produced. Rather than producing a node in a parse tree, a function thus manipulates already-existing parse trees. This allows for the addition of so-called *syntactic sugar*, that is, syntactical constructions whose semantics are entirely encompassed by those of other constructions.

Grammar 9 demonstrates the use of defined functions by those of `for` and `inc`. The function `for` is invoked whenever a C-style for-loop is encountered, and transforms (*desugars*) the loop into a while-loop, which, by contrast, has its own definition in the grammar. Furthermore, the function `inc` is invoked whenever a C-style increment is encountered, e.g. `i++` which is expanded to the statement `i = i + 1`.

```
Assign.     Stm ::= Ident "=" Exp ";"                       ;
Block.      Stm ::= "{" [Stm] "}"                           ;
While.      Stm ::= "while" "(" Exp ")" Stm                 ;
terminator Stm ";"                                          ;

for.        Stm ::= "for" "(" Stm ";" Exp ";" Stm ")" Stm   ;
inc.        Stm ::= Ident "++" ";"                          ;

define for i c s b = Block [i, While c (Block [b, s])]     ;
define inc x       = Assign x (EOp (EVar x) Plus (EInt 1)) ;
```

Grammar 9 : An LBNF specification in which for-loops and an increment operator are specified using function definitions, adapted from Ranta [22, p. 186f]

2.5 BNF Converter

The *BNF Converter* (BNFC) is a compiler construction tool developed at Chalmers University of Technology [23]. BNFC acts upon a user-specified formal language specification written in LBNF and generates a corresponding lexer and parser, along with an abstract syntax tree, by which the output of the generated parser is constituted. The output files generated by BNFC may target any of the currently six supported programming languages. This is accomplished by converting the common input specification, viz. LBNF, into specifications that are distinct to certain parser and lexer generators, whereto the files are then passed. Moreover, back-ends generate a *linearizer* in the target language, which converts an abstract syntax tree to a human-readable text format, resembling the input that was used to generate the AST. A visualization of this process is shown in Figure 8.

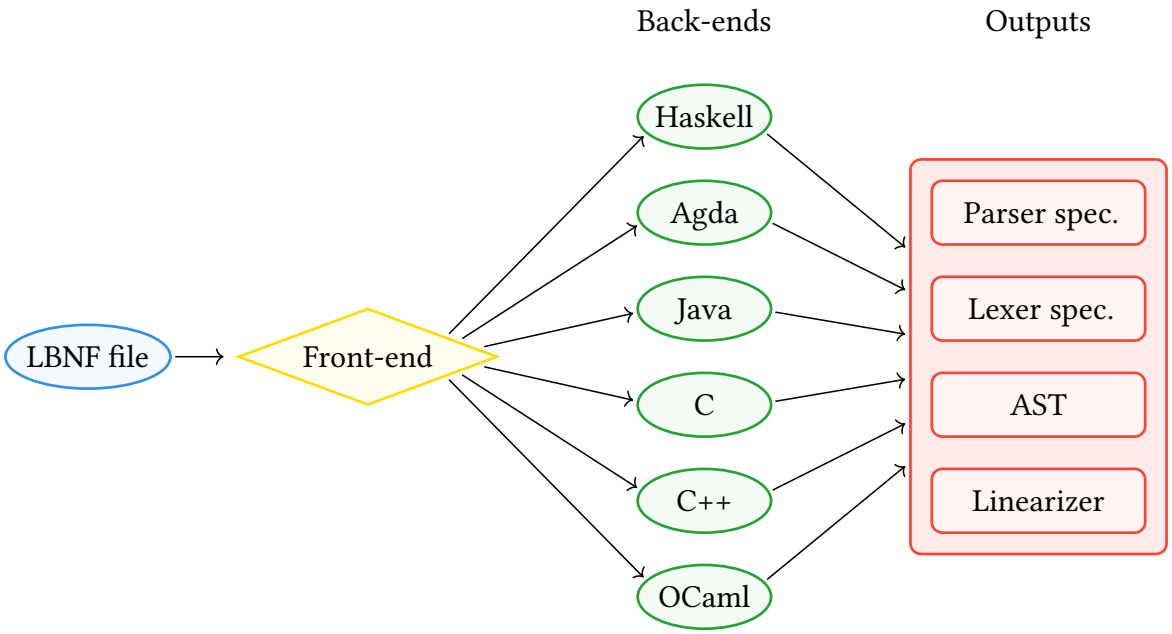


Figure 8 : The structure of the BNF Converter

2.5.1 Program structure

The BNFC codebase is almost entirely written in Haskell and can be logically divided into a *front-end* and several *back-end* components. The responsibility of the former is that of reading the contents of the input LBNF file, lexing the character sequence by which it is constituted, parsing the sequence of tokens produced by the lexer, and, finally, converting the abstract syntactic representation produced by the parser into a more efficient internal representation as well as performing various semantic analyses upon it. The internal representation is then passed to one of several back-ends representing the various supported target languages and generators, the choice of which being under the discretion of the user. A lexer specification and a parser specification, along with other files, are then to be produced by the designated back-end.

As for BNFC version 2.9.5, there are six back-ends available, one of which is to be selected by the user by means of a command-line flag. Examples of usage of such command-line flags for the purpose of selecting a back-end is demonstrated in Listing 1.

```
$ bnfc --haskell grammar.cf
$ bnfc --agda    grammar.cf
$ bnfc --cpp     grammar.cf
```

Listing 1 : Invocations of various back-ends of BNFC version 2.9.5 using its command-line interface

2.5.2 Previous interface

The back-end interface presented in Vergani 2021 [24] provided a function declaration for each of the components by which a back-end was expected to be constituted, as shown in Listing 2. Each such function is expected to return a list of files and their contents, which is denoted by the `Result` type alias. The decision to construct the interface in this manner seems to have been taken under the assumption that the back-ends were equal in nature and constituted by the same components. For instance, there may have been a perception that each back-end consisted of distinct and independent components, the responsibility of which being the generation of files relating to the lexer, the parser, the abstract syntax tree, et cetera. This interface is thus largely a formalization of what was perceived to be the logical structure of the BNFC version 2.9.5 codebase, and is now a part of an unreleased version of BNFC.

```
type Result = [(FilePath, String)]

class Backend (target :: TargetLanguage) where
  type BackendOptions target
  type BackendState target
  parseOpts      :: Parser (BackendOptions target)
  initState      :: LBNF -> GlobalOptions -> BackendOptions target
                         -> Except String (BackendState target)
  abstractSyntax :: LBNF -> State (BackendState target) Result
  printer        :: LBNF -> State (BackendState target) Result
  lexer          :: LBNF -> State (BackendState target) Result
  parser         :: LBNF -> State (BackendState target) Result
  parserTest     :: LBNF -> State (BackendState target) Result
  makefile       :: LBNF -> State (BackendState target) Result
```

Listing 2 : The BNFC back-end interface as presented in Vergani 2021

2.5.3 Recent developments

BNFC has undergone development since its inception. The multilingual nature of the tool renders it appropriate for use as a sandbox for exploring and developing programming language concepts related to parsing, as well as for creating prototypes of compiler front-ends.

For instance, an old and somewhat crude implementation of support for indent-sensitive languages was found unsatisfactory, whereupon a new layout syntax system was developed by Burreau [25]. The solution found was as general as BNFC; the resulting features may be used in any target language, providing that a *layout resolver* is added to the back-end.

3

The Interface

In the construction of an interface purposed to unite a diverse range of components within one common framework, it may be necessary to reconcile two opposing considerations. On the one hand, in order that the idiosyncrasies and varying requirements are accommodated – both of existing components and of those yet to be introduced – the interface must provide a level of abstraction sufficiently versatile. On the other hand, the interface should maintain a degree of precision, so as to ensure consistency and uniformity across all components in the system.

These considerations are relevant in the design of interfaces within many software architectures. This chapter shall seek to create an interface for a software project constituted by a single front-end component and multiple back-end components, namely one for the BNF Converter (cf. Section 2.5.1 for a more comprehensive description of this architecture). This interface constitute a crucial part of the broader back-end compatibility check, described in Section 1.2.

The considerations outlined above can be further articulated in a more concrete manner with respect to BNFC and the back-end compatibility check that has been implemented within the scope of this thesis. The interface should be sufficiently general in order that it may accommodate the many aspects and peculiarities that characterize the programming languages and generators that the various back-ends of BNFC target. In addition, it should expedite future additions to BNFC, whether it be new capabilities of LBNF or a new back-end, and allow for a uniform means of communication between the front-end and all the back-ends. A concise and strict interface may facilitate such uniformity. Thus, in order to determine which aspects of programming languages may be of interest and in which way these differences may affect the design of the interface, several languages have been analysed.

For this analysis, an array of programming languages has been selected. These languages and the bases for their inclusion are presented in what follows.

- **Haskell** and **OCaml** were chosen on grounds of being functional programming languages, thus contrasting imperative languages in certain aspects. Furthermore, there are existing back-ends for these languages, rendering them appropriate for this study.
- **C** and **C++** are ubiquitous system programming languages that are relatively old. The tooling commonly employed in conjunction with these languages might have been different in nature and capability to those of more recent languages.

- **JavaScript** and **Python** are widely used interpreted scripting languages. A back-end for Python has been developed [26], but none exists for JavaScript.
- **Java** was selected on account of it being a widely used object-oriented programming language, the peculiarities of which paradigm might have been of interest for this analysis. There is also a BNFC back-end for Java.
- **Rust** was included on the grounds that a new back-end was to be developed for this language within the scope of this thesis. Thus, any peculiarities of Rust were of particular interest. Further justification for making a back-end for Rust is provided in Chapter 4.

The interface presented in Vergani 2021, which is described in greater detail in Section 2.5.2, merely provided means by which the different back-ends could be invoked in a unified fashion by the front-end. Thus, the front-end could perform no assessment on the validity of the internal representation with regard to its compatibility with a specific back-end, besides the requirements stipulated by the LBNF formalism. As a result of this, each back-end, if applied to input with which it is, for any reason, not compatible, would result in the BNFC process crashing with communication of all error messages under the discretion of the specific back-end in question. As for the user experience, it should be stated that a system behaving thus lends to confusion on account of the great diversity of error messages that it may convey. Furthermore, should all communication, or a great part of it, be conveyed by the front-end, a unified and more regular system of communication to the user could be established. Therefore, it has been found that an interface through which all communication to the user is conveyed is preferred to the alternative.

The interface should provide means by which the back-ends are able to specify their capabilities to the front-end, so as to enable the front-end to verify whether a given LBNF specification is compatible with a specified back-end. In order to accomplish this, the interface presented by Vergani is to be revised, insofar as to provide means by which the aforementioned properties can be realized.

For this reason, a third stage was added to the LBNF validation pipeline, to allow for checks to be performed on the LBNF specification after the existing checks, but before the back-end is invoked. This stage, which is named Stage 3 in Figure 9, contains many of the checks that is part of the back-end compatibility check. Part of the check has also been implemented in Stage 2. Details about what Stage 3 contains, and what has been added to Stage 2, is presented in this chapter.

Furthermore, the `Backend` type class, of which each back-end is an instance, was amended in order that the back-ends be able to provide the information required for the front-end to perform the necessary checks. The resulting type class can be seen in its entirety in Listing 3. Details about what the semantics of that information is, and when it is used, is also presented in this chapter.

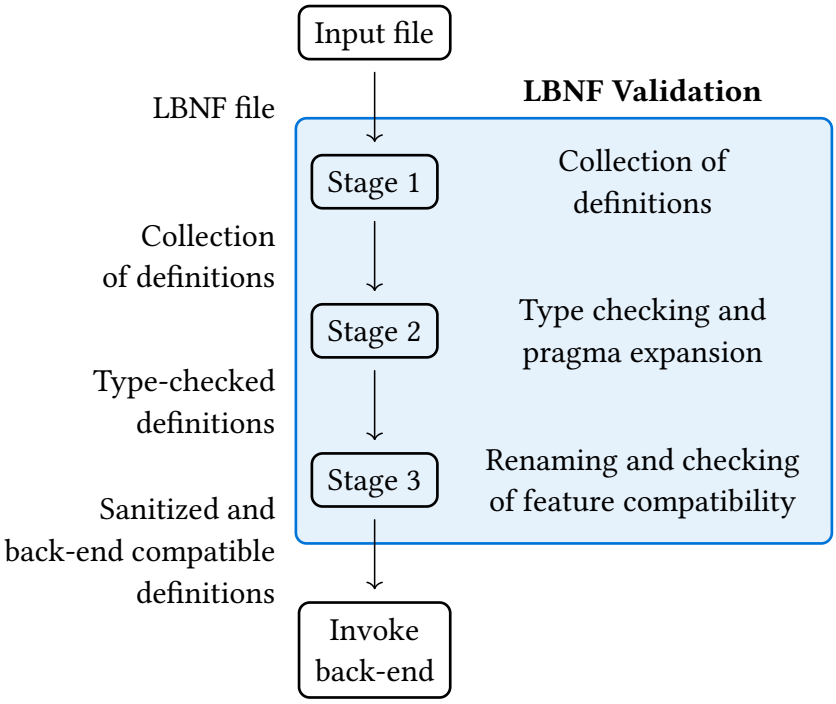


Figure 9 : The three stages of the validation pipeline, as of the new version of BNFC

3.1 General properties of the interface

A back-end in the BNFC source code is a Haskell type that is an instance of the `Backend` type class, as seen in Listing 3; there is one type, as well as an instance of `Backend`, for each available back-end, e.g. `instance Backend Java` and `instance Backend Haskell`.

3.1.1 Manifest

By way of the interface, each back-end provides a `Manifest`, the definition of which can be seen in Listing 3. Through it, the name of the back-end, the *command-line interface* (CLI) command by which it is to be invoked, a description of the back-end and its capabilities, and the kind of files that it can generate, such as lexer and parser specifications or documentation, is declared by each back-end.

By means of the interface and specifically the manifest, each back-end is itself in control of how it is to be invoked by the user. This is in stark contrast to the manner by which BNFC is invoked in previous versions, both in the version 2.9.5 and the unreleased version; the code declaring the name of the commands in previous iterations unequivocally reside within the front-end codebase.

```haskell
class Backend b where
  -- | The type containing the parsable command-line option
  -- for this back-end.
  type BackendOptions b
  -- | The type containing the state for this back-end.
  type BackendState b
  -- | A 'Manifest' describing the name, description, invocation,
  -- and type of the back-end.
  manifest :: Manifest
  -- | A 'Rule' to which labels adhere.
  labelRule :: Rule
  -- | A 'Rule' to which categories (non-terminals) adhere.
  categoryRule :: Rule
  -- | A 'Rule' to which (function) definitions adhere.
  defineRule :: Rule
  -- | A 'Rule' to which function arguments adhere.
  argRule :: Rule
  -- | A 'Options.Applicative.Parser' that parses the back-end specific
  -- command-line options (@BackendOptions b@).
  parseOpts :: Parser (BackendOptions b)
  -- | An initializer for the state of this back-end.
  initState :: BackendEnv b -> Except Text (BackendState b)
  -- | A 'BackendExec' action that computes the output files.
  runBackend :: BackendExec b FileOutputs
  -- | A set of features that this back-end supports. Specified by the
  -- 'Features' type.
  features :: Features b
  -- | The default type of recursion for the @separator@ and @terminator@
  -- pragmas, which is either to left- or right-recursive. This only applies
  -- when neither the @left@ nor the @right@ qualifier is specified.
  defaultListRecursion :: Recursion

-- | The type of recursion of a list definition.
data Recursion = LeftRecursion | RightRecursion

-- | A manifest describing an instance of the
-- 'Backend' type class.
data Manifest = Manifest
  { -- | The name of the back-end.
  , name :: Text
  -- | The name of the command that is to be
  -- invoked in order to run the back-end.
  , commandName :: Text
  -- | A description of the back-end.
  , description :: Text
  -- | The 'BackendKind's of the back-end.
  , kind :: [BackendKind]
  }

-- | The kind of functionality provided by a 'Backend'.
data BackendKind = Parser | Lexer | Documentation | Other
```

Listing 3 : The back-end interface represented by the Haskell type class `Backend`

3.1.2 Options

When a specified back-end is invoked using the command-line interface, several options are available. Some of them are *common* options, that is those options that are available regardless of which back-end is being invoked. An example of this is the `--dry-run` option, which instructs BNFC to perform all operations in the normal fashion, with the exception that no files on the disk are to be modified. Other command-line options are specific to a distinct back-end. For the sake of example, the `--jflex` option for the Java back-end instructs BNFC to produce a lexer specification for the JFlex lexer generator instead of that for JLex, which is the default.

In version 2.9.5 of BNFC, all command-line options are parsed and collected into a structure common to all options, whether common or back-end specific, in the same component of code. This structure of options thus contains all back-end specific options, even though any given back-end can only make use of the common options and those particular to itself in any meaningful way; options pertaining to one back-end bear no significance in the context of another. Furthermore, it necessitates that all back-end specific options be declared at the same location within the codebase. Although this manner of declaring and passing disjoint sets of input parameters to separate and independent components is by no means ideal, it is nevertheless not unrepresented in codebases of other projects with similar structure. A specimen of a project of this nature is the document converter Pandoc [27], the source code of which exhibits a division between a front-end and various back-end components, similar to that of BNFC.

This deficiency was addressed in the interface proposed by Vergani [24]. It was devised so that each back-end may present to the front-end a structure containing its specific command-line options and a means by which this structure can be parsed, respectively represented by the type synonym `BackendOptions` and the declaration `parseOpts`, the latter of which were to utilize the command-line parsing utilities provided by the `optparse-applicative` library [28]. A definition of back-end specific options and their accompanying parser for the Haskell back-end, as part of this interface, is demonstrated in Listing 4.

Consequently, the interface proposed by Vergani allows for the parameters, on which a specific back-end is to be applied, and the means by which these parameters are obtained to be defined in the same source code component as the rest of the back-end. Additionally, it enables the front-end to exclusively pass the options particular to the back-end that is being invoked. As for the communication with the user, the front-end can display a list of available command-line options to the user, which only includes those options that are relevant for any given back-end. In conclusion, this part of the interface provides the necessary means through which the back-end source code can be decoupled from that of the front-end, and more efficient and precise information can be displayed to the user of the software. For those reasons, this part of the interface was kept in the interface developed as part of this thesis.

```haskell
data HaskellOptions = HOpts
    { inDir   :: Bool
    , functor :: Bool
    , generic :: Bool
    , xml     :: Bool
    , gadt    :: Bool
    }
```

```haskell
instance Backend Haskell where
  BackendOptions Haskell = HaskellOptions
  parseOpts = HOpts
    <$> oInDir
    <*> oFunctor
    <*> oGeneric
    <*> oXml
    <*> oGadt
    where
      oInDir = switch (
        short 'd' <>
          help "Put Haskell code in modules"
        )
      oFunctor = switch ( ... )
```

Listing 4 : Definition of command-line options specific to the Haskell back-end and the associated parser

3.1.3 Monadic computations

The concept of *monads* has its origins in category theory, but they have proven to be of great benefit in functional programming. Monads allow pure functional code to express computations involving environments and states and may be used to encapsulate impure effects in a pure manner [29].

In Haskell, the type class `Monad` provides an interface for monadic computations that is utilized by various types in the standard library, as well as third party libraries. Amongst the latter are the *reader* and *state* monads [30], which allow for computations to, as a side effect, read an enclosing environment and to manipulate a state, respectively. Furthermore, the monadic interface provides convenient means by which such computations can be composed and by which environments and states may be implicitly passed between computations.

Monads can be further composed by way of *monad transformers*, which allow for multiple side effects within the same environment. For instance, using a `ReaderT` monad transformer in conjunction with a `State` monad, yields an environment in which some value can be read using the outer monad, the `Reader`, whilst also enabling the manipulation of state using the inner monad, `State`. Examples demonstrating how monad transformers may be defined in Haskell are shown in Listing 5.

```haskell
newtype Reader  r   a = Reader  (r ->   a)
newtype ReaderT r m a = ReaderT (r -> m a)

newtype State   s   a = State   (s ->   (a, s))
newtype StateT  s m a = StateT  (s -> m (a, s))
```

Listing 5 : Definitions of reader and state monads with the corresponding monad transformers in Haskell

The main functionality of a BNFC back-end is encapsulated by the interface declaration `runBackend :: BackendExec b FileOutputs` (cf. Listing 3), in which the type parameter `b` specifies the back-end and `FileOutputs` represents relative paths and contents of files that are to be written to disk. The `BackendExec` type constructor, when applied to an instance `b` of the `Backend` type class, is a monad in which the computations of the back-end are to take place. It is composed by a reader monad transformer on top of a state monad, as seen in Listing 6.

```
type BackendExec b a = ReaderT (BackendEnv b) (State (BackendState b)) a
```

Listing 6 : The `BackendExec b` monad

As described earlier, this is a composition of two monads, the `Reader` monad and the `State` monad, which allows for two effects. The `Reader` monad allows for reading the `BackendEnv b` environment containing the intermediate representation of the LBNF specification, as well as common and back-end specific options parsed from the command-line. See Listing 7 for a definition of `BackendEnv b`. The state monad contained within allows the back-end to read and store some state during its execution. The contents of this state are fully controlled by the back-end in question; the type representing the state of a back-end must be provided, for each back-end, through the `BackendState b` associated type alias, as well as an initialization function `initState`, both part of the interface.

```
data BackendEnv b = BackendEnv
  { lbnf           :: LBNF
  , compileOptions :: CompileOptions
  , backendOptions :: BackendOptions b
  }
```

Listing 7 : The `BackendEnv` data type

3.2 Increasing the encapsulation of BNFC

The part of the codebase pertaining to the front-end component of version 2.9.5 of the BNF Converter, the latest released version as of writing, exhibits a high degree of coupling with those of the various back-ends. As has been illustrated previously, the code that relate to declaration and parsing of command-line options, even those that are back-end specific, is contained within the front-end portion of the codebase. Furthermore, discrepancies in capabilities between back-ends is handled ad hoc in the front-end code on a case-by-case basis. For instance, some back-ends are not capable of generating output for LBNF specifications that contain a rule label with the same name as that of a category, although some back-ends may support such specifications. In order that it be possible to determine whether a given back-end supports *duplicate names* in the front-end, a conditional branching on the selected back-end is performed, as demonstrated in Listing 17. This is described in greater detail in Section 3.6.

The work of Vergani [24] made all back-ends instances of the `Backend` type class, which contained functions common to all back-ends at that time. Most of the communication

between the front-end and the back-ends happened through that common API, which lowered the coupling compared to the previous version. However, the resulting product still had the front-end and back-ends too highly coupled – there were parts of the BNFC front-end that directly depended on parts of the back-ends that were not exposed by the interface, and code residing in the back-end subdirectory that had little to do with back-ends, hinting at too little separation. See Listing 8 for some exhibits.

```
-- Code in the executable directly
-- dependent on the back-ends
commandsParser :: Parser Command
commandsParser = hsubparser
  (  command "c" (info (C <$>
cOptionsParser) (progDesc "Output C
code"))
  <> command "cpp" (info (Cpp <$>
cppOptionsParser) (progDesc "Output C++
code"))
  <> command "haskell" (info (Haskell <$>
haskellOptionsParser)
    (progDesc "Output Haskell code for use
with Alex and Happy"))
  <> ...
```

```
-- Code living in the back-end which is
-- only used in the CLI
module BNFC.Backend.CommonInterface.Write
(printFiles, writeFiles) where

import BNFC.Prelude
import
BNFC.Backend.CommonInterface.Backend

import System.IO        (pattern ReadMode,
hClose, hGetContents, openFile)

printFiles :: FilePath -> Result -> IO ()
printFiles root = mapM_ (printFile root)
...
```

Listing 8 : Samples of source code from the unreleased version of BNFC that indicate high degrees of coupling

For the purpose of decreasing the coupling between software components and discouraging future coupling, various methods of source code encapsulation exist. In many programming languages, *modules*, whether explicitly declared or implicitly created, e.g. at source file boundaries, are available as a means for source code encapsulation and are typically provided as a language construct. Whilst a large codebase can be, and often is, divided into separate modules, on account of the fact that modules typically only comprise single files, higher-level components may encompass several modules, each of which may depend on arbitrary modules within the entire codebase, regardless of whether they are part of the same logical component. Thus, modules do not provide sufficient means by which boundaries between components can be enforced or clearly communicated. Instead, other means for achieving these objectives can be employed. *Packages*, which are generally not part of language specifications, but rather provided by a package manager or build system, are both a means for source code distribution as well as encapsulation. Inter-package dependencies are typically only possible through a fixed and well-defined set of code units. Ad hoc dependencies between packages are generally not possible; a package may not use code units that are intended to be private to another package. Thus, through the division of a codebase into separate packages, clearer boundaries for its various components can be established and enforced.

In terms of the BNFC codebase, which is written in Haskell, the build system *Cabal* may be leveraged as a means to decrease the coupling between the front-end and the back-end components, as well as to discourage future coupling, should the tool be developed further. Cabal imposes the restriction that packages may not be dependent on each other in a way that cyclic dependencies are created, which can be leveraged to force a weaker coupling between

the front-end and back-ends. Furthermore, which parts of a Cabal package that are exposed to other packages that import it, can more easily be controlled.

As a part of this project, the BNFC codebase has been split into three separate Cabal packages: `bnfc3-lib`, `bnfc3-backends`, and `bnfc3-executable`. A dependency graph of these packages can be seen in Figure 10. Structuring the BNFC codebase in this manner provides greater separation of concerns compared to previous versions, in part by virtue of the limitations provided by the package boundaries. The three packages are described in more details in the following subsections.

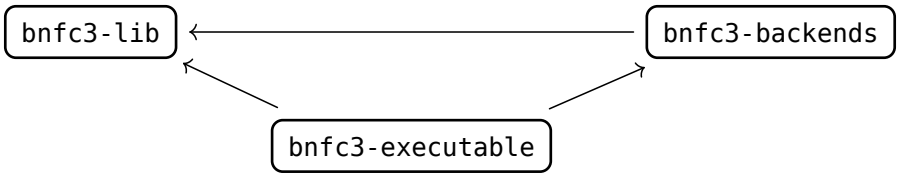


Figure 10 : A dependency graph of the Cabal packages that constitute the new version of BNFC, in which the outbound node is dependent on the inbound, for any given vertex

3.2.1 The `bnfc3-lib` package

The `bnfc3-lib` package contains all source code that does not depend on any specific back-end and that is not related to the execution of the BNFC command-line program. On account of the restrictions imposed by the package boundaries and their respective dependencies, demonstrated in Figure 10, the `bnfc3-lib` package must by necessity include the entire API by which the front-end and back-ends communicate. It also includes the internal representation of LBNF that is provided as the input format to the back-ends.

Furthermore, the source code relating to parsing and validation of LBNF, including all three stages described by Figure 9, is part of the `bnfc3-lib` package. This fact enables a third party tool to employ `bnfc3-lib` for the purpose of parsing LBNF. For instance, the package could be used to develop an LBNF language server implementing the Language Server Protocol (LSP) [31].

Since the various back-end components are not declared in `bnfc3-lib`, but rather in the `bnfc3-backends` package, it is inconvenient from within the `bnfc3-lib` package to figure out what specific back-end is in use when the LBNF checking takes place. This means that developers are encouraged to not make hard-coded exceptions for certain back-ends within the LBNF validation phases, but to develop general solutions that will work well both for current and future back-ends. The entirety of `bnfc3-lib` is written without the knowledge, in any part of the execution pipeline, of what back-end has been selected by the user.

3.2.2 The `bnfc3-backends` package

The back-end components are contained within the `bnfc3-backends` package. It only exposes one data type, `Backends`, each constructor of which represents one of the back-ends, which is shown in Listing 9. Each of these constructors are, by means of the `DataKinds` language extension, promoted to types [32] – whilst the type `Backends` is promoted to a *kind*[6] – and these types are instances of the `Backend` type class.

```
data Backends
  = Agda  | C
  | Cpp   | Haskell
  | Java  | Latex
  | OCaml | Rust
  | Txt2Tags

instance Backend Agda where
  ...
instance Backend C where
  ...
```

Listing 9 : The `Backends` data type, containing `Backend` instances

Internally, the `bnfc3-backends` contains all the logic for executing each back-end along with a parser for the command-line options, a manifest, and all other definitions of the `Backend` type class (cf. Listing 3). The package also facilitates sharing code between different back-ends, such as certain utilities and alternate representations of the LBNF structure. However, since only the `Backends` data type and the instances of the `Backend` type class, corresponding to its constructors, are exposed, all code dependent on the `bnfc3-backends` package is restricted to using the API defined by the `Backend` type class for all tasks. This eliminates the possibility of depending on back-end specific code not exposed by this interface.

3.2.3 The `bnfc3-executable` package

The `bnfc3-executable` package contains the command-line interface to BNFC, which is the main way a user is intended to interact with the program. The package provides the entry point through which the command-line application is run. By means of command-line options, the application can be instructed to generate the appropriate files for a specified back-end and LBNF input. If the LBNF input passes both the common validation and the back-end specific compatibility check, the expected output files are generated by the designated back-end. Should an error occur, it is instead displayed to the user.

Furthermore, the `bnfc3-executable` facilitates end-to-end testing, that is, testing by providing an input LBNF file and comparing the output to an expected output.

[6] In type theory, a kind denote the 'type' of a type constructor (data type). E.g. the unary type constructor `Maybe` is of kind `Type -> Type`, whilst the kind of `Int` is just `Type`.

3.3 Reserved keywords

To express constructions in programming languages, *keywords* are often employed. Keywords are, most commonly, strings of alphabetic characters, which bear special meaning in the language and cannot be used as identifiers. Whilst it might, for instance, be convenient for a developer to use the identifier `if` for a variable, it is, however, not permitted to do so in any of the studied programming languages, on account of `if` being defined as keyword in all of them.

Table 1 shows some of the keywords in the examined languages, together with how they are categorized in their respective language.

Table 1 : Reserved keywords in some languages

Language Specification	Reserved keywords
Haskell 2010 [33]	23 reserved identifiers: `case`, `class`, `module`, `_`, 11 reserved operators: `..`, `:`, `::`, `=`, `\` Additional reserved identifiers can be added by enabling certain language extensions
OCaml 5.3 [34]	56 reserved keywords: `downto`, `lor`, `rec`, `false`, `functor`, …
Java 21 [35]	51 reserved keywords: `abstract`, `assert`, `native`, `goto`, `instanceof` 17 contextual keywords: `exports`, `opens`, `non-sealed`, `yield`
C23[7] [7]	59 keywords[8]: `do`, `const`, `int`, `__Alignas`, …
ECMAScript 2024 [36]	38 reserved words: `await`, `break`, `catch`, `debugger`, `void`, `yield`, …[9] 8 reserved words in strict mode: `let`, `static`, `package`, `protected`, …
Python 3.13.2 [37][10]	35 keywords: `True`, `False`, `None`, `nonlocal`, `elif`, `del`, `lambda`
Rust 2024[11]	38 strict keywords: `crate`, `false`, `impl`, `loop`, `self`, `Self`, `unsafe`, … 14 reserved keywords: `abstract`, `become`, `final`, `unsized`, `gen`, … 5 weak keywords: `macro_rules`, `union`, `'static`, `safe`, `raw`

Generally, reserved keywords only consist of lower-case letters, as is the case for ECMAScript. However, in some languages, keywords may begin with a capital letter. For instance, `Self` and `True` are reserved keywords in Rust and Python, respectively. Furthermore, reserved keywords may include non-alphabetic characters, e.g. an underscore, which is the case for Haskell, C, and Rust.

In Haskell, where all reserved keywords are lower-case, it is a common practice to derive a new name from an already existing one by means of appending an apostrophe.

[7] For the purpose of counting the number of keywords, the latest working draft of the standard has been consulted.

[8] This number includes five alternative spellings of the 54 distinct keywords.

[9] `await` and `yield` may contextually be used as identifiers

[10] Due to the lack of an official specification, 'The Python Language Reference' has been consulted.

[11] Due to the lack of an official specification, 'The Rust Reference' has been consulted.

In Python and JavaScript, there is no generally accepted solution to this problem. In Java, it is common to use the identifier `clazz` in place of the keyword `class`. Both the contextual keywords of Java and the weak keywords of Rust behave in the same way; they are only reserved in certain positions. For instance, in Java, a variable may be named `exports`, which is also a contextual keyword.

In Rust, another method can be employed; an identifier may be formed by prepending the special escape sequence `r#` to an otherwise reserved name. Thus, it is possible to name an identifier `r#impl`, whilst it is not possible to do so for `impl`, a reserved keyword.

3.3.1 Current behaviour

In BNFC version 2.9.5, as well as in the unreleased version, no input sanitization, with respect to names of labels, categories, functions, and function arguments, which are henceforth collectively referred to as *names*, is performed during the course of parsing and validation of the LBNF input. Consequently, this task is delegated to the various back-ends.

The Haskell back-end, for instance, appends an apostrophe whenever a reserved keyword is used in the position of a category, rule label, or function definition, as can be seen in Listing 10. In function argument positions, an underscore is appended instead. In the Java back-end, an underscore is appended in every case a reserved keyword is used.

```
avoidReservedWords1 x =
  if toList x `elem` hsReservedWords
    then x <> "'"
    else x
```

Listing 10 : Sanitization of reserved keywords in the Haskell back-end

The LBNF structure is large and has the same names spread over multiple internal structures. It is thus very possible for a back-end developer to, by mistake, only apply sanitization in some places but not in all. Additionally, there is no way for the back-end to communicate to the end-user that a renaming has occurred.

Furthermore, in the unreleased version of BNFC, the only way a back-end may fail gracefully is when initializing its state, which is one of the many stages it will execute. This is also a stage that may not produce any file content as its output. If it fails to properly reject any forbidden names, the only remaining options are to yield invalid output, or fail catastrophically during a future stage of execution.

3.3.2 Proposed solutions

It would be convenient for the front-end of BNFC to facilitate a common way to apply the syntax-imposed keyword sanitization of a back-end. To successfully solve the problems described above, it should at minimum do the following:

- Allow for a back-end to disallow certain names.

- Provide a reasonable error message if a disallowed name is encountered.
- Allow for a back-end to change certain names according to some format.
- Provide messages informing the user what names have been changed.

Furthermore, since names may be represented differently in the different languages, it may be possible that some names are disallowed in one position whilst allowed in another.

One possible solution would be for the back-ends to each expose four arbitrary functions that would be used for checking label names, function names, function arguments, and category names respectively. The function would take the name as a parameter, and either allow the name, reject it, or change it. This could be represented in a couple of different ways, some of which are shown in Listing 11.

```
-- One signature for the filter function, that would take the old name and
-- possibly modify it, returning Nothing if it is rejected
type FilterFunction = Name -> Maybe Name

-- Another possible signature for the filter function
type FilterFunction = Name -> FilterResult
data FilterResult = Unchanged | Changed Name | Rejected
```

Listing 11 : Examples of name transformation functions

A problem of this solution would be that there is no way to inspect a function, to see what it does. Another problem is that if the back-ends may create the functions on their own accord, they may create inefficient functions. Furthermore, there is no general way to efficiently combine two functions.

Another possible solution is to use an *embedded domain-specific language* (EDSL) with a *deep embedding*. This could be done by having a data type where the constructors are the different checks that the back-ends may perform on the names. Since the number of finite hand-made constructors is smaller than the indefinitely many functions of the types shown above (Listing 11), this solution is less expressive. However, with some building blocks, advanced behaviours may emerge. The anticipated operations a back-end is expected to do to reject, accept, or change a name is easy to cover with an EDSL.

A deeply-embedded EDSL was developed for accepting, rejecting, and changing names. A subset of this EDSL, henceforth known as the *Rules DSL*, can be seen in Listing 12.

The rules are supplied to the front-end by the back-end by four new class values, which are added to the `Backend` type class. The extension of this class can be seen in Listing 13. Furthermore, these four rules default to a rule accepting all names, if not explicitly defined. This aims at making it easier in the initial stages of back-end development.

```
-- | The result of applying a rule on a name
data Decision = Unchanged | Rejected | Changed Name

-- | Applying a rule on a name
decide :: Rule -> Name -> Decision

-- | A rename rule that either accepts, rejects, or changes the name of a
-- function, label, or category.
data Rule
  = -- | Modifies everything matching the pattern according to the given
transformation
    Modifying Pattern Transform
  | -- | Rejects everything matching the given pattern
    Rejecting Pattern
  | -- | Performs the first rule, followed by the second rule. If any of those
    -- rejects the name, the name is rejected. If the first rule modifies the
    -- name, the modified name is passed to the second rule
    Sequence Rule Rule
  | -- | Performs the first rule, and if it rejects the name, performs the
    -- second rule
    Alternative Rule Rule
  | -- | Accept all input as-is
    Accept

-- | A pattern that either matches a @Name@, or does not.
-- The patterns are designed to be efficient at matching strings
data Pattern
  = -- | This pattern matches if the tested name is in the set
    PSet (Set Name)
  | -- | This pattern always matches
    Always
  | -- | This pattern matches if the name starts with the given prefix.
    Prefix Name
  | -- | This pattern matches if the name ends with the given suffix.
    Suffix Name
  | -- | This pattern matches if the name does not match the inner pattern
    Inverse Pattern
  | -- | This pattern matches if the name matches either the first or the second
pattern
    Union Pattern Pattern
  | -- | This pattern matches if the name matches both the first and the second
pattern
    Intersection Pattern Pattern

-- | Transformation of labels, categories, or definitions.
data Transform
  = -- | Prepends some text to the name
    Prepending Text
  | -- | Appends some text to the name
    Appending Text
  | -- | Replaces one substring with another
    Replacing Text Text
  | -- | Does nothing
    Nop
```

Listing 12 : A subset of the Rules DSL

```
class Backend b where
  ...
  -- | The 'Rule' that labels have to adhere to
  labelRule :: Rule
  -- | The 'Rule' that categories have to adhere to
  categoryRule :: Rule
  -- | The 'Rule' that function names have to adhere to
  defineRule :: Rule
  -- | The 'Rule' that function arguments have to adhere to
  argRule :: Rule
```

Listing 13 : The extension of the Rules DSL to the `Backend` type class

The current behaviour of the sanitization that the Haskell back-end does, which is shown in Listing 10, can be expressed in the Haskell `Backend` instance, as seen in Listing 14. If the Python back-end wants to reject all input containing reserved keywords, it may use the rule `Rejecting (PSet (fromList ["True", "False", ...]))`.

```
instance Backend Haskell where
  ...
  labelRule = Modifying (PSet (fromList hsReservedWords)) (Appending "'")
  categoryRule = Modifying (PSet (fromList hsReservedWords)) (Appending "'")
  defineRule = Modifying (PSet (fromList hsReservedWords)) (Appending "'")
  argRule = Modifying (PSet (fromList hsReservedWords)) (Appending "_")
```

Listing 14 : Using the `Rules` interface to escape reserved keywords in Haskell

The checking of names occur in the front-end after the LBNF file has successfully been parsed and type-checked. Thus, this becomes a post-processing step on the LBNF file. If any name gets rejected, the front-end rejects the whole file. If any name gets changed into another name, a message is shown to the user that a renaming has occurred. Thus, the back-ends themselves may depend on the guarantee that all names obey the specified rules, whilst not having to care to apply them at every usage site themselves.

When using a deeply-embedded EDSL, the structure is preserved at every step. Figure 11 shows how the structure of the rules given in Listing 14 are represented in memory. One can walk the structure to achieve various results; to interpret the rules and patterns, to create a textual representation of the rules, or to transform it. This is inherently different from functions, the operations of which are opaque within Haskell.

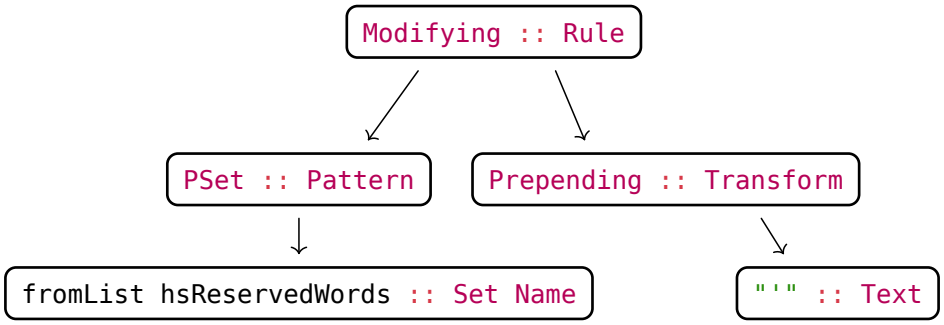


Figure 11 : Representation of the Haskell rules of Listing 14

Deeply-embedded EDSLs play well with optimizations too; since the constructors are preserved, it is possible to inspect any value and do transformations on them. This is a well-known advantage of using such deeply-embedded EDSLs [38]. For this specific instance, many laws may be formalized for how the constructors relate to each other. Many of these laws arise from the Rules DSL being a Boolean algebra. Some of these laws for patterns are shown in Table 2.

Table 2 : Algebraic laws for `Patterns`, where x, y are `Sets` of `Names`, and p, p' are `Patterns`.

Law	Description
Union $p\ p' \equiv$ Union $p'\ p$	*Union is commutative*
Intersection $p\ p' \equiv$ Intersection $p'\ p$	*Intersection is commutative*
Union (PSet x)(PSet y) $\equiv$ PSet $(x \cup y)$	$\cup$ *distributes over Union*
Intersection (PSet x)(PSet y) $\equiv$ PSet $(x \cap y)$	$\cap$ *distributes over Intersection*
Union Always $p \equiv$ Always	*Always is annihilator for Union*
Intersection Always $p \equiv p$	*Always is identity for Intersection*
Inverse (Inverse p) $\equiv p$	*Inverse is involutory*
Inverse (Intersection (PSet x)(PSet y)) $\equiv$ Union (Inverse (PSet x))(Inverse (PSet y))	*De Morgan laws*
Union (Intersection (PSet x)(PSet y)) $\equiv$ Intersection (Inverse (PSet x))(Inverse (PSet y))	

The Rules DSL utilizes a combination of these laws to optimize patterns before usage. The result of this is that, for instance, `Union p Always` is substituted with `Always` transparently before usage, such that `p` is never used. This is done by having a function `optimizePattern` that optimizes a pattern, and that is called before any pattern is used in a `Rule`. A small part of that function can be seen in Listing 15. Note that the ability to inspect and optimize values like this is inherent to deep-embedded DSLs; if arbitrary functions were used instead, it would be impossible to inspect the functions and perform such optimization from within Haskell.

```
-- | Optimizes a pattern, by distributing intersections over unions, combining
sets, etc
optimizePattern :: Pattern -> Pattern
optimizePattern (Inverse i) = case optimizePattern i of
  -- De Morgan
  (Union a b) -> optimizePattern (Intersection (Inverse a) (Inverse b))
  (Intersection a b) -> optimizePattern (Union (Inverse a) (Inverse b))
  Always -> never
  -- Cancel inverse
  (Inverse x) -> optimizePattern x
  x -> Inverse x
optimizePattern (Intersection ...) = ...
...
```

Listing 15 : The `optimizePattern` function that optimizes a pattern

3.4 Sequences in grammars and programming languages

Ordered linear data structures may be implemented differently depending on the programming language paradigm. In imperative programming languages, dynamic arrays, a data structure that provides random access to its contained elements and that may dynamically increase in size, are commonly employed. In some high level languages, these may be part of the language specification, whilst in system programming languages they are generally provided by a standard or third party library[12]. Another linear data structure are linked lists, which are commonly employed in functional languages, especially in those that emphasizes purity. These two data structures exhibit various properties that affect their usage in different ways.

Dynamic arrays, being implemented using arrays, in which elements are stored in contiguous memory, allows for constant time random access to its elements. Furthermore, appending elements to the *end* of a dynamic array can be accomplished in amortized constant time. Conversely, prepending elements to the *beginning* of a dynamic array requires linear time with respect to the number of elements contained within the array at the point of the operation. Thus, dynamic arrays provide efficient means for the gradual expansion of an ordered sequence of elements, when the elements are inserted in their desired order.

Each element of a linked list is stored in a separate node, which is connected by means of a pointer to the node of the following element in the sequence[13]. The pointer to the root node is stored in the structure itself. The nodes of a linked list thus form a graph or, more specifically, a degenerate tree. This structure allows for constant time prepending of elements to the beginning of the list, as it only requires the altering of the pointer to the root node and that of the node to be prepended. Appending an element to the end of the list, on the

[12]Scripting languages such as ECMAScript and Python provide dynamic arrays as part of the language specifications. On the other hand, in system programming languages such as Rust and C, they are instead provided by standard and third party libraries, respectively.

[13]This description is accurate for a *singly* linked list. However, other variations exist, such as a *doubly* linked list, in which each pair of adjacent nodes forms a cycle.

other hand, necessitates that the entire list be traversed, an operation exhibiting linear time complexity[14].

Another aspect of these data structures that may greatly affect run-time performance of a program is *spatial locality* [39]. Caches in modern processors have the potential to drastically reduce the latency of reading memory, provided that the requested memory is resident in a cache. The probability of a memory address residing in a cache is significantly increased, should a memory address close to the one requested have recently been loaded. Greater spatial locality thus generally equates to lower latency of memory retrieval and thus greater run-time efficiency. On account of the elements of dynamic arrays being stored in continuous blocks of memory, they naturally exhibit *sequential locality*. The same, however, cannot be said with regard to linked lists; the elements of linked lists, whilst affected by the behaviour of the memory allocator employed, are generally stored in a fragmented manner on the heap, and hence do not exhibit high spatial locality.

3.4.1 Parsing sequences

Sequences of elements of arbitrary length can be described by a context-free grammar through recursive definitions. A non-terminal representing a sequence may be defined by two production rules: one expressing the recursive case and the other the base case, which terminates the recursion. However, there are two ways in which the recursive production rule may be defined: the non-terminal representing the recursive structure may either precede or succeed the symbol representing an element. The former is known to be *left-recursive* or *left-associative*, and likewise, the latter is referred to as being *right-recursive* or *right-associative*. For the purpose of illustration, Grammar 10 provides two context-free grammars, the languages of which are identical and contain non-empty sequences of the capital letter A, separated by asterisks, where the left-hand grammar is left-recursive and the right-hand is right-recursive. The languages of both grammars thus include sentences such as A, $A * A$, $A * A * A * A$, etc.

```
SRecurse. S ::= S "*" "A" ;          SRecurse. S ::= "A" "*" S ;
SBase.    S ::= "A"       ;          SBase.    S ::= "A"       ;
```

Grammar 10 : Left- and right-recursive definitions of a sequence of letters A separated by asterisks, in LBNF

Whilst sequences, such as the one presented above, may equally be constructed using left- or right-recursive definitions, the type of recursion has ramification on the manner by which they can be parsed. Firstly, the class of grammar may impose restrictions on the type of recursion that is permitted. An LL(k) grammar strictly prohibits any left-recursion, thus leaving right-recursive definitions as the only option. On the other hand, an LR(k) grammar permits left-recursive definitions but does not prohibit all right-recursive ones. Thus, parser generators that construct LL(k) parsers require all sequence definitions to be right-recursive, whilst left-

[14]This is under the assumption that the linked list does not store a pointer to the ultimate element of the list, in addition to that of the root node. Whilst this revision might seem trivial to implement, it is usually absent in functional programming languages such as Haskell.

recursive definitions may be advantageous for those producing LR(k) parsers. Furthermore, the programming language targeted by a parser generator, with respect to the sequence data structure that it employs, determine the type of recursion that the generated parser can more efficiently parse. Parsers that represent sequences by dynamic arrays can more efficiently parse those defined by left-recursion. Conversely, those that represent sequences by means of linked lists are better suited to parse sequences defined by right-recursion.

For the purpose of illustration, consider the following. In the programming language Rust, a dynamically sized sequence of elements is most typically represented by the standard library structure `std::vec::Vec`, an implementation of a dynamic array. This structure provides efficient appendment of elements through the method `std::vec::Vec::push`, an operation that exhibits amortized constant time complexity, as explained above. A grammar for a language, the sentences of which are to be parsed by a parser implemented in Rust, may thus expediently employ left-recursion, should sequences be defined for that language. A functional programming language, such as Haskell, may on the other hand employ a linked list data structure, e.g. the Haskell list type `[]`, for the same purpose of representing sequences. As such, grammars may instead define sequences by right-recursion, in order that lists be constructed more efficiently with the list *cons* constructor, in Haskell denoted `(:)`, by which an element is prepended to the beginning of a list.

Table 3 : Common data structures used to represent dynamic sequences and their respective efficient operations, prepend or append, in various programming languages

Language Specification	Idiomatic List	Efficient Operation
Haskell 2010 [33]	Linked list, `[a]`	Prepend
OCaml [34]	Linked list, `'a list`	Prepend
Java 21 [35]	Dynamic array, `ArrayList<T>`	Append
C23 [7]	Various[15]	
ECMAScript 2024 [36]	Dynamic array, `Array`	Append
Python 3.13.2	Dynamic array, `List`	Append
Rust 2024	Dynamic array, `Vec<T>`	Append

3.4.2 Enhancements of lists in BNFC

In BNFC 2.9.5, the `terminator` and `separator` pragmas, which facilitate the creation of sequences within an LBNF specification, both generate right-recursive definitions. Furthermore, left-recursive sequence definitions are explicitly disallowed in LBNF. Given that BNFC intends

[15] No data structures such as linked lists or dynamic arrays are included in the C standard library as defined by ISO/IEC 9899:2018 [7]. Although different implementations of this standard may include additional definitions, the widely used GNU C Library [40] defines neither linked lists nor dynamic arrays. Instead, software written in C may use implementations of lists as defined by third party libraries or composed by hand, typically in the form of dynamic arrays, on account of their run-time performance characteristics.

to support back-ends targeting programming languages of different paradigms, it is desirable, on account of increasing parser performance, grammar compatibility, and conformance to programming language conventions, that left-recursive definitions be permitted, in addition to right-recursive ones. Furthermore, it would be expedient if the aforementioned pragmas were to provide a means by which the type of recursion can be specified.

With these issues in consideration, it was decided that the definition of LBNF was to be amended and that the behaviour of BNFC with respect to the `terminator` and `separator` pragmas was to be revised. In order that it be possible to define left-recursive sequences, the LBNF specification was altered to include the left-recursive equivalent of the right-recursive `cons` operation, which was denoted by `(:)`, the symbol used in Haskell for list prependment. Thus, the special label `snoc` was added to the LBNF specification for this purpose. Furthermore, the labels `[]`, `(:[])`, and `(:)` were renamed to `empty`, `sing`, and `cons`, respectively[16], that the meaning and usage of these would be more comprehensible and not be based upon the conventions of Haskell, so as to render BNFC more language-agnostic. The result of the renaming of list labels can be seen in Figure 12.

```
[].     [Exp] ::=              ;        empty.  [Exp] ::=              ;
(:[]).  [Exp] ::= Exp          ;  ⟶     sing.   [Exp] ::= Exp          ;
(:).    [Exp] ::= Exp ";" [Exp] ;       cons.   [Exp] ::= Exp ";" [Exp] ;
```

Figure 12 : Renaming of the special list labels in LBNF

The definitions of the `terminator` and `separator` pragmas have been amended in order that they accept the additional optional qualifiers `left` and `right`, by which left- and right-recursion can be specified, respectively. Thus, the pragma invocation `separator left Exp ","` would define a left-recursive sequence of comma separated expressions, whilst the right-recursive equivalent could be defined by `separator right Exp ","`. Usage of this pragma with qualifiers is demonstrated in Figure 13.

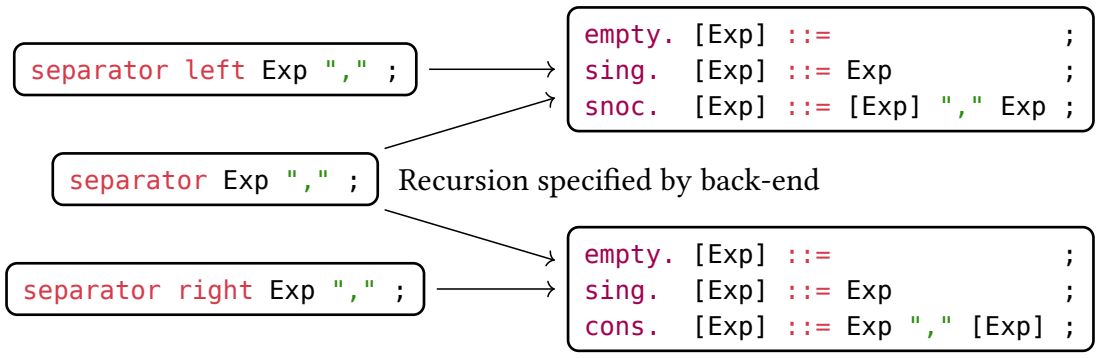


Figure 13 : Invocations of the `separator` pragma with and without the recursion qualifiers `left` and `right`, as well as the equivalent production rules

[16]Note that although the old labels are deprecated, they are nevertheless available for the purpose of backwards compatibility.

The default behaviour of the `terminator` and `separator` pragmas, when no recursion qualifier is specified, has also been revised. In order that performance, grammar restrictions, and programming language conventions are taken into account, the type of recursion is determined by each back-end through the declaration `defaultListRecursion` in the interface, as shown in Listing 16. Thus, leaving the type of recursion unspecified may produce either left- or right-recursive list definitions depending on the back-end specified, as seen in Figure 13.

```
instance Backend Rust where
  defaultListRecursion = LeftRecursion
  ...
```

Listing 16 : Declaration of the default type of recursion for list pragmas in the interface for the Rust back-end

3.5 Relation between tasks and files

When a back-end is invoked, it needs to fulfil certain *tasks*: creating a lexer specification, a parser specification, a makefile, etc. The result of fulfilling these tasks is multiple *output files*, that is, files that are to be written to disk. The interface needs to capture the nature of the relationship between these tasks and output files. The interface by Vergani [24] assumed a *disjoint one-to-many mapping*; each task was assumed to produce a set of output files, where all sets are pairwise disjoint. However, this solution is not sufficient, and the details of why that is will be discussed in this section.

3.5.1 Build tools and package managers

In order to execute a program, its source code must either be compiled into an executable or run by an interpreter. Depending on the programming language and compiler or interpreter used, different tools may be needed or expedient to employ for this purpose. Source files of a program in C and C++ need to be compiled and linked in order for an executable file to be created. This process is typically facilitated in larger projects by means of a *build automation tool*, an example of which is *Make*. Although Make is essentially ubiquitous on Unix-like systems, it is not as commonly found on those running Windows. Instead, a platform-independent *build system generator*, such as *CMake*, may be employed. It is used to transform a common configuration format into one of a platform-specific build tool, such as Make on Linux or Visual Studio on Windows. In the process of writing software comprising intermediate to large codebases, the utilization of libraries is ubiquitous. Libraries, which provide means by which standardized and dependable components may be distributed, facilitate efficient and robust software development. Build systems generally do not provide means by which third party libraries can be installed or managed. For this purpose, it is expedient to employ a *package manager*, to which required dependencies can be specified.

Software projects using interpreted programming languages such as JavaScript and Python generally do not need build systems in order to run the program, forasmuch as the interpreter

or runtime system can load the required code modules automatically. Package managers are, on the other hand, extensively used. Such is the case for JavaScript and Python, with which *NPM* and *Pip* are the most commonly used package managers, respectively.

Common to build systems and package managers is the use of configuration or manifest files that specify the behaviour of these tools. Furthermore, certain directory structures or files paths may need to be present in order to utilize these tools for a project.

For Rust, which is the newest of the languages compared, the build system and package manager Cargo is tightly coupled with the compiler itself. Whilst one may choose to use the compiler directly, the Rust standard library is smaller than many of the ones of other languages, with the general philosophy that packages may be used to add any desired functionality. For instance, the Rust standard library lacks facilities for random number generation, which is included in the standard libraries for C and C++, Python, and Java. This often leads to tens of dependencies, even for smaller projects, which in turn necessitates a convenient build tool and package manager. Cargo requires a manifest file, `Cargo.toml`, in the project, providing at minimum the name of the project, and optionally listing all dependencies, and compilation options.

Haskell developers often employ the package manager and build tool named the *Common Architecture for Building Applications and Libraries* (Cabal). A large difference with Cabal compared to previously mentioned tools is that all Haskell modules – that is, all Haskell files in the project – explicitly need to be mentioned one by one. For developers, this means that as soon as a project expands by one file, the Cabal file needs to be updated too. The tool `hpack` [41], which generates a Cabal manifest file from a simpler format, can, however, be used to remedy this inconvenience.

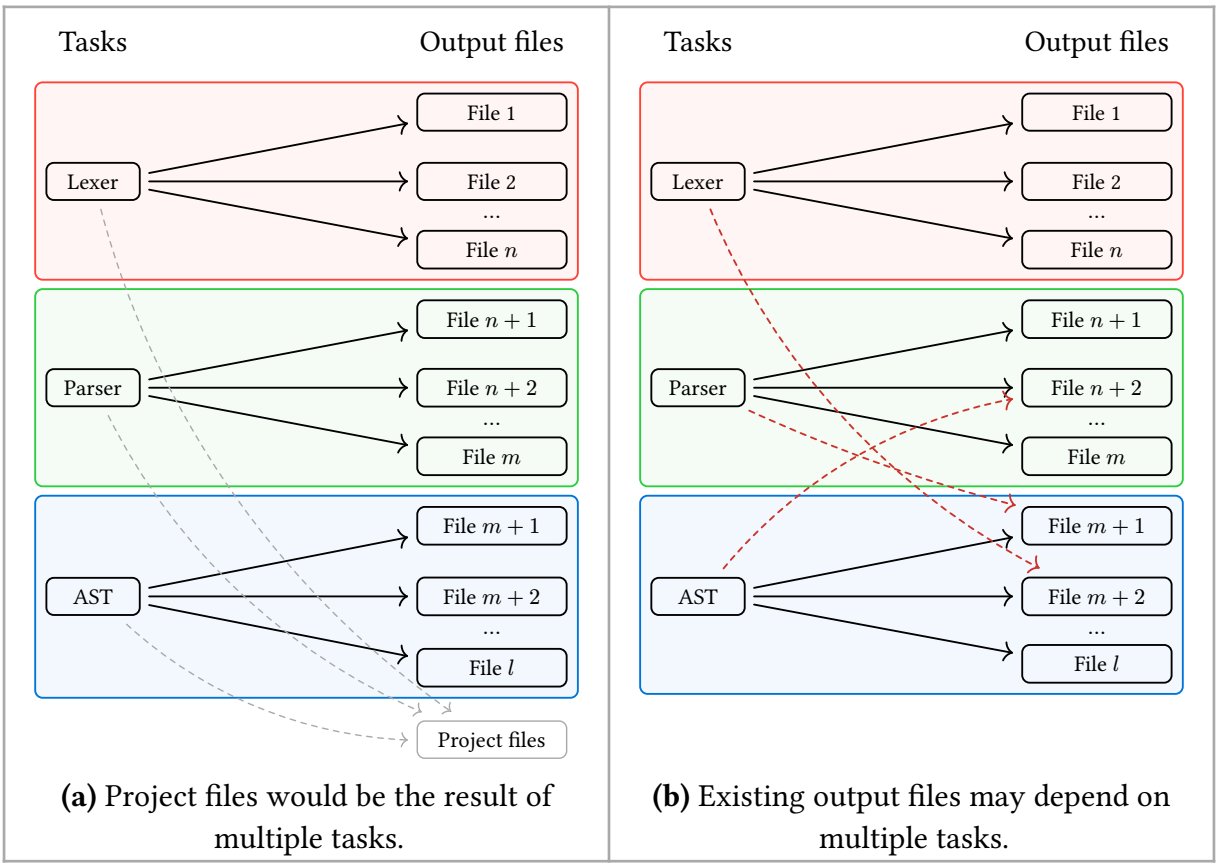
Currently, the disjoint one-to-many mapping of tasks to output files puts a limitation on what can be achieved by the back-ends; the project files that are often required do not correspond to any one task but rather to the union of tasks. The structure of having a given number of tasks that runs in a specific order does not currently account for this. A possible solution to this would be to define an additional task, which should generate 'additional files'. However, this solution does not take into account that the project files may be dependent on the output of previous tasks – if one would expand BNFC by allowing the user to disable the linearizer task, for instance, the Cabal project file would need to change as well. This 'hidden' dependency, breaking the disjoint one-to-many mapping, is illustrated in Figure 14 (a). This was not taken into consideration during the development of the interface of Vergani.

3.5.2 Language-imposed requirements

Programming languages may pose requirements that break the disjoint one-to-many relationship between tasks and output files. This relationship hints at each file being generated by exactly one task, which is certainly not always the case. The lexer and parser specifications are inherently coupled, as the names of the lexer tokens are used in the parser specification. This is solved in the existing back-ends by pre-computing the names of the lexer tokens already during state initialization, which happens before any of the tasks run. In the end, one may argue that the separation is still valid since each task produces its own file.

However, in languages like Rust, another problem occurs. The Haskell back-end produces a linearizer that simply declares an instance of the `Pretty` type class for each abstract syntax type. This is done in a separate file than the one defining the abstract syntax, rendering the output of the linearizer task different from that of the abstract syntax task. In Rust, it is desired to implement the `Display` trait, which is analogous to the `Pretty` type class. However, this can only be done from within the same file as the abstract syntax is defined, meaning that both the abstract syntax task and the linearizer task need to both operate on a single file. This breaks the relationship assumed by the interface of Vergani, and it is thus necessitated that the same task be used to output both the abstract syntax and the linearizer to the same file. Rust is an example where this structure does not hold.

There may also be back-ends for which it is required or more convenient that the parser and lexer generator specifications be defined in the same output file. In such a case, only one function would be needed to generate the single file for the lexer and parser specifications, even though the interface requires that all back-ends define separate functions for the parser and lexer tasks. This is indeed the case should a back-end be created that employs the lexer and parser generators provided by the Rust library LALRPOP [42], inasmuch as the lexer and parser specifications are prescribed to reside in the same file. One even simpler example stands out; if one would write a back-end producing an (expanded) LBNF file, where once again, the parser and lexer specification is in the same file.



(a) Project files would be the result of multiple tasks.

(b) Existing output files may depend on multiple tasks.

Figure 14 : The disjoint one-to-many mapping between tasks and files may break.

Furthermore, the existing documentation-generating back-ends, such as those of the Latex and the Txt2Tags back-ends, do not follow the previous task structure. Rather, the responsibility of creating the contents of the single document file is conferred upon one of the tasks; the remaining tasks serve no purpose. Moreover, the name of the one task-function employed, whichever is designated for the purpose, does not accurately portray the manner in which it is used, namely that of producing a text formatted document.

The issues associated with existing output files being dependent on multiple tasks is illustrated in Figure 14 (b). This issue, in combination with those presented in Section 3.5.1, led to the the development of the resulting interface seen in Listing 3 – compare the `runBackend` function, in which all required tasks are executed, with the task-specific functions in Listing 2. The new, singular function now allows the back-ends to split the work into any number of tasks as they see fit. Combining sub-tasks is simply done by the Haskell function `sequence`, so the old behaviour can be replicated with very little effort from back-end developers if desired.

3.6 Features

The BNF Converter dates back to 2004 [43]. As the project develops, new back-ends are added, others are removed[17], and some back-ends are updated[18]. Some features are revised, such as Burreau improving layout syntax support in 2023 [25]. Substantial effort is required to ensure that every back-end supports every feature. This effort has not been done; for example, positional tokens are not supported by the C++ 'No STL' back-end[19].

Furthermore, the back-ends target different parser generators, which may generate parsers of different classes, such as *LL(1)*, *LR(1)*, *SLR(1)*, or *LALR(1)* parsers. Since some grammars that are expressible in LBNF may not be supported by all current and future parsers and parser generators, BNFC may not assume that any valid input generates valid output in a specific back-end. The same issue is present with lexer generators; BNFC has a notion of arbitrary subtraction of regular expressions, whilst this is only supported on the character class level by most lexer generators.

Complicating things further, the encoding of the AST in the target language may also be problematic for some grammars. Firstly, reserved keywords in the target language may restrict certain names; this is discussed in Section 3.3. Secondly, the encoding of names in the target language is inherently target-specific. In the Haskell back-end, for example, categories correspond to data types and rule labels to data constructors. In the Java back-end, on the other hand, both categories and labels are represented by classes. Owing to the fact that multiple classes may not be defined with the same name, an LBNF specification containing a label that shares its name with a category cannot yield a valid output in the Java back-end. As for the Haskell back-end, a data type and a constructor may be defined with the same name; an LBNF specification may be a valid input, in regard to the relationship between names of labels and categories contained within it, for one back-end, whilst, simultaneously, being invalid for another.

[17] The F# and Java 1.4 back-ends were removed in BNFC version 2.7.0.0

[18] A Java back-end targeting Java 1.5 replaced the one targeting Java 1.4

[19] This is a variant of the C++ back-end that does not depend on the Standard Template Library (STL).

This issue is solved in BNFC 2.9.5 by a hard-coded list of back-ends that do not support the same name being used for both a label and a category, referred to as *duplicate names*. The list of back-ends that do not support this is defined in the front-end. If two names collide, and the active back-end is in that list, the front-end terminates with an error; if the back-end is not in that list, a warning is emitted and compilation continues. This can be seen in Listing 17. In the unreleased version of BNFC, duplicate names are not accounted for.

```
case nonUniqueNames of
  [] -> return ()
  names | target `elem` [ TargetC, TargetCpp, TargetCppNoStl, TargetJava ]
        -> dieUnlessForce $ unlines $ concat
            [ [ "ERROR: names not unique:" ]
            , printNames names
            , [ "This is an error in the backend " ++ show target ++ "." ]
            ]
        | otherwise
        -> putStrLn $ unlines $ concat
            [ [ "Warning: names not unique:" ]
            , printNames names
            , [ "This can be an error in some backends." ]
            ]
```

Listing 17 : The check in BNFC 2.9.5 for duplicate names

Whilst the validation mechanism of BNFC version 2.9.5 works for checking duplicate names, it is clear that many more checks are required for the front-end to ensure that an LBNF specification is supported by a given back-end. This is possible, however, rather than the more logical idea of each back-end indicating what feature it supports, this solution would require multiple hard-coded lists similar to the one in Listing 17 to be maintained in the front-end.

If these checks are instead conducted by each back-end, in such a way that a back-end may reject an incompatible LBNF specification, there would be several issues. Firstly, significant code duplication may occur, inasmuch as similar checks may be required to be done in multiple back-ends. Secondly, there is a risk that the same check is done differently in different back-ends, which may lead to inconsistent behaviour. Finally, should additional capabilities be added to LBNF, each back-end by which these are not supported would require modification in order to explicitly reject specifications utilizing these capabilities. A sane behaviour, with respect to new capabilities, would be that all specifications using these be rejected by default and that support for these must be explicitly declared.

3.6.1 Declaration of supported features

As a solution to the aforementioned problems, the Features interface, shown in Listing 18, was created.

As a part of the `Backend` type class, shown in Listing 3, a value of type `Features` is provided by the back-end to the front-end. This data type has a field for each feature that a back-end may or may not support. The value of each field is of type `FeatureSupport`, where a back-

```
-- | The sets of features that a backend may or may not support.
data Features backend = Features
  { -- | Whether the back-end supports "duplicate" names.
    featDuplicateNames :: FeatureSupport backend
  -- | Whether the back-end supports the LBNF layout syntax.
  , featLayoutSyntax :: FeatureSupport backend
  -- | Whether the back-end can output a Makefile.
  , featMakefile :: FeatureSupport backend
  -- | Whether the back-end supports LBNF position token rules.
  , featPositionTokens :: FeatureSupport backend
  -- | Whether the back-end supports arbitrary difference
  -- of two regular expressions.
  , featRegexDifference :: FeatureSupport backend
  }

-- | A feature can, for a given backend, either be supported, unsupported, or
have its support
-- dependent on some specific condition (such as command-line options, or the
grammar itself)
data FeatureSupport backend
  = -- | The feature is supported by this backend.
    Supported
  | -- | The feature is not suppored by this backend.
    Unsupported
  | -- | The support of this feature is dynamically determined by a
    -- 'BackendExec' action. A description of the conditions that
    -- governs the conditional support is also to be supplied.
    Dependent (BackendExec backend Bool) Text
```

Listing 18 : The `Features` data type represents features that a back-end may optionally support. Whether a back-end supports a given feature is represented by the `FeatureSupport` data type.

end may indicate that the given feature is either `Supported`, `Unsupported`, or that the feature support is dependent on the result of a `BackendExec` computation. The motivation of letting a back-end perform a `BackendExec` computation to evaluate if a feature is supported or not is that some features may be dependent on command-line options. For example, in BNFC 2.9.5, there exist two back-ends for OCaml, utilising two different lexer generators. If some features are only supported by one of the lexer generators, one may combine the two back-ends to one single back-end, and utilising the `BackendExec` computation, the back-end may indicate different supported features depending on the chosen lexer generator.

Five features that are supported by LBNF but that are not supported by all BNFC back-ends have been identified:

- collision of names, which is, for example, not supported by the Java back-end;
- layout syntax, which is only supported by the Haskell back-end;
- generation of makefiles, which may be a benefit for some target languages, but makes less sense for languages with a well-integrated package manager and build system;
- positional tokens, that is, tokens that bear information pertaining to its position in the source code; and

- subtraction of arbitrary regular expressions.

In Stage 3 of the LBNF validation pipeline (cf. Figure 9), the BNFC front-end deduces what features are required in order for the back-end to successfully compile the given LBNF specification. Then, the active back-end is queried for those features; if any of the required features are unsupported by the back-end, an error is emitted and the LBNF checking fails.

3.6.2 Features as an additive monoid

Should additional features be added to the `Features` data type, it would be highly desired, with respect to the maintainability of BNFC, that the code of existing back-ends would not require any modification. For this purpose, it is prudent not to expose the constructor of the `Features` data type, lest it be required that every use of the constructor in the entire codebase be updated in order to accommodate the new feature. This established, another means of specifying supported features is to be sought.

A potential solution is to construct a function for every combination of features that are not to be set to the default `Unsupported` value. Whether these be written by hand or automatically generated, this solution would be unfeasible; for n features, a total of 2^n functions of this sort would be needed.

Instead, by treating the `Features` data type as an *additive monoid*[20] [44], represented by the type class `Monoid` in Haskell, a more elegant solution emerges; for each feature, two functions yielding the `Features` type can be defined. In one, only the field of the feature in question is set to `Supported` – and all other to `Unsupported`. In the other function, the feature field is set to `Dependent`. As such, there are two functions pertaining to duplicate names: `supportsDuplicateNames` and `duplicateNamesIf`. In order to specify support for multiple features, two values of the `Features` type may be combined through the additive binary operation of the monoid, defined as `(<>)` for the Haskell type class, e.g. `supportsDuplicateNames <> supportsLayoutSyntax`. In Listing 19, the support for duplicate names and layout syntax, as well as conditional support for position tokens is declared in three ways: firstly by direct use of the `Features` constructor, secondly by the unfeasible solution mentioned above, and thirdly by additive monoidal operations.

When a new feature flag is added to BNFC, a new field is added to the `Features` record type, together with two new feature constructors: `supports{Feat} :: Feature` and `{feat}If :: BackendExec b -> Text -> Features b`. Any existing back-ends will automatically have that feature marked as `Unsupported`. Then, a guard is added to Stage 3 of the LBNF checking pipeline (see Figure 9) to ensure that if the feature is required, it must also be supported by the current back-end. This allows developers to add new features to BNFC and only supporting them in some back-ends, without having to worry about older back-ends receiving unexpected input. The API works similarly to *feature toggles*, with the exception that the `Dependent` case is context-dependent [45].

[20] In an additive monoid, the 'sum' of two terms is greater or equal than its operands, according to some preordering.

```
instance Backend Foo where
  features = Features
    { featDuplicateNames  = Supported
    , featLayoutSyntax    = Supported
    , featMakefile        = Unsupported
    , featPositionTokens  = Dependent cond "description"
    , featRegexDifference = Unsupported
    }
```

```
instance Backend Foo where
  features = featDuplicateNamesLayoutSyntaxPositionTokens
               Supported Supported (Dependent cond "description")
```

```
instance Backend Foo where
  features =
    supportsDuplicateNames <>
      supportsLayoutSyntax <>
      positionTokensIf cond "description"
```

Listing 19 : Equivalent usages of three candidates for the Features data type

4

The Rust Back-end

For the purpose of evaluating the back-end interface with respect to the objectives declared in Section 1.2 and improving it – were it not to adequately fulfil these requirements – a back-end targeting the programming language *Rust* was developed.

4.1 The Rust programming language

Rust is a modern system programming language [46] that has seen wide adoption in a variety of fields and several large software projects during its relatively short period of existence[21]. The extensive use of Rust stems from its inherent *memory safety*, which is accomplished by way of a *Resource Acquisition Is Initialization* (RAII) [47] memory management model and a *borrow checker*. Furnished with these measures, as well as other modern language features[22], Rust is an instrument by which faults in software relating to memory or thread safety can be greatly reduced, whilst maintaining run-time performance comparable to that of C and C++ [48]. In fact, its has been reported by The Chromium Projects, which develops the browser engine Chrome, that 70% of severe security bugs were caused by issues relating to memory safety [49]. Reports of equal nature have also been issued by Microsoft [50].

Based on these considerations, Rust has been employed in many large-scale software projects, such as the Android Open Source Project [51] and the Windows kernel [52]. The programming language has also seen extensive use in compilers. Other than the compiler of Rust itself [53], projects such as Typst [54], SWC [55], Roc [56], Gleam [57], Fish [58], Wasmer [59], Wasmtime [60], and an implementation of Starlark [61] all use Rust as the programming language of choice.

On account of the qualities and emergence of Rust laid out herein, it is to be considered advantageous and proper for the BNF Converter that a back-end targeting the Rust programming language has been developed in the scope of this thesis.

[21] The first stable release of the Rust programming language was on 5 May 2015.

[22] Through the rejection of null pointers in favour of *algebraic data types*, used in conjunction with *pattern matching*, Rust equips the programmer with convenient tools, whilst also promoting sound coding practices.

4.2 Cargo

The Rust toolchain, used by developers for developing programs using Rust, contains *Cargo*, the official build system and package manager for Rust [62]. Cargo facilitates dependency management; only a list of dependencies with their respective versions, need to be supplied. It will then automatically install these dependencies and link them when the project is built. This is in contrast to how dependencies are managed in C and C++, where library installation, linking, and compatibility require more manual work. Due to its convenience, Cargo has been reported as the second greatest benefit of using Rust [63].

To use Cargo in a Rust project, one simply needs to add a manifest file to the directory, named `Cargo.toml`. This makes the project a Rust *crate*. A minimal manifest file contains just the package name, but may additionally include dependencies, language version, compiler flags, and more. When requested to build or run a crate, Cargo will search in the `src` directory for a file named `lib.rs` for building a library crate, and for `main.rs` to build an executable crate.

Additionally, Cargo allows for the use of *build scripts*, which are Rust source files attached to a crate that is run before that crate is built. The build script may execute arbitrary work before the crate compiles, but are often leveraged to set up dynamically linked libraries according to the host environment, to pre-process some files and include a processed version of them in the final crate, or to set up environment variables for the build process. At most one build script may be used per crate.

4.3 Implementation

As part of this project, a new BNFC back-end has been developed that targets Rust. The output of the back-end is a set of files that together constitute a crate containing a lexer and parser specification, along with an AST, linearizer and a *parse-and-print program*.

For the purpose of illustration, consider the grammar given in Grammar 11. The result of compiling this grammar using the Rust back-end of BNFC will be shown in this section.

```
ELet.       Exp  ::= "let" Ident "=" Exp "in" Exp ;
EAdd.       Exp1 ::= Exp1 "+" Exp2               ;
EMul.       Exp2 ::= Exp2 "*" Exp3               ;
ELit.       Exp3 ::= Double                      ;
EVar.       Exp3 ::= Ident                       ;
coercions   Exp  3                               ;
entrypoints Exp                                  ;
comment     "//"                                 ;
```

Grammar 11 : An LBNF specification for a language of expressions

4.3.1 Lexer

The resulting lexer specification is Rust source code defining a *sum type*, which is equivalent to the resulting data type that the Haskell back-end produces. Example of such code can be seen in Listing 20. Attached to this data type are certain *attributes*, such as `#[derive(Logos)]`, `#[token("let")]`, and `#[regex(...)]`.

```
#[derive(Logos, Debug, Clone)]
#[logos(error = Error)]
#[logos(skip r"//[^\n]*\n?|[ \t\n]")]
pub enum Token {
    #[token(r"let")] KW_let,
    #[token(r"in")]  KW_in,
    #[token(r"(")]   SYM_lparen,
    #[token(r")")]   SYM_rparen,
    #[token(r"*")]   SYM_star,
    #[token(r"+")]   SYM_plus,
    #[token(r"=")]   SYM_eq,
    #[regex(
        r"[a-zA-Z][a-zA-Z0-9_']*",
        callbacks::custom)]
    TIdent(String),
    #[regex(
        r"\d+\.\d+(e-?\d+)?",
        callbacks::double)]
    Double(f64),
}

#[derive(Default, Debug, Clone)]
pub enum Error {
    #[default] NoTokenMatch,
    ParseInt(ParseIntError),
    ParseDouble(ParseFloatError),
}
```

Listing 20 : A Logos lexer specification for the grammar shown in Grammar 11

The `#[derive(Logos)]` attribute invokes a macro named `Logos`, which collects all the `#[logos]`, `#[token]`, and `#[regex]` attributes to build, at compile-time, a *deterministic finite automaton* that efficiently recognizes tokens. The `#[logos(skip ...)]` attribute takes a regular expression which the lexer tries to match and then discards. In this case, it is used to skip all whitespace characters and all characters starting from `//` to the next end-of-line control sequence, the latter of which correspond to line comments. The `#[token(...)]` and `#[regex(...)]` attributes, which are attached to the variants of the data type itself, specify what string literal or regular expression that produces that variant. If a variant is specified using a regular expression, some post-processing may be done to produce the appropriate variant, such as `callbacks::integer`, which parses the integer as a numeric value. When

using positional tokens, the token variant will contain the position represented as the byte offset range to the matched tokens.

As can be seen in the example, `f64` (a double-precision floating point number) is used as the default internal representation for doubles. For integers, `usize` is used, which is the unsigned machine-word sized integer type. If another representation is desired, the user may choose one using command-line options `--float=f32`, `--integer=i128`, etc.

Besides the Logos lexer, an Error type is generated. This Error type represents all errors that may occur during lexing, which is that some input may fail to generate a token, or that parsing an integer or double may fail due to overflow. The lexer itself will take a string as input and generate a stream of either valid tokens or errors, which in Rust is represented as an `Iterator<Item = Result<Token, Error>>`.

4.3.2 Abstract syntax

The Rust back-end generates a set of data type definitions, which act as nodes in its abstract syntax representation. Thus, structure and enumeration items in the generated Rust source code represent the non-terminal symbols in the provided LBNF specification. For instance, the non-terminal symbols utilized in Grammar 11, viz. `Ident` and `Exp`, correspond to the structure and enumeration items of the same names in Listing 21. It is to be noted that the predefined tokens, which are enumerated in Section 2.4, with the exception of the `Ident` token, are not represented by a structure definition, but by the equivalent Rust data types. The `Double` and `Integer` tokens are thus represented by primitive floating point and unsigned integer types, such as `f64` and `usize`, respectively, and are not contained by an enclosing structure. Likewise, the `String` token is represented by the `std::string::String` structure, defined in the standard library, but the `Ident` token is, however, represented by the structure `Ident`, as seen in Listing 21, so that it may not be conflated with the representation of the `String` token.

```rust
#[derive(Debug, Clone)]
#[repr(transparent)]
pub struct Ident(pub String);

#[derive(Debug, Clone)]
pub enum Exp<'a> {
    EAdd(&'a Exp<'a>, &'a Exp<'a>),
    ELet(&'a Ident, &'a Exp<'a>, &'a Exp<'a>),
    ELit(f64),
    EMul(&'a Exp<'a>, &'a Exp<'a>),
    EVar(&'a Ident),
}
```

Listing 21 : The non-terminals `Ident` and `Exp` in Grammar 11 are represented by structure and enumeration items in the Rust abstract syntax tree. Each of the variant constructors of the latter corresponds to a production rule of `Exp`.

It is a common practice to augment syntax trees with additional information, such as in the process of type checking, in which the nodes of the tree may be augmented with additional type information [22]. Thus, the unannotated syntax tree, as produced by the parser, can be *annotated* by including additional information in the nodes of the trees. The notion of annotated trees should not be confused with that of annotated terminal symbols, as described in Section 2.2.2. For the sake of example, consider a subsequence of symbols in some programming language that represents a function invocation on the result of an addition: `isPrime(n + 5)`. Upon lexical analysis having been conducted on this string, the following sequence of tokens may be obtained: `IDENT(isPrime)`, `SYM_LPAREN`, `IDENT(n)`, `SYM_PLUS`, `INT(5)`, and `SYM_RPAREN`. These tokens serve as input to a parser, which yields an unannotated abstract syntax tree. This AST may thence be subject for modification in subsequent stages of the compilation process, such as type checking. An annotated abstract syntax tree, exhibiting the same structure as the unannotated one, may thus be furnished with additional type information, such as that illustrated in Figure 15, in which each node contains an annotation for its type representation.

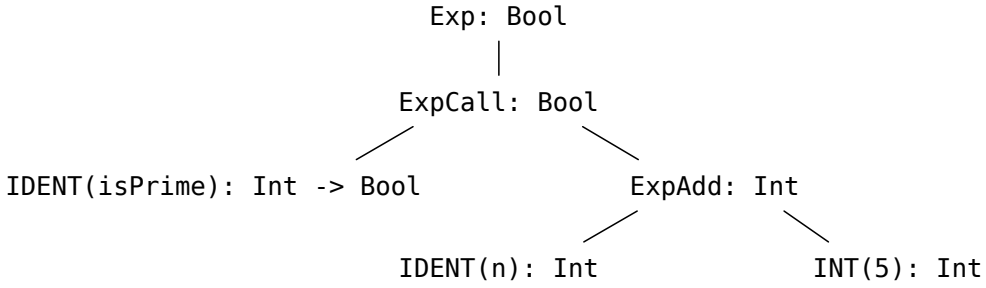


Figure 15 : An annotated abstract syntax tree representing the string `isPrime(n + 5)` and the types of its syntactic constituents

Whilst an algorithm that traverses a tree and substitutes annotated nodes for unannotated ones can easily be implemented, it may negatively affect run-time performance, if each annotated node needs to be copied in order to accommodate the additional annotation data. A strategy of this kind would thus effectively make a copy of the complete tree, which may be of some significance with respect to the execution time of the program, should the tree be large. Another method is to reserve space in the data structures that constitute the nodes of the tree. Then, the unannotated node contains some *default* value, which can eventually be substituted by the annotation value, when the tree is to be annotated. This process, as described, can be implemented without copying the tree; it is only necessary to replace the unannotated default values with those constituting the annotations.

Abstract syntax trees are, as all trees, logically composed of a set of nodes which are interconnected by edges. In a computer program, the nodes, in conjunction with any associated data, is represented by some data structure that may vary depending on which non-terminal in the grammar it corresponds to. In programming languages that employ static type checking, such as Rust, the type of a node can be attained through statical analysis at compile time. However, nodes representing non-terminal symbols with more than one associated production rule typically need to be disambiguated through run-time information, which is commonly implemented using an unsigned integer tag in the node data structure.

The manner by which the edges of a tree are represented in a computer program may vary depending on the implementation. A representation of an edge can either be stored within the data structure that represents a node or detached from it in a separate data structure. In the case of the former, the data structure of the parent node contains the edge to the data structure of the child node. As for the latter, a table can encode this parent-child relation, where the parent node, acting as the key, is used to attain the node of the child. One typical implementation of a tree follows the former manner of representing edges, namely, by means of pointers. In such an implementation, the pointer representing an edge indicates the memory address of the data structure representing the child node. The data structures representing the nodes can in this model be stored anywhere in memory. Generally, the memory for the data structures are allocated on the heap, separately for each node. Thus, the spatial locality exhibited by the data of the nodes, and the tree in its entirety, is to a great extent under the discretion of the memory allocator used. In practice, *fragmentation* may occur [64] when a general-purpose allocator is employed, such as certain implementations of `malloc` of the C standard library. This may significantly inhibit the run-time performance of a program that traverses a tree [65].

Alternatively, other memory allocation strategies can be leveraged, such as *bump allocation*, which is also known as *stack allocation*. An allocator employing this strategy operates by maintaining a pointer into a large region of consecutive memory, known as an *arena*. At the address of the pointer, the next object to be allocated is written, upon which the pointer is advanced by the size of the allocated object. As such, the objects allocated in this manner are stored consecutively in memory, provided that they be allocated within the same arena. An illustration of an arena employed by a bump allocator is displayed in Figure 16. Furthermore, this strategy incurs little overhead and may reduce the number of system calls to the operating system. Thus, data allocated in this manner naturally exhibit high sequential locality, and the number of inefficient operations used is reduced, which may significantly improve run-time performance, as compared to general-purpose allocation strategies. However, it is limited by the fact that it only allows for the deallocation of the most recently allocated object or, alternatively, the entire arena. Other strategies may thus be more suitable for general-purpose allocators, although for certain applications, such as that of constructing abstract syntax trees in a compiler, bump allocation is generally preferable; in a compiler, the data structures representing the nodes of the AST are constructed and allocated, by the parser, in a linear fashion after one another and are typically deallocated all at once.

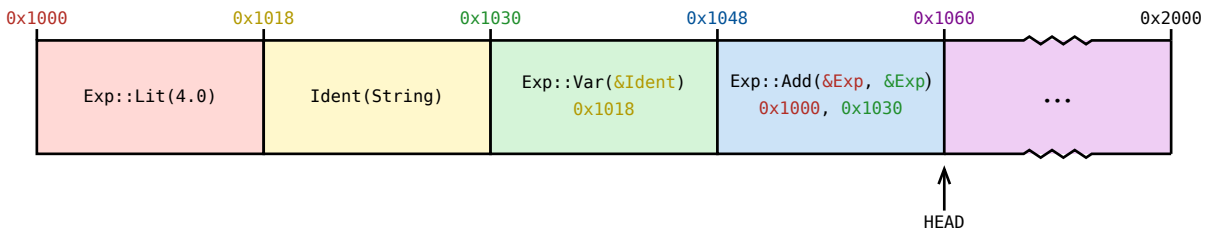


Figure 16 : An arena as employed by a bump allocator, where `HEAD` points to the memory address of the next object to be allocated

The abstract syntax tree implemented by the Rust back-end employs the bump allocation strategy through the use of the *Bumpalo* [66] crate. A simpler and more traditional approach, namely that of employing the default general-purpose allocator, can also be leveraged, which is enabled by means of a command-line option.

4.3.3 Parser

The Rust back-end utilizes the LALRPOP parser generator [42]. LALRPOP generates LR(1) parsers that are optimized according to the Lane Table algorithm, as presented by Pager and Chan in 2012 [67]. In contrast to the LR(1) variants SLR(1) and LALR(1), presented in Section 2.2.3, which have lower expressibility than LR(1), parsers constructed with the lane table method are just as expressive as LR(1) parsers whilst being much smaller in size and more efficient at parsing.

The generated parser specification can be seen in Grammar 12. This specification is, at compile time, processed by LALRPOP to generate the parser in the form of Rust source code. The generated parser for this example consists of 980 lines of code, most of which are the generated tables. The processing is invoked from a build script, described in Section 4.2, and involves verifying that the grammar is *unambiguous*.

The grammar file itself consists of three principal parts: the grammar header, the non-terminal definitions, and an external lexer specification. These three parts are described in further detail below.

Grammar declaration

The first part of the grammar file is the `grammar` declaration, in which any generic parameters or arguments for other resources used in the grammar are specified. The type signature of the function serving as the entry point of the generated parser corresponds to this declaration.

The Rust compiler maintains a construct of *lifetimes* within the borrow checker that serves to enforce its *aliasing* rules. According to these rules, every value has an *owner* and it may be *borrowed* by means of a reference. A reference may either be *shared* or *exclusive*, where the latter allows for mutable access to the underlying value but the former does not. A *lifetime* refers to a section of code in which a reference is valid and it may not exceed, or *outlive*, the section of the referent. For instance, a reference to a value stored within the stack frame of a function may not be accessed after the function has returned and the stack frame has been destroyed; the *lifetime* of the reference to the value is at most that of the function scope. By means of lifetimes, the Rust compiler can guarantee[23], through static analysis, that no aliasing of *exclusive references* occurs. A lifetime may be named `'a`, which may be used as a generic parameter of a function or type declaration.

[23]This is under the assumption that the program uses the *safe* subset of the Rust language; a program may employ *unsafe* operations in order to disable certain safety measures of compiler, such as the borrow checker. This may be needed in order to perform some sound actions, inasmuch as the rules imposed by the compiler are generally overly restrictive.

```
grammar<'a, T>(arena: &'a bumpalo::Bump) where T: 'a + Default;

Ident: abs::Ident = "Ident" => abs::Ident::new(<>);
Double: f64 = "Double";

pub Exp: abs::Exp<'a, T> = {
    "let" <Ident> "=" <Exp> "in" <Exp> => abs::Exp::newELet(arena, <>),
    Exp1,
}

Exp1: abs::Exp<'a, T> = {
    <Exp1> "+" <Exp2> => abs::Exp::newEAdd(arena, <>),
    Exp2,
}

Exp2: abs::Exp<'a, T> = {
    <Exp2> "*" <Exp3> => abs::Exp::newEMul(arena, <>),
    Exp3,
}

Exp3: abs::Exp<'a, T> = {
    Double => abs::Exp::newELit(<>),
    Ident => abs::Exp::newEVar(arena, <>),
    "(" <Exp> ")",
}

extern {
    type Location = ();
    type Error = lexer::Error;

    enum lexer::Token {
        // Symbols and keywords
        "("   => lexer::Token::SYM_lparen,
        ")"   => lexer::Token::SYM_rparen,
        "*"   => lexer::Token::SYM_star,
        "+"   => lexer::Token::SYM_plus,
        "="   => lexer::Token::SYM_eq,
        "let" => lexer::Token::KW_let,
        "in"  => lexer::Token::KW_in,
        // Builtin tokens
        "Double" => lexer::Token::Double(<f64>),
        // Text tokens
        "Ident" => lexer::Token::TIdent(<String>),
    }
}
```

Grammar 12 : A LALRPOP specification for the grammar shown in Grammar 11

In Grammar 12, the first line constitutes the grammar declaration. In this case, the entry point of the parser, as represented by the grammar declaration, is respectively parameterized by the lifetime and type parameters `'a` and `T`, where the latter represents the type of the annotated data, described in Section 4.3.2. Furthermore, the argument of the grammar declaration is a reference to a bump allocator, by which the AST nodes are allocated. Should the standard general-purpose allocator be desired, this argument would not be declared.

Non-terminal definitions

Non-terminal definitions consist of four parts, highlighted in Listing 22. The portion denoted by ① declares the visibility modifier and name of the non-terminal. Thus, `pub Exp` declares a non-terminal named `Exp`, which is public, and thus exposed by LALRPOP to the rest of the Rust code. The non-terminals which are deemed entry points by BNFC are marked `pub`. The portion denoted by ② declares the Rust type representation for values of the non-terminal, such that the non-terminal `Exp` is represented by the type `abs::Exp<'a, T>` of the abstract syntax.

The portions denoted by ③ correspond to the right-hand sides of the production rules pertaining to the non-terminal in question. Thus, `"let" <Ident> "=" <Exp> "in" <Exp>` is the equivalent of the production rule labelled `ELet` in the LBNF specification declared in Grammar 11. The portion denoted by ④ is the action that is to be executed in order to construct the Rust value representation for the production rule in question, the type of which must be the same as that denoted by ②. In this case, the function `abs::Exp::newELet` function is called. The first argument to this function is the bump allocator `arena`, specified as an argument to the parser by the grammar declaration above. Following this, the Rust representations of all items enclosed within angle brackets in left-hand side of the arrow, viz. non-terminals in the corresponding production rule, are passed as arguments to this function as well. Note, however, that no action has been explicitly defined for the second production rule `Exp1`, in which case its Rust representation, the type of which is identical to that denoted by ②, is used as-is; `Exp1` is a short-hand for `<Exp1> => <>`.

```
① pub Exp: ② abs::Exp<'a, T> = {
  ③ "let" <Ident> "=" <Exp> "in" <Exp> => ④ abs::Exp::newELet(arena, <>),
  ③ Exp1,
}
```

Listing 22 : The anatomy of a non-terminal definition in LALRPOP, in which ① concerns its visibility and name, ② the data type of its Rust representation, ③ the right-hand sides of its production rules, and ④ the Rust structural representation of the corresponding production rule

External lexer specification

LALRPOP has its own lexer generator, which generates token definitions based on what is used in the non-terminal definitions, similar to how tokens are implicitly defined in LBNF. However, to allow for higher control of the lexer generator, an external lexer may be used,

which is then linked to the LALRPOP grammar by an external lexer specification. This is the way that the Rust back-end functions; it uses the Logos crate for the purpose of lexing, as earlier described.

The external lexer specification consists of an `extern { ... }` block, containing three items: a type used to specify locations within the token stream, a type used for erroneous tokens, and an enumeration of the tokens themselves. Regarding the BNFC back-end, there is no notion of location information for non-terminals themselves, only for specific tokens. Thus, no location information is provided. The type used for erroneous tokens has been described earlier. The enumeration of tokens is a mapping from the names used in the rest of the LALRPOP grammar file, to the names used in the Logos lexer specification. For instance, `"+" => lexer::Token::SYM_plus` indicates to LALRPOP that whenever `"+"` is encountered in a production rule, it corresponds to the token `SYM_plus`. For annotated tokens, the type of the data is specified, such as `f64` in `"Double" => lexer::Token::Double(<f64>)`.

4.3.4 Linearizer

There are two canonical ways to produce a textual representation of the generated AST. The first way is to simply provide a tree representation, where each AST node is described with its constructor and the nodes that it contains. The second way is to *linearize* the AST into a textual representation. The term linearization denotes the conversion of a value from a structured tree representation to a flat and linear, non-recursive representation. Linearizing an AST involves utilizing the original grammar to produce a string, with identical AST representation. Informally, this process may be though of as reconstructing the original character sequence, as fed to the lexer. Note that there may be multiple possible sentences which yield the same abstract syntax tree, for instance `1`, `((1))` and `( ( 1))` in many common expression grammars.

The Rust back-end produces a linearizer, which is utilized in implementations of the `Display` trait for each type of the AST. The `Display` trait is the preferred way in Rust to attach a function producing a human-readable textual representation of a type. The `Debug` trait, used for attaching alternate textual representations used for debugging, is auto-generated for each AST type, and produces the simpler tree-based representation.

4.3.5 Parse-and-print program

When compiling an LBNF grammar using the Rust back-end, the source code of a Rust program, called a *parse-and-print program* (PPP), is generated, also known as ‘test parsers’ in the context of other back-ends. The purpose of the PPP is firstly to facilitate testing of the generated parser and secondly to provide the user with a concrete example on how to incorporate the generated parser into a larger codebase. An extract of a generated PPP is seen in Listing 23.

```
fn parse_and_print(s: &str) {
  let start = Instant::now();
  let alloc = Bump::new();
  let (parsed, parse_time): (Program<()>, Instant) =
      match ProgramParser::new().parse(&alloc, Token::lexer_for_lalrpop(s)) {
          Ok(g) => (g, Instant::now()),
          Err(e) => {
              eprintln!("Error during parsing: {e:?}");
              println!("Took {}ms", start.elapsed().as_millis());
              exit(-1);
          }
      };
  ...
}
```

Listing 23 : An extract from a generated Rust parse-and-print program

For the purpose of building the project and running the PPP, the Cargo command `cargo run` may be used. Running the PPP with a file path as its argument, or providing input to the standard input, will parse the input according to the first defined entry point in the LBNF specification, print the AST according to the linearizer and the `Debug` trait, as well as a measurement on the time take to perform the parsing and printing.

5

Evaluation

The three objectives, presented in Section 1.2 are evaluated in this chapter. Objective **O1**, which regards the back-end compatibility check and interface (cf. Chapter 3), is evaluated in Section 5.1. Objective **O2**, which regards the encapsulation and decoupling of the BNFC codebase, is evaluated in Section 5.2. Objective **O3**, which regards the Rust back-end (cf. Chapter 4), is evaluated in Section 5.3.

5.1 The back-end compatibility check and interface

> **O1.** *A back-end compatibility check shall be developed for the BNF Converter.*

The back-end compatibility check and interface presented in Chapter 3 shall be evaluated, both from a theoretical point of view as well as from a practical point of view.

The evaluation from the theoretical point of view will be based on the properties defined in Section 1.2, and is presented in Sections 5.1.1 through 5.1.5.

The evaluation from the practical point of view will be based upon the efforts made to convert the existing back-ends to the interface, presented in Section 5.1.6, as well as the development of the new Rust back-end (cf. Chapter 4) and future back-ends, for which the impact of the interface is discussed in Section 5.1.7. Additionally, this evaluation will encompass the experience of using BNFC as a tool, and how the new back-end compatibility check and interface affects that.

By the motivations provided in this section, the objective **O1** is considered well fulfilled.

5.1.1 Confinement of input validation to the front-end

> **P1.** *The source code relating to the execution of the back-end compatibility check shall be located in the code of the front-end; no further validation shall be performed by the back-ends.*

Property **P1** dictates that the source code relating to the execution of the back-end compatibility check shall be located within that of the front-end. This has been accomplished by devising a check, augmenting the LBNF validation pipeline in the front-end as described in

Chapter 3, that operates over an abstraction compatible with all back-ends. The identified differences in nature between the various back-ends are represented and conveyed by means of the interface to the logic performing the check. The check is thus fully performed in the front-end; the back-ends may merely provide a configuration specifying *how* the check is to be performed, as stipulated by property **P2**. The mechanisms of the interface providing the means for this configuration are the Features API, Rules DSL, and default list recursion. The manner in which these mechanisms may influence the back-end compatibility check and the exhaustiveness of which, with respect to the configurations expressible by these mechanisms, are discussed in Section 5.1.3.

Certain mechanisms of the back-end compatibility check, such as setting the preferred type of recursion to be employed by the list pragmas, are specified by the provision of simple values. Nevertheless, the interface is sufficiently expressive as to in some positions allow for advanced, stateful computations, to retrieve certain configuration values. These computations are defined and controlled by the back-end implementation. Even though these computations are invoked by the front-end, execution of arbitrary back-end logic may nevertheless occur during the back-end compatibility check. An example of a definition for a stateful computation to be employed for the purpose of retrieving a configuration value, namely that of a feature flag, is demonstrated in Listing 24.

```
data FooOptions = FooOpts
  { enablePosTok
  , forbidPosTok
  }

instance Backend Foo where
  type BackendState Foo = State
  type BackendOptions Foo = FooOptions
  features = positionTokensIf enPosTok
               "enablePosTok && !forbidPosTok"

enPosTok :: BackendExec Foo Bool
enPosTok = do
  modify fn1 -- Arbitrary modification of state
  -- Get value of "forbidPosTok" from command line
  forbid <- gets envBackendOpts <&> forbidPosTok
  if forbid
    then pure False
    else do
      modify fn2 -- More arbitrary modification of state
      -- Return value of "enablePosTok"
      gets envBackendOpts <&> enablePosTok

fn1, fn2 :: State -> State
```

Listing 24 : Using a stateful computation to retrieve a configuration value

At first glance, this may appear to violate property **P1**, inasmuch as the back-ends are not intended to conduct any validation. Note, however, that these computations are not a part of the *validation* itself. Rather, they serve as means to configure the validation process.

On account of the nature of the back-end compatibility check and the interface described herein, the property **P1** is to be considered fulfilled.

5.1.2 Back-end specific control through the interface

> **P2.** *The behaviour of the back-end compatibility check shall be controlled by a designated back-end through the use of an* interface.

The Features API, Rules DSL, and default list recursion, which are part of the interface, provide the necessary mechanisms by which the back-end compatibility check can be adapted to the requirements of each back-end, in accordance with property **P2**. This is described in greater detail in Section 5.1.1.

5.1.3 Exhaustiveness of the back-end compatibility check

> **P3 (a).** *The back-end compatibility check shall be sufficiently expressive that no further validation would be required for any input successfully passing the check.*

A way in which property **P3 (a)** could be fulfilled is by allowing for the greatest amount of generality so that a back-end may impose any conceivable requirements. Clearly, this could be implemented in the interface through an arbitrary predicate function that maps the otherwise validated LBNF specification and the command-line options to a boolean value: `checkBackend :: (LBNF, Options) -> Bool`. However, this level of generality is evidently against the principle stipulated by property **P1**, forasmuch as the logic pertaining to the back-end compatibility check would be fully contained within this function; although the check would be conveyed through the interface and eventually executed by the front-end, it would nevertheless be completely defined by each back-end, repeatedly and separately. This would have the further ramification that there be no mechanism to enforce consistency and uniformity, as required by property **P4**. A more restrictive interface is therefore justified.

The developed interface is indeed more restrictive. It merely provides means by which the back-end compatibility check can be *configured*, so as to fit back-end-specific needs; the check proper is fully conducted by the logic defined in the front-end codebase, as stipulated by property **P1**. These mechanism are, as previously noted, the Features API, the Rules DSL, and default list recursion.

The Rules DSL, presented in Section 3.3.2, allows expressing various rules that either accepts, rejects, or modifies a string. The Pattern DSL is designed such that the most commonly-used patterns can be expressed without using arbitrary functions, which is supported by the case

study. However, recognizing names as part of an arbitrary language, for instance one defined by an unrestricted grammar, is the task of a Turing machine.

Clearly, all aspects of the DSL can be expressed by means of arbitrary functions. It is, however, desirable to retain a DSL, which is judged to provide sufficient means for most use cases, on account of its benefits: optimization and introspection, the latter of which is of importance to property **P4**. Thus, use of the Rules DSL is preferred to that of arbitrary functions.

As can be seen from the interface definition, the Rules supplied by the back-end must be a pure value and may not be the result of a `BackendExec` monadic computation. This is a technical limitation, due to the way the `BackendExec` monad is defined; it allows access to the intermediate LBNF representation, in addition to the back-end state and the command-line options. However, no access should be given to the LBNF representation before the names have been validated; it would otherwise be possible for a back-end to base its internal state on the original names of the LBNF specification, rather than the validated names. In order to prohibit this, the rules must be retrievable without any dependence on the LBNF specification; they may not be the result of a computation in the `BackendExec` monad.

This may be seen as a limitation, since names may not be dependent on command-line options. However, when developing a back-end where there are different reserved names based upon the command-line options provided, one can simply choose to reject all the names which are rejected by any of those options. The resulting rule may be more restrictive than desired, but the property **P3 (a)** is nevertheless upheld. In fact, this consideration is the reason that the Rules may not be dependent on just the command-line options, which would not be as problematic.

In addition to having an expressive Rules DSL, it is crucial that the Features declaration is expressive enough in order that the back-ends need not do further input validation. Declaring what features a back-end supports, which means choosing a combination of features that a back-end can handle, must be granular enough that all limitations of all back-ends can be accounted for. This may, at first, seem hard to achieve. However, the Features API is designed to be expandable, such that if a new back-end is added that does not support a feature that was previously assumed to be universal, a new feature flag can simply be added. This leads to a dynamic granularity where the features are only as granular as they need to be for the back-ends that are actually implemented.

One possible limitation is that an entirely new class of possible compatibility problems is discovered, that is not accounted for by either the Rules DSL or the Features API. This is certainly possible, and if such compatibility problems are discovered, the back-end compatibility check of this thesis does not suffice. There is still a way for back-ends to perform arbitrary checks; a back-end may throw an error during state initialization, at which point the back-end has access to both the LBNF structure and the command-line options. However, even though this is possible, such arbitrary checks performed in the back-end clearly violates **P3 (a)**.

5.1.4 Limitation of dependence on back-end internals

> **P3 (b).** *The back-end compatibility check shall be sufficiently expressive that no back-end specific knowledge, besides that specified through the interface, be required in order for the check to be performed in the front-end.*

Property **P3 (b)** dictates that the front-end should not need specific knowledge about any back-end other than what can be retrieved by way of the interface. This property is achieved on account of two design patterns employed in the BNFC project.

Firstly, the `bnfc-backends` package, which contains the code of the back-ends, only exports the types representing the back-ends and their type class instances, and nothing else. All items exposed from the back-ends are exposed through the interface, ensuring encapsulation.

Secondly, the code of the front-end is parameterized by a generic back-end, wherever possible. In some instances, such as for back-end specific command-line parsing, a list of concrete back-ends must be specified. This results in the inherent possibility that the behaviour of the check be defined differently for a specific back-end, which is undesirable on account of inconsistency. However, by limiting the use of back-end-specific code and, instead, by employing back-end-generic code, the risk of introducing back-end-specific behaviour in the front-end codebase is reduced. This is illustrated in Listing 25, in which the function `runBnfcGrammar`, which is provided with the user-specified back-end to be invoked through the value argument `BackendCommand`, immediately invokes the function `runSpecialized`, which is generic over the back-end by means of a type parameter[24].

```
data BackendCommand
  = CmdAgda    (BackendOptions Agda)
  | CmdC       (BackendOptions C)
  | CmdCpp     (BackendOptions Cpp)
  | CmdHaskell (BackendOptions Haskell)
  | ...

runBnfcGrammar :: Grammar -> CompileOptions -> BackendCommand -> CommandM CompileResult
runBnfcGrammar grammar compileOpts = \case
  CmdAgda    backendOpts -> runSpecialized @Agda    compileOpts grammar backendOpts
  CmdC       backendOpts -> runSpecialized @C       compileOpts grammar backendOpts
  CmdCpp     backendOpts -> runSpecialized @Cpp     compileOpts grammar backendOpts
  CmdHaskell backendOpts -> runSpecialized @Haskell compileOpts grammar backendOpts
  ...

runSpecialized :: forall b. Backend b =>
  CompileOptions -> Grammar -> BackendOptions b -> CommandM CompileResult
runSpecialized compileOpts grammar backendOpts = ...
```

Listing 25 : Back-end-generic code

[24] The *at sign* (`@`) denotes *type application*.

5.1.5 Uniformity of user communications

> **P4.** *The interface through which the back-end compatibility check is conducted shall enable a consistent and uniform means of communication to the user.*

In addition to validating an LBNF specification, the back-end compatibility check can be leveraged to improve the user experience by providing the user with informative messages, warnings and errors messages. Since the check is performed in the front-end, these messages can be generated in a uniform and consistent manner, similar to how such messages are generated by other compilers.

Listing 26 demonstrated the messages conveyed to the user as the result of invoking a back-end that, by means of the Rules DSL, either transforms or rejects a name from the user-provided LBNF specification. Note that precise information about which renamings and rejections occur is provided to the user. Furthermore, Listing 27 illustrates an error that is yielded, should an LBNF specification require a feature that is not supported by the chosen back-end.

```
Prog. Program ::= Exp                    ;
Self.   Exp   ::= "Self"                 ;
Block.  Exp   ::= "{" [Exp] "}"          ;
While.  Exp   ::= "while" "(" Exp ")" Exp ;
for.    Exp   ::=
    "for" "(" Exp ";" Exp ";" Exp ")" Exp ;
define for init cond post body = Block [
        init,
        While cond (Block [body, post])
] ;
```

```
$ bnfc3 rust grammar.cf
Checking grammar.cf
===========================================
Messages:
grammar.cf:5:1: The definition 'for' was
renamed to 'r#for'.
===========================================
Errors:
grammar.cf:2:1: The name 'Self' was rejected
by a rule.
===========================================
```

Listing 26 : An LBNF specification rejected by the Rust back-end, on account of the use of a reserved keyword as the name for a label

```
Program. Prog ::= MyToken     ;
position token MyToken char + ;
```

```
$ bnfc3 cpp-no-stl grammar.cf
Checking grammar.cf
===========================================
Errors:
grammar.cf: The specified backend does not
support positional tokens, which is required by
this grammar.
===========================================
```

Listing 27 : The error message resulting from the invocation of the C++ 'No STL' back-end on an LBNF specification utilizing an unsupported feature: positional tokens

In the currently released version and in that presented by Vergani, the user may choose to perform validation on an LBNF specification without any back-end being invoked, so as to display information and error messages relating to the validation process. In both these versions, this was implemented through a special back-end named 'Check', which did not

produce any output files. The behaviour exhibited by this does not conform to the general concept and characteristics to which the other back-ends adhered.

In order to accommodate the demand of behaviour not compliant with that of ordinary back-ends, such as that provided by the Check pseudo-back-end, the notion of *auxiliary commands* was introduced to BNFC. Indeed, an auxiliary command `check` providing the functionality of the previously mentioned pseudo-back-end was created. Furthermore, so as to more comprehensibly relate the capabilities of the various available back-ends to the user, another auxiliary command, `bnfc3 list`, was introduced. The name, description, and main function of each back-end, obtained from the manifest, are displayed by this command. Likewise are the features supported by each back-end, as declared through the Features API, in the form of a table. Furthermore, should the option `--rules` be supplied, a description of the rules specified by means of the Rules DSL is provided for each back-end. The output of this command is demonstrated by Listing 28.

```
$ bnfc3 list --rules
Agda (`bnfc agda`):         Lexer/Parser  - Output Haskell code for use with Alex and
Happy together with Agda bindings for AST, parser and printer
 - Category rule: Modifying the names 'as' 'case' ... 'type' 'where' by appending '''
 - Label rule: ...
 - Define rule: ...
 - Argument rule: ...
C (`bnfc c`):               Lexer/Parser  - Output C code
C++ (`bnfc cpp`):           Lexer/Parser  - Output C++ code
Haskell (`bnfc haskell`):   Lexer/Parser  - Output Haskell code for use with Alex and
Happy
Java (`bnfc java`):         Lexer/Parser  - Output Java code
LaTeX (`bnfc latex`):       Documentation - Output LaTeX code to generate a PDF
description of the language
OCaml (`bnfc ocaml`):       Lexer/Parser  - Output OCaml code
Rust (`bnfc rust`):         Lexer/Parser  - Output Rust code for use with Logos and
LALRPOP
Txt2Tags (`bnfc txt2tags`): Documentation - Output a text file to feed to txt2tags
```

	Agda	C	C++	Haskell	Java	LaTeX	OCaml	Rust	Txt2Tags
Colliding Cat/Label	✔	✖	✖	✔	✖	✔	✖	✔	✔
Layout syntax	✔	✖	✖	✔	✖	✔	✖	(1)	✔
Makefile	✔	✖	✖	✔	✖	✔	✖	✖	✔
Position tokens	✔	✖	✖	✔	✖	✔	✖	(2)	✔
Regex difference	✖	✖	✖	✖	✖	✔	✖	✖	✔

```
(1): Not supporting top-level layout syntax
(2): Position in bytes
```

Listing 28 : The output of the `bnfc3 list --rules` command, in which some rules have been excluded for the sake of brevity

The addition of the back-end compatibility check allows for a more powerful notion of checking an LBNF specification; in addition to the auxiliary command `bnfc3 check` mentioned above, which only performs general validation and none relating to any specific back-end, viz. the back-end compatibility check, any back-end may be invoked with the option `--check-only`. Upon the invocation of a back-end with this option enabled, all validation, both general and specific to the chosen back-end, will be performed, but the back-end itself will not invoked. By this option, the validity and compatibility, with respect to a given back-end, of an LBNF specification can be determined.

5.1.6 Migration of existing back-ends to the new interface

The work on BNFC conducted as part of this thesis continues that of the unreleased version utilizing the interface developed by Vergani. That version of BNFC contains four functioning back-ends: one for Haskell; one for Agda, which heavily depends on that of Haskell; one for Latex; and one for Txt2Tags.

Previously, the codebase of the Haskell back-end included a list of reserved keywords, by which names were filtered or modified in the following ways.

1. The name of a type, constructor, or function, would be suffixed by an apostrophe character, if it were in the list of reserved words.
2. The name of a function argument would be suffixed by an underscore character, if it were in the list of reserved words.
3. The name of an argument, in the context of the generated output for the linearizer, is the name of its type, the first letter of which was converted to lowercase. This name would then be suffixed by an underscore character if it were in the list of reserved words.

Using the new interface, the modifications stipulated by 1 and 2 can be achieved by means of the Rules DSL, as exhibited in Listing 29. The four Rule declarations in the interface, `labelRule`, `categoryRule`, `defineRule`, and `argRule`, allow for application of different Rules for the respective names.

```
rule, argRule :: Rule
rule = R.modifying (R.setPattern (map pack hsReservedWords)) (R.Appending "'")
argRule = R.modifying (R.setPattern (map pack hsReservedWords)) (R.Appending "_")
```

Listing 29 : The two different `Rules` used in the Haskell back-end

The behaviour stipulated by 3, however, was deemed to be particular to the Haskell back-end and not generally applicable. Even though no general facility is provided for this purpose as part of the back-end compatibility check, it is nevertheless possible to employ the Rules DSL, including the use of already-defined Rules, in order to simplify this task. This is seen in Listing 30, where the function `toVarName`, providing the behaviour specified by 3, is redefined in terms of the Rules DSL, in particular using the rule `argRule`.

```haskell
toVarName :: String -> String
toVarName [] = []
toVarName (x:xs) =
  avoidReservedArg (toLower x : xs)

avoidReservedArg ::
  String -> String
avoidReservedArg x
  | x `elem` hsReservedWords = x ++
"_"
  | otherwise = x
```

(a) Previous variable generation and sanitization

```haskell
toVarName :: String -> String
toVarName [] = []
toVarName (x:xs) =
  fromJust $
    R.applyRule'
      argRule
      (toLower x : xs)
```

(b) New functions, leveraging a Rule defined in Listing 29

Listing 30 : Usage of the rule `argRule` for the transformation of names to those beginning with lowercase letters, potentially suffixing by an underscore character, outside of the interface

The Agda back-end utilizes several components from the Haskell back-end, including the lexer specification, parser specification, and abstract syntax, to which it produces bindings for Agda. For the reason that identifiers in Agda are governed by different rules from those in Haskell, this arrangement is not without its difficulties.

The previous Agda back-end firstly invoked the Haskell back-end, which applied certain naming rules, the intermediate result obtained from which were utilized with the aforementioned components. Secondly, another set of naming rules were applied to this intermediate result, namely the following.

1. All underscores are replaced by hyphens, in all positions.
2. The name of a type, function, or function argument would be suffixed by an apostrophe, if it is a reserved keyword.
3. The name of a label, the first letter of which is converted to lowercase, is suffixed by an apostrophe, if it is a reserved keyword.

The rules 1 and 2 are expressible by means of the Rules DSL, as demonstrated in Listing 31, but rule 3 is not.

```haskell
rule :: R.Rule
rule = R.Sequence
    (R.modifyingAll (R.replacing "_" "-"))
    (R.modifying (R.PSet agdaReservedWords) (R.Appending "'"))

constructorName :: String -> String
constructorName [] = []
constructorName (x : xs) = fromJust $ R.applyRule' rule (toLower x : xs)
```

Listing 31 : The rules of the Agda back-end described in the Rules DSL

The Agda back-end, as it was implemented, requires two versions of each name supplied by the intermediate LBNF representation, on account of the fact that the Haskell components used impose different restrictions on the names, as compared to the Agda-specific components. It is for this reason not possible to generate all names to be used in the output of the Agda back-end by means of the back-end compatibility check. Thus, the same rules employed by the Haskell back-end were also specified through the interface by the Agda back-end. In this way, the names to be used with the aforementioned components from the Haskell back-end are obtained by means of the interface through these rules. The names to be used with the Agda-specific components, on the other hand, are obtained by applying the appropriate rules within the code of the back-end.

In addition to naming rules, the new interface modified the task structure heavily. It was uncomplicated to adapt previous back-ends to the change done to the task structure, where each back-end previously needed to expose one function each for the tasks of producing a makefile, abstract syntax, linearizer, lexer specification, parser specification, and more. Since combining the previous functions requires checking the command-line options to decide if the makefile task is to be executed, a helper function was made to run a list of output-producing tasks, optionally appending the makefile task if needed, shown in Listing 32. Using this function, the code of the back-ends became much conciser, as seen in Listing 33. Firstly, it is much more clear what order the functions run, and secondly, for back-ends such as the Txt2Tags back-end, which did not use all of the previous functions, the implementation became much shorter.

```
-- | Concatenates one makefile job with multiple other jobs, if needed
concatMakefile
  :: forall a b.
     Backend b
  => BackendExec b [a]
  -- ^ The makefile job
  -> [BackendExec b [a]]
  -- ^ All other jobs, in order of execution
  -> BackendExec b [a]
  -- ^ The combined output of all jobs, possibly including the makefile job
concatMakefile makefileJob otherJobs = do
  initialResults <- sequence otherJobs <&> concat
  doMakefile <- asks envCompileOpts <&> optMakeFile
  if doMakefile
    then do
      makefileResult <- makefileJob
      pure $ initialResults ++ makefileResult
    else pure initialResults
```

Listing 32 : Function to adapt previous task structure to one large function

Lastly, the `BackendExec` monad, presented in Section 3.1.3, further simplified the code for the back-ends in multiple areas. One such area is the persistent access to `BackendEnv`, which contains the compilation options, the back-end specific options, and the LBNF grammar. When inside the `BackendExec` monad, the back-end may thus at any time retrieve those options, which was not possible before. In the earlier versions of the back-ends, the compilation options

and back-end specific options were only available when initializing state, and thus the back-end developers needed to explicitly store these options in the state, to be able to retrieve them later, seen in Listing 34. This is no longer needed, which simplifies the state of every back-end, and in the case of the Txt2Tags back-end which did not have any other state, the state type and state initializer could be removed completely.

```
instance Backend Agda where
  ...
  abstractSyntax :: BackendExec Agda
FileOutputs
  abstractSyntax = agdaAbstractSyntax
  printer :: BackendExec Agda FileOutputs
  printer = agdaPrinter
  lexer :: BackendExec Agda FileOutputs
  lexer = agdaLexer
  parser :: BackendExec Agda FileOutputs
  parser = agdaParser
  parserTest :: BackendExec Agda FileOutputs
  parserTest = agdaParserTest
  makefile :: BackendExec Agda FileOutputs
  makefile = agdaMakefile
```

(a) Previous version of the Agda back-end

```
instance Backend Agda where
  ...
  runBackend :: BackendExec Agda FileOutputs
  runBackend =
    concatMakefile
      agdaMakefile
      [ agdaLexer
      , agdaParser
      , agdaParserTest
      , agdaAbstractSyntax
      , agdaPrinter
      ]
```

(b) New version of the Agda back-end

```
instance Backend Txt2Tags where
  ...
  abstractSyntax :: BackendExec Txt2Tags
FileOutputs
  abstractSyntax = txt2tags
  printer :: BackendExec Txt2Tags FileOutputs
  printer = return []
  lexer :: BackendExec Txt2Tags FileOutputs
  lexer = return []
  parser :: BackendExec Txt2Tags FileOutputs
  parser = return []
  parserTest :: BackendExec Txt2Tags
FileOutputs
  parserTest = return []
  makefile :: BackendExec Txt2Tags FileOutputs
  makefile = txt2tagsmakefile
```

(c) Previous version of the Txt2Tags back-end

```
instance Backend Txt2Tags where
  ...
  runBackend :: BackendExec Txt2Tags
FileOutputs
  runBackend = concatMakefile txt2tagsmakefile
[txt2tags]
```

(d) New version of the Txt2Tags back-end

Listing 33 : Migrating back-ends to utilize one single entry point in contrast to multiple

```
data HaskellBackendState = HaskellSt
  { compileOpt :: CompileOptions
  , haskellOpts :: HaskellBackendOptions
  , lexerParserTokens :: [Token]
  , astRules :: AST
  , parserRules :: [ParserRuleGroup]
  , functions :: [(LabelName, Function)]
  , tokens :: [(CatName, TokenDef)]
  }
```

(a) Previous Haskell back-end state

```
data Txt2TagsBackendState = Txt2TagsSt
  { compileOpt :: CompileOptions
  , txtOpts :: Txt2TagsBackendOptions
  }
```

(b) Previous Txt2Tags back-end state

Listing 34 : Back-end state containing compile options and back-end options

5.1.7 Development of additional back-ends

A new back-end can more easily be added to BNFC using the interface presented in this thesis, as compared to previous versions. In what follows, a review of the aspects by which the development of additional back-ends for BNFC is expedited, as a consequence of the interface developed as part of this thesis, is presented.

The name of a back-end and its CLI command can easily be defined in the manifest, presented in Listing 3, of the respective back-end. Thus, no additional changes to the front-end codebase is necessary for this purpose. For instance, the manifest of the Rust back-end, which is to be invoked by the `bnfc3 rust` command, is defined as shown in Listing 36.

Names are often required to be changed or rejected on back-end specific grounds. This matter is readily resolved by use of the Rules DSL, introduced in Section 3.3.2. In the Rust back-end, names may be prepended by the escape sequence `r#` or unconditionally rejected, as is the case of `Self`, the matter of which is indeed facilitated by the Rules DSL. Moreover, it may also be employed within the back-end code to facilitate the generation of unique identifiers compliant to the various restrictions imposed in the output files. This is illustrated in Listing 30, in which the Rules DSL is leveraged to transform category names to lowercase identifiers, to which an underscore character is appended, for use as pattern bindings[25] in case branches in the generated code of the Haskell back-end.

The interface utilized by the back-end compatibility check, as opposed to the previous one presented in Section 2.5.2, makes no assumptions regarding the relationship between tasks and output files. Although the previous interface did allow for generation of output files dependent on multiple tasks, by means of stateful monadic computations, the imposed task to output file relationship was nevertheless excessively strict and based on false assumptions, as demonstrated in Section 3.5.2. As such, this potential impediment to the development of additional back-ends was removed from the interface developed as part of this thesis. The Rust back-end, which generates a Cargo crate as its output, is required to produce a manifest file that is dependent on various tasks, such as that of generating a lexer, parser, and abstract

[25] In a case expression such as `case listToMaybe xs of Just x -> ...`, the variable `x` in the pattern `Just x` is bound as part of a case branch.

syntax. Therefore, the generation of this file must occur after all other tasks, so as to collect all dependencies. The implementation of this task was facilitated by the new interface, since the back-end itself is in control of the evaluation order of tasks.

The previous interface also provided a means by which a parser for back-end-specific command-line options could be defined. This capability, concluded to be of further use, was retained in the new interface.

Furnished with the ability to dictate the default type of recursion, with respect to the list pragmas of LBNF, a back-end may specify the behaviour most appropriate to the targeted programming language and parser generator, in accordance with the principles outlined in Section 3.4. As for the Rust back-end, this is of great benefit inasmuch as it facilitates the usage and improves the performance of the dynamic array from the standard library, for which left-recursive grammatical structures are more efficient. This would previously not have been possible, on account of the fact that use of right-recursion was prescribed.

The use of optional features of LBNF in a back-end can be indicated to the front-end by means of the Features API, introduced in Section 3.6.1. Should it be desired, in the course of creating a new back-end, additional capabilities to LBNF can be incorporated without the need to modify any existing back-end. The Rust back-end utilizes the Features API in order to indicate its support for duplicate names – that the same identifier may be used for a category and a rule label. It does, however, not support layout syntax, the matter of which is conveyed to the front-end by the very absence of the corresponding field from the declaration of supported features.

In order to add a new back-end to BNFC, it must be registered as such in the `bnfc3-executable` package; additional source code is to be inserted at a total of five locations within the package, so as to link the instance of the back-end interface to the appropriate CLI commands, e.g. `bnfc3 rust` and `bnfc3 list`. For this purpose, only trivial changes need to be made, such as adding a constructor to a data type, as seen in Listing 35, and no additional knowledge of the code in the package is necessary.

```
data BackendCommand
  = CmdAgda     (BackendOptions Agda)
  | CmdC        (BackendOptions C)
  | CmdCpp      (BackendOptions Cpp)
  | CmdHaskell  (BackendOptions Haskell)
  | CmdLatex    (BackendOptions Latex)
  | CmdJava     (BackendOptions Java)
  | CmdOCaml    (BackendOptions OCaml)
  | CmdTxt2Tags (BackendOptions Txt2Tags)
  | CmdRust     (BackendOptions Rust)
```

Listing 35 : An example of a data type definition that requires an additional constructor, should a new back-end be added to BNFC

Nevertheless, it would be preferable if no modifications at all were required to be made in order to add a back-end to BNFC. One way of accomplishing this is through the use of *Template Haskell* [68], a system providing compile-time *meta-programming*[26] for Haskell. By leveraging the powers of Template Haskell, the aforementioned required additions to the source code can be automatically generated at compile-time, thus eliminating the need for any alteration of source code outside of the `bnfc3-backends` package. A solution employing Template Haskell in this manner was successfully implemented, but it was, however, ultimately decided against, on account of perceived difficulties in comprehension of the Template Haskell code and its apparent lack of documentation; there was a concern that the rather obscure nature of Template Haskell could induce difficulties in the maintenance of the BNFC codebase.

5.2 Encapsulation and decoupling

> **O2.** *The codebase of BNFC shall be restructured, so as to further its encapsulation and to limit its coupling.*

Objective **O2** concerns the increased encapsulation and decoupling of the BNFC codebase. Increasing encapsulation in a project, such as by dividing its codebase into separate packages, is not merely an exercise to grouping parts of code. If executed well, encapsulation allows the constituent units to be developed independently of each other.

As presented in Section 3.2, the BNFC codebase was split into three packages: `bnfc3-backends`, `bnfc3-executable`, and `bnfc3-lib`. The `bnfc3-backends` package only exposes the types representing the individual back-ends. The number of components exposed from this package has thus been greatly constrained, even though it comprises over 10,000 lines of code. Furthermore, the front-end codebase does not directly depend on, nor can it access, the internals of the back-end codebase. Consequently, the source code of the back-ends can readily be altered, should any back-end require future modification, provided that the name of the exported type implementing the interface is retained.

Additional back-ends may also be added with relative ease, the process of which is described in Section 5.1.7. With all these facts in mind, the `bnfc3-backends` package is well-structured and contributes to the overall encapsulation of the project as a whole.

The `bnfc3-executable` package contains the code pertaining to the command-line interface of BNFC. Most of the code contained within concerns the declaration of the sub-commands of the `bnfc3` command-line application. This package can be used as a library that exposes the entry point function of the executable, so as to facilitate testing. This may be used by third party projects to invoke the executable. However, the main purpose of the package is to be compiled directly into an executable program. Since this package has no dependents, apart from the tests, it may be developed independently from the rest of the project. In addition to it externally being well encapsulated, it does internally define common structures, so as

[26] The term denotes the generation and manipulation of source code, or equivalent construct, of a programming language.

```
instance Backend Rust where
  type BackendOptions Rust = RustBackendOptions
  type BackendState Rust = RustBackendState

  manifest = Manifest
      { name        = "Rust"
      , commandName = "rust"
      , description = "Output Rust code for use with Logos and LALRPOP"
      , kind        = [Parser, Lexer]
      }

  labelRule    = upperRule
  defineRule   = lowerRule
  categoryRule = upperRule
  argsRule     = lowerRule

  parseOpts :: Parser RustBackendOptions
  parseOpts = rustOptionsParser

  initState :: BackendEnv Rust -> Except String (BackendState Rust)
  initState = rustInitState

  runBackend :: BackendExec Rust FileOutputs
  runBackend = runRust

  features = supportsDuplicateNames <>
               positionTokensIf (pure True) "Position in bytes"

  defaultListRecursion = LeftRecursion

-- | The rule lowercase names must adhear to
lowerRule :: Rule
lowerRule = Rule.Sequence
      (Rule.rejecting lowerForbidden)
      (Rule.modifying lowerKeywords (Rule.Prepending "r#"))

-- | Lowercase identifiers forbidden in Rust
lowerForbidden :: Rule.Pattern
lowerForbidden = Rule.setPattern ["crate", "self", "super"]

-- | Lowercase identifiers which are reserved but allowed as raw identifiers
lowerKeywords :: Rule.Pattern
lowerKeywords =
  Rule.setPattern
    [ "as"
    , "break"
    , "const"
    , "continue"
    , "else"
    , ...
    ]
```

Listing 36 : The `Backend` instance for the Rust back-end

to facilitate the development of new and the modification of existing sub-commands without breaking the functionality of others.

The `bnfc3-lib` package is a library containing various utilities relating to the LBNF format, such as those pertaining to lexical, syntactic, and semantic analysis of LBNF; data types for the intermediate representation of LBNF; and the back-end interface. This library is crucial to the development of back-ends and for a user-facing tool to interact with such back-ends. A common pattern employed in Haskell libraries is to expose what is regarded as 'internal' or 'private' components, thus providing the user of the library with all necessary tools, so as not to limit its utility. This principle has been retained, which should facilitate the development of other projects that are intended to interact with LBNF.

It is to be noted, however, that the exposure of internal components in this way impedes the aim of encapsulation; the exposure of an internal component encourages higher coupling. Furthermore, were the dependence on internal components to increase, potential future modifications to the package may become more laborious. Internal components of a library, as opposed to those comprising its public API, are generally subject to future breaking changes. Therefore, it may be of interest, in future additions to BNFC, to move internal components into a separate `BNFC.Internal` module, following the convention used by many Haskell packages, which is exemplified by the package `bytestring` [69]. In doing so, the status of these components is explicitly communicated.

The components that provide the means for interacting with the LBNF formalism, viz. lexing, parsing, and validation, as well as the component responsible for invoking a back-end can all be altered without any of their dependent components requiring modifications of their own. These components of the `bnfc3-lib` package are thus well encapsulated.

The interface of the back-end compatibility check also contributes to the decoupling of the BNFC codebase. In previous versions of BNFC, the name, CLI command, and description of each back-end were defined within the codebase of the front-end. This circumstance contributed to increased coupling of BNFC. By virtue of the manifest, these are instead defined through the new interface, thus decreasing the coupling between the front-end and back-end components.

In conclusion, for the reasons stated herein, the objective **O2** is to be considered well fulfilled.

5.3 The Rust back-end

> **O3.** *A back-end for BNFC targeting the Rust programming language shall be developed.*

A BNFC back-end targeting Rust has been developed, as described in Chapter 4. The back-end was developed as a means to evaluate the back-end compatibility check, but it is also subject to evaluation itself. The Rust back-end constitutes the fourth functioning back-end for the new version of BNFC, along with those of Haskell, Agda, Txt2Tags, and Latex.

The size of the Rust back-end, in terms of its codebase, is on the high end compared to other back-ends, as demonstrated in Table 4. Moreover, it supports most of the LBNF specification.

Table 4 : Lines of code for the back-ends of BNFC (non-exhaustive)

Old version of BNFC as of 22 November 2024	
Back-end	**Lines of code**
Haskell	3146
$\text{Haskell}_{\text{GADT}}$	407
Agda*	1007
OCaml	1448
Java	3092
C	2967
C++†	2179
Python§	1663
Latex¶	323
Txt2Tag [sic]¶	236

New version of BNFC as of 18 May 2025	
Back-end	**Lines of code**
Haskell‡	5046
Agda*	1789
Rust	3351
Latex¶	455
Txt2Tags¶	399

* The Agda back-end utilizes large parts of the Haskell back-end and thus uses more code than what is contained within the Agda back-end itself.
† The C++ back-end utilizes the same entry point as the C back-end, and thus share some code not counted
‡ The Haskell back-end in the new version of BNFC encompasses the functionality of the previous two back-ends Haskell and $\text{Haskell}_{\text{GADT}}$
§ This back-end is not yet merged, so the preview version was used
¶ This back-end generates documentation describing the input grammar, rather than a parser, lexer, and abstract syntax specification.

It does allow the developer to choose between multiple integer and floating-point representations, as well as allocation strategies, which all taken together, manifests it as an advanced back-end. However, one limitation with the parser generator used is that it does not allow shift-reduce conflicts in the input grammar. This may be accepted by certain other back-ends.

The Rust back-end has been tested using grammars provided as example grammars for the BNFC project. These include a grammar of the C programming language that exhibits shift-reduce conflicts due to the *dangling else* problem [70]. A new version of this grammar was developed that solved these shift-reduce conflicts. It was then used to produce a lexer, parser, printer, and *parse-and-print program* (PPP) using the Rust back-end. The output files compiled, the resulting program was used to successfully parse a C source file. The file chosen for this purpose was `core.c` of the task scheduler of the Linux kernel scheduler [71], which comprises 7,752 lines of code. Moreover, it was possible to parse the linearized output again, and, consequently, the back-end passes *round-trip testing*.

This modified C grammar was also used for the purpose of benchmarking. The respective generated Rust and Haskell files having been compiled into executable programs, the time taken to read a C source file, parse its contents, and print both a debug representation and its linearized output was measured using the benchmarking tool Hyperfine [72]. In addition to using the aforementioned `core.c` file, two additional variants of this file were used for this

purpose: `core25.c` and `core125.c`, which are composed by the contents of `core.c` concatenated with itself 25 and 125 times, respectively.

This was done in order to compare the performance of code generated by the two back-ends with respect to inputs of varying scale. The Haskell back-end was chosen as the subject of comparison to the Rust back-end on account of it being the most advanced[27] back-end. It was also attempted to employ the C back-end, from version 2 of BNFC, for the same purpose. However, the generated parser was not able to parse the aforementioned source file.

The results of the aforementioned benchmarks are presented in Table 5, and it was found that the generated Rust program exhibits a performance improvement, with respect to runtime, by a factor of approximately 17 compared to those generated by the Haskell back-end for the smallest file, `core.c`. Furthermore, it was found that runtime improvement factor increased for larger inputs; the improvement factors for the `core25.c` and `core125.c` were respectively measured to approximately 21 and 24. It is to be noted that between 60% and 65% of the execution time of that measured for the generated Rust code was spent on linearization of the parsed input, a use case atypical of compilers. In addition, it was also found that the runtime of the generated Rust program employing the bump allocation strategy was reduced by 11-18% relative to using the default general-purpose allocator.

Table 5 : A comparison of the runtime of the PPPs, generated by the Rust and Haskell back-ends, for input files of various sizes

File	Back-end	Runtime	Mean	**Speed factor** Rust[B] vs Haskell
`core.c` 7,752 lines 183 KiB	Haskell	125.6 to 130.3 ms	129.4 ms (σ 1.1 ms)	16.64 (±0.86)
	Rust[B]	7.1 to 8.9 ms	**7.8 ms** (σ 0.4 ms)	
	Rust[GP]	7.9 to 10.0 ms	8.6 ms (σ 0.4 ms)	
`core25.c` 193,801 lines 4.5 MiB	Haskell	3.182 to 4.073 s	3.383 s (σ 0.288 s)	20.51 (±3.22)
	Rust[B]	151.7 to 212.9 ms	**164.9 ms** (σ 21.7 ms)	
	Rust[GP]	174.0 to 268.0 ms	195.0 ms (σ 27.5 ms)	
`core125.c` 969,001 lines 22 MiB	Haskell	16.556 to 21.306 s	18.165 s (σ 1.462 s)	23.70 (±2.02)
	Rust[B]	756.9 to 826.8 ms	**766.3 ms** (σ 21.5 ms)	
	Rust[GP]	867.7 to 873.1 ms	870.3 ms (σ 1.8 ms)	
All benchmarks were conducted on an M1 MacBook Air with 8 cores and 16 GB of RAM, running on macOS 15.4.1. [B] The benchmark was conducted using the bump allocation strategy [GP] The benchmark was conducted using the default general-purpose allocation strategy				

These benchmarks were inspired by those conducted by Werner [26], which he also employed for benchmarking a new back-end. The obtained results suggest that the generated Rust code is significantly faster at parsing, traversing, and linearizing the resulting AST, compared to that of the Haskell back-end. The relative runtime difference may be dependent on the input

[27] In terms of feature support, size of codebase, and command-line options

file and LBNF specification used. Consequently, further benchmarking is necessary in order to draw more definite conclusions in this matter.

In conclusion, the Rust back-end provide most of the functionality that other back-ends provide, with an exception being the lack of support for layout syntax, which is only supported by the Haskell back-end. Additionally, the code generated by the Rust back-end provide the means to develop an efficient compiler front-end in a memory-safe programming language.

To this end, the objective **O3** is considered well fulfilled.

6

Conclusions

In this master's thesis, the development of a *back-end compatibility check* for the BNF Converter has been presented. Through this process, insights have emerged regarding the design of validation mechanisms for software projects that exhibit similar architectures to that of BNFC – namely, those comprising a single front-end and multiple back-end components, each imposing different requirements on its input.

If the set of back-end components should be subject to future additions, two opposing considerations may need to be reconciled in order to attain the most favourable outcome. On the one hand, the check should be conducted by means of an interface that allows for the greatest possible extensibility, with regard to any requirements that a future back-end may impose upon it. On the other hand, a highly-generalized interface exhibiting higher expressiveness may have disastrous consequences for optimization, introspection, a streamlined user experience, and maybe most crucially, the extendability of the check itself. It may thus be necessitated that a compromise between these considerations be made, a decision preferably made as the result of careful analysis and deliberation.

To this end, the implementation of the back-end compatibility check for BNFC was preceded by an inquiry set to ascertain which back-end components were likely and desired candidates for future additions, as well as the conditions that these candidates may come to impose on the checking mechanism and its interface. In this way, the check may be devised in accordance with the purpose of the project at large.

When applying this methodology to the BNFC project, multiple areas were identified, and different solutions were applied to the different areas. The problem of partial support of the LBNF specification was solved by the *Features API* (cf. Section 3.6), and the problem of rejection and transformation of names was solved by the *Rules DSL* (cf. Section 3.3.2). These problems of different natures required different solutions, but crucially, both of these adhered to the governing properties of the back-end compatibility check. Excessively general solutions to these problems – for instance, by means of arbitrary functions – would decrease the ability for optimization, introspection, and the expansibility of BNFC.

An additional back-end for the Rust programming language was conjointly developed in close association with the back-end compatibility check. This back-end made appropriate use of this check and stands as a testament to the benefits that a back-end compatibility check may provide. Moreover, the back-end by itself provides users opportunities to prototype fast, memory-safe parsers and lexers.

The portion of the project dedicated to coding was conducted by the two authors during a period of three months. The time taken to develop the Rust back-end was estimated to account for approximately half of that time. However, since the new interface, which reduces the minimum required complexity of a back-end, was developed conjointly with the Rust back-end, and since this back-end is more complex than what is required – for instance, it provides two AST representations – the minimum work required to develop a new back-end may be significantly lower than the work spent on the Rust back-end.

The methodology used to develop the back-end compatibility check was applied to the BNFC project, however the authors do believe that it may also be used successfully for other software projects inhabiting a similar structure. It is possible that some of the solutions leveraged for this project are broadly applicable, whilst others are more specific to BNFC. It is also clear to the authors that only the areas identified where back-ends may pose requirements were ensured to reach a satisfying result; this is a limitation inherent to this procedure. However, the advantages of carefully developing a fitting solution for each area, in terms of extendability, backwards compatibility, and user experience, are crucial to any large software project.

6.1 Future work

There exist future work, both on checking mechanisms in general, and in the BNFC project in particular. For the general case, work remains to prove (or disprove) if the methodology used in this project may be successfully applied to other projects as well. It might be the case that this methodology only works for some projects and not for others. Some of the solutions provided to particular problems that arose during the development of the back-end compatibility check may be formalized into solutions for general problems. Moreover, this methodology, or a more general variation of it, may be formalized into a procedure that could be applied more broadly.

Possible future work on the BNFC project includes developments closely related to what has been touched upon by this project, as well as more general developments. This section presents three areas which are directly enabled by this project, followed by three areas that relates to BNFC in general.

6.1.1 Development of additional facilities for BNFC

The restructuring of BNFC, which was initiated by Vergani [24] and continued in this project, allows for easier development of additional facilities for BNFC. This may include developing additional auxiliary commands to further improve the user experience. One such example could be a `lint` command, which checks a file for any possible compatibility issues. Furthermore, the usage of BNFC as a library may allow for developing editor integrations, such as a language server implementing the Language Server Protocol (LSP) [31], or meta-programming tools similar to the already-existing BNFC-meta [73].

6.1.2 Further improvements to naming rules

The Rules DSL developed as a part of this thesis is enough for what is needed in the current back-ends, in terms of what the back-end-compatibility check aims to do. However, since the Agda back-end replaces underscore characters with hyphens, it avoids the more exotic rule of the naming rules of the language; when splitting an identifier on underscores, it is not allowed for any such component to be empty. This rule is not expressible by the Rules DSL. Further research may continue exploring programming languages, their keywords, and improving the DSL.

Furthermore, it is common for back-ends to need to provide multiple versions of the same identifier but with different casing conventions. One example of such a need is discussed in Section 5.1.6 Migration, where the Haskell back-end creates new identifiers from existing types by converting their first character to lower-case. Similar situations occur in the Rust back-end.

Care must be taken when doing such conversions. Converting a set of unique identifier starting with uppercase letters by converting the first character to lowercase will produce a set of unique identifiers, but this is not guaranteed if the entire identifier is converted to uppercase or lowercase. For instance, if identifiers are to be converted to 'screaming kebab case' (e.g. `SCREAMING-KEBAB-CASE`), then `FOO_bar` and `FooBar` would both yield the same identifier, `FOO-BAR`.

One area worth exploring may be representing identifiers in LBNF as a list of case-insensitive *constituents*. Some examples of such in Table 6. Then, such constituents may be combined in multiple ways, where distinct input always produces non-colliding output. Furthermore, this is often what programming languages prefer to use anyways. Indeed, constituents are a part of the supporting library to BNFC, usable from within the back-ends. This idea may be explored further, which may include expanding the Rules DSL to filter and modify list of constituents.

Table 6 : Some identifiers and their respective constituents

Identifier	Constituents
`foo_bar`	`["foo", "bar"]`
`MyOwn_Ident`	`["my", "own", "ident"]`
`LbnfSpec`	`["lbnf", "spec"]`
`LBNFSpec`	`["lbnf", "spec"]`

Another area closely related to this is to have multiple distinct internal representations for identifiers in BNFC. This may be helpful in situations as seen when migrating the Agda back-end, where each identifier had two different names, one according to the Haskell naming rules and one according to the Agda naming rules. Development within this area may make it easier to develop other back-ends which produces bindings to other languages.

6.1.3 Back-end-tailored Features

The focus of this project was to develop general methods to account for back-end specific restrictions and limitations. The differences between the back-ends and the parser generators they employ may, however, allow for more efficient implementations of parts of the current or future LBNF specification. Tailoring features would allow some grammar specifications to be extra efficient when compiled to some back-ends.

A similar concept can be seen in some compilers, such as the Agda compiler [74]. The Agda compiler has multiple back-ends converting Agda code to other programming languages[28], and using special instructions, called *pragmas*, the compilation process may be customized dependent on which back-end is used. In BNFC, this may instead be realized by allowing back-ends to change certain parts of the LBNF pre-processing to better suite their needs. In part, list recursion achieves this in one specific instance, but further research is needed to achieve a solution general enough.

6.1.4 Expanding BNFC to new languages and paradigms

BNFC is in large parts only as powerful as its back-ends. Developing new back-ends targeting new programming languages or parser generators is a way of expanding the BNFC project. In addition to the plethora of parser generators that may be targeted by future back-ends, there is also a possibility to develop back-ends targeting languages of whole other paradigms. One example of such could be *Prolog*, which is a logic programming language, and in which parser generators are most often not needed, since the built-in syntax may be used to define a certain class of grammars [75]. Another example could be graphic programming languages, in which the resulting parser might be able to be modified easier than other parsers. Expanding the reach of BNFC to new paradigms may bring new possibilities and issues to light, and further research is needed to ensure that this is possible in the current state of BNFC.

6.1.5 Improvements to the general efficiency of BNFC

Not much focus has been given to the performance of BNFC itself; a low-hanging fruit to improving the performance may be to stream the output of a back-end rather than eagerly collecting the whole string to memory. There are certainly other areas of improvements as well, but further research into the efficiency of different components of BNFC is needed.

6.1.6 General enhancements to LBNF

The LBNF specification format may be amended with new pragmas and features to allow for developing more advanced back-ends. One area of potential improvement would be token definitions, where the built-in tokens currently cannot be defined in LBNF itself; it is impossible to do post-processing, such as parsing numbers, or trimming enclosing quotation marks

[28] Commonly referred to as *transpilation*

around strings, on tokens defined in LBNF. A general way to define tokens capturing anything else than the regular expression it is defined by may be researched in the future.

7

References

[1] C. Jaspan *et al.*, 'Advantages and Disadvantages of a Monolithic Codebase', in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[2] R. Morgan, *Building an optimizing compiler*. Digital Press, 1998, ISBN: 155558179X.

[3] D. Novillo, 'From Source to Binary: The Inner Workings of GCC', *Red Hat Magazine*, no. 2, Dec. 2004, [Online]. Available: https://web.archive.org/web/20160410185222/https://www.redhat.com/magazine/002dec04/features/gcc/

[4] N. Chomsky, 'Three models for the description of language', *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956, doi: 10.1109/TIT.1956.1056813.

[5] N. Chomsky, 'On certain formal properties of grammars', *Information and Control*, vol. 2, no. 2, pp. 137–167, 1959, doi: 10.1016/S0019-9958(59)90362-6.

[6] D. Quinlan, J. B. Wells, and F. Kamareddine, 'BNF-style notation as it is actually used', in *Intelligent Computer Mathematics: 12th International Conference, CICM 2019, Prague, Czech Republic, July 8–12, 2019, Proceedings 12*, Springer, 2019, pp. 187–204. doi: 10.1007/978-3-030-23250-4_13.

[7] 'Information technology — Programming languages — C', no. ISO/IEC 9899:202y, n3467 working draft. International Organization for Standardization, 2024.

[8] 'javac - The Java Compiler'. Oracle America, Inc., Java Platform, Standard Edition Tools Reference. Accessed: May 23, 2025. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javac.html

[9] T. Lindholm, F. Yellin, G. Bracha, A. Buckley, and D. Smith, 'The Java® Virtual Machine Specification', no. JSR-396 Java SE 21. Oracle America, Inc., Aug. 2023.

[10] M. Neumann, J. Ortiz, R. Feldt, and L. Johnson, *Ruby Developer's Guide*. Syngress Publishing, 2002, ISBN: 978-1928994640.

[11] 'parse, v., Etymology', *Oxford English Dictionary*. Oxford University Press, Sep. 2024. Accessed: Mar. 04, 2025. [Online]. Available: https://www.oed.com/dictionary/parse_v?tab=etymology

[12] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers principles, techniques & tools*. Pearson Education, 2007, ISBN: 0-321-48681-1.

[13] A. Amarilli, L. Jachiet, M. Muñoz, and C. Riveros, 'Efficient enumeration for annotated grammars', in *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, in PODS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 291–300. doi: 10.1145/3517804.3526232.

[14] D. E. Knuth, 'On the translation of languages from left to right', *Information and Control*, vol. 8, no. 6, pp. 607–639, 1965, doi: 10.1016/S0019-9958(65)90426-2.

[15] J. C. Beatty, 'On the relationship between LL (1) and LR (1) grammars', *Journal of the ACM (JACM)*, vol. 29, no. 4, pp. 1007–1022, 1982, doi: 10.1145/322344.322350.

[16] T. Anderson, J. Eve, and J. J. Horning, 'Efficient LR (1) parsers', *Acta Informatica*, vol. 2, pp. 12–39, 1973.

[17] F. L. DeRemer, 'Practical translators for LR (k) languages.', Massachusetts Institute of Technology, Cambridge, MA, USA, 1969. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/888578

[18] S. C. Johnson, *Yacc: Yet another compiler-compiler*, vol. 32. Murray Hill, NJ, USA: Bell Laboratories, 1975.

[19] A. Gill, S. Marlow, and other contributors, 'happy'. [Online]. Available: https://hackage.haskell.org/package/happy

[20] Z. Gan, 'YACC Calculator'. [Online]. Available: https://github.com/jerrymakesjelly/yacc-calculator

[21] M. Forsberg and A. Ranta, 'The Labelled BNF Grammar Formalism', Gothenburg, Sweden, Feb. 2005.

[22] A. Ranta, *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012, ISBN: 1848900643.

[23] M. Pellauer, M. Forsberg, and A. Ranta, 'BNF Converter: Multilingual Front-End Generation from Labelled BNF Grammar', Gothenburg, Sweden, 2004.

[24] B. Vergani, 'A common backend API for the BNF Converter', Chalmers University of Technology, Gothenburg, Sweden, 2021.

[25] B. Burreau, 'Layout Syntax Support in the BNF Converter', Chalmers University of Technology, Gothenburg, Sweden, 2023. doi: 20.500.12380/307579.

[26] B. Werner, 'Python BNF Converter - Extending the languages supported by BNFC with Python'. Chalmers University of Technology, Gothenburg, Sweden, 2024. doi: 20.500.12380/309053.

[27] J. MacFarlane, A. Krewinkel, and J. Rosenthal, 'Pandoc'. [Online]. Available: https://github.com/jgm/pandoc

[28] P. Capriotti, 'optparse-applicative'. [Online]. Available: https://hackage.haskell.org/package/optparse-applicative

[29] P. Wadler, ‘Comprehending monads’, in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, in LFP '90. Nice, France: Association for Computing Machinery, 1990, pp. 61–78. doi: 10.1145/91556.91592.

[30] M. P. Jones, ‘Functional programming with overloading and higher-order polymorphism’, in *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24–30, 1995 Tutorial Text*, in Lecture Notes in Computer Science, vol. 925. 1995, pp. 97–136. doi: 10.1007/3-540-59451-5_4.

[31] ‘Language Server Protocol Specification’, no. 3.17. Microsoft, Inc., Oct. 2022. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[32] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães, ‘Giving Haskell a promotion’, in *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, in TLDI '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 53–66. doi: 10.1145/2103786.2103795.

[33] S. Marlow, ‘Haskell 2010 Language Report’. 2010. [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf

[34] X. Leroy *et al.*, ‘The OCaml system release 5.3’. Institut National de Recherche en Informatique et en Automatique (INRIA), 2025. [Online]. Available: https://ocaml.org/manual/5.3/ocaml-5.3-refman.pdf

[35] J. Gosling *et al.*, ‘The Java® language specification’, no. JSR-396 Java SE 21. Oracle America, Inc., Aug. 2023.

[36] S. Guo, M. Ficarra, and K. Gibbons, ‘ECMAScript® 2024 Language Specification’, no. ECMA-262, 15th edition. Ecma International, 2024.

[37] ‘The Python Language Reference’, no. 3.13.2. Python Software Foundation, 2025. [Online]. Available: https://docs.python.org/3.13/reference/index.html

[38] T. Rompf and M. Odersky, ‘Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs’, *Communications of the ACM*, vol. 55, no. 6, pp. 121–130, 2012, doi: 10.1145/2184319.2184345.

[39] A. J. Smith, ‘Cache memories’, *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982, doi: 10.1145/356887.356892.

[40] T. Rothwell and J. Youngman, ‘The GNU C Reference Manual’, *Free Software Foundation, Inc*, p. 86, 2007, [Online]. Available: https://phoenix.goucher.edu/~kelliher/s2014/cs311/gnu-c-manual.pdf

[41] S. Hengel, ‘hpack’. [Online]. Available: https://hackage.haskell.org/package/hpack

[42] ‘LALRPOP’. [Online]. Available: https://crates.io/crates/lalrpop

[43] M. Forsberg and A. Ranta, 'BNF converter', in *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, in Haskell '04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 94–95. doi: 10.1145/1017472.1017475.

[44] W. Kuich, 'Algebraic systems and pushdown automata', *Algebraic Foundations in Computer Science: Essays Dedicated to Symeon Bozapalidis on the Occasion of His Retirement*, pp. 228–256, 2011, doi: 10.1007/978-3-642-01492-5_7.

[45] R. Mahdavi-Hezaveh, J. Dremann, and L. Williams, 'Software development with feature toggles: practices used by practitioners', *Empirical Software Engineering*, vol. 26, pp. 1–33, 2021, doi: 10.1007/s10664-020-09901-z.

[46] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, 'System programming in Rust: Beyond safety', in *Proceedings of the 16th workshop on hot topics in operating systems*, in HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 156–161. doi: 10.1145/3102980.3103006.

[47] T. Ramananandro, G. Dos Reis, and X. Leroy, 'A mechanized semantics for C++ object construction and destruction, with applications to resource management', *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 521–532, 2012, doi: 10.1145/2103621.2103718.

[48] N. Heer, 'Speed comparison of programming languages'. [Online]. Available: https://github.com/niklas-heer/speed-comparison

[49] 'Memory safety'. The Chromium Projects. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[50] T. Gavin, 'A proactive approach to more secure code', *Microsoft Security Response Center*, Jul. 2019, [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[51] J. Vander Stoep and S. Hines, 'Rust in the Android platform', *Google Security Blog*, Apr. 2021, [Online]. Available: https://security.googleblog.com/2021/04/rust-in-android-platform.html

[52] D. Weston, 'Windows 11: The journey to security by-default', 2023, [Online]. Available: https://www.youtube.com/watch?v=8T6ClX-y2AE&t=3100s

[53] The Rust Foundation, 'The Rust compiler'. [Online]. Available: https://github.com/rust-lang/rust

[54] L. Mädje, M. Haug, and The Typst Project Developers, 'Typst'. [Online]. Available: https://github.com/typst/typst

[55] The SWC Project, 'Swc'. [Online]. Available: https://github.com/swc-project/swc

[56] Roc programming language foundation, 'Roc'. [Online]. Available: https://github.com/roc-lang/roc

[57] L. Pilfold, 'Gleam'. [Online]. Available: https://github.com/gleam-lang/gleam

[58] A. Liljencrantz, 'Fish shell'. [Online]. Available: https://github.com/fish-shell/fish-shell

[59] S. Akbary and other contributors, ‘Wasmer’. [Online]. Available: https://github.com/wasmerio/wasmer

[60] The Bytecode Alliance, ‘Wasmtime’. [Online]. Available: https://github.com/bytecodealliance/wasmtime

[61] Meta Platforms, Inc., ‘Starlark in Rust’. [Online]. Available: https://github.com/facebook/starlark-rust

[62] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023, ISBN: 978-1-7185-0310-6.

[63] A. Zeng and W. Crichton, ‘Identifying barriers to adoption for Rust through online discourse’, *arXiv preprint arXiv:1901.01001*, 2019, doi: 10.48550/arXiv.1901.01001.

[64] A. Bohra and E. Gabber, ‘Are mallocs free of fragmentation?’, in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 105–117.

[65] Y. Jo and M. Kulkarni, ‘Automatically enhancing locality for tree traversals with traversal splicing’, in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, New York, NY, USA: Association for Computing Machinery, 2012, pp. 355–374. doi: 10.1145/2384616.2384643.

[66] N. Fitzgerald, ‘bumpalo’. [Online]. Available: https://crates.io/crates/bumpalo

[67] D. Pager and X. Chen, ‘The Lane Table Method Of Constructing LR(1) Parsers’, APPLC'12, 2012. doi: 10125/33112.

[68] T. Sheard and S. P. Jones, ‘Template meta-programming for Haskell’, in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, in Haskell '02. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 1–16. doi: 10.1145/581690.581691.

[69] D. Stewart, D. Coutts, D. Roundy, S. Meier, and K. Ross, ‘bytestring’. [Online]. Available: https://hackage.haskell.org/package/bytestring

[70] C. Clark, ‘What to do with a dangling else’, *ACM SIGPLAN Notices*, vol. 34, no. 2, pp. 26–31, 1999.

[71] The Linux Kernel Organization, ‘The Linux Kernel’. [Online]. Available: https://www.kernel.org/

[72] P. David, ‘hyperfine’. [Online]. Available: https://crates.io/crates/hyperfine

[73] J. Duregård and P. Jansson, ‘Embedded parser generators’, in Haskell '11, vol. 46. Tokyo, Japan: Association for Computing Machinery, 2011, pp. 107–117. doi: 10.1145/2034675.2034689.

[74] U. Norell and other contributors, ‘Agda Compilers’. [Online]. Available: https://agda.readthedocs.io/en/v2.7.0.1/tools/compilers.html

[75] W. F. Clocksin and C. S. Mellish, *Programming in Prolog. Using the ISO Standard.*, 5th ed. Springer-Verlag, 2003, ISBN: 978-3-540-00678-7. doi: 10.1007/978-3-642-55481-0.