CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A common backend API for the BNF Converter

Implementing a new layer of abstraction to provide a common structure for the backends of the BNF Converter tool

Master's thesis in Computer science and engineering

BEATRICE VERGANI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

A common backend API for the BNF Converter

Implementing a new layer of abstraction to provide a common structure for the backends of the BNF Converter tool

BEATRICE VERGANI

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

A common backend API for the BNF Converter
Implementing a new layer of abstraction to provide a common structure for the backends of the BNF Converter tool
BEATRICE VERGANI

Supervisor: Andreas Abel, Computer Science and Engineering
Examiner: Aarne Ranta, Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

A common backend API for the BNF Converter
Implementing a new layer of abstraction to provide a common structure for the backends of the BNF Converter tool
BEATRICE VERGANI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The BNF Converter is a compiler construction tool that, starting from a labelled BNF grammar, generates a compiler front-end, an abstract syntax and a pretty printer in a chosen target language, together with a makefile, a TeX file and a txt2tags file containing readable specifications of the language described by the BNF grammar.
The original implementation of the tool presents a backend for each target language. This thesis is part of a reimplementation of BNFC and provides a common API for the backend part of the project, so that the tool can work with common structures and data types to express invariants, enhance type safety and ease backend modifications and maintenance as they can be compiler-driven.

Keywords: Computer science, engineering, thesis, programming languages, programming languages technology, compilers construction, front-end, API, haskell.

Acknowledgements

First and foremost I thank my supervisor, Andreas Abel, for following and guiding me during this thesis work. He suggested this project and made the reimplementation of BNFC possible by contributing to its frontend. It has been a pleasure to collaborate with him. I would also like to thank Aarne Ranta for having been my examiner.
I would like to express my gratitude towards the University of Chalmers, I regard my Master here as an extremely important piece of my education. The courses I was able to attend represented important opportunities that, together with the brilliant people I met in this journey, helped me grow in many ways. I will always remember the last two years as happy and full.
Finally, I thank my family, my boyfriend Francesco and my best friend Jack for their support.

Beatrice Vergani, Gothenburg, December 2021

Contents

1

Introduction

The *BNF Converter* (BNFC) [4] [29] is a compiler construction tool that, starting from a labelled BNF grammar, generates a compiler front-end in a chosen target language. The tool is currently able to generate Haskell, Agda, C, C++, Java, and OCaml front-end code. For each supported programming language the BNF Converter produces a lexer and a parser specification that will be processed by the lexer and parser generator tools specific to the language. It also produces an abstract syntax implementation, a pretty printer, a makefile, a TeX file and a txt2tags file with readable specifications for the language. The fact that BNFC supports several target languages makes it a versatile tool, broadening its potential users.

1.1 Uses of BNFC

BNFC can be used for educational purposes in the programming language technology field. In this case its multilingual feature becomes particularly important as students can develop their projects in some of the supported target languages, depending on what languages they already know or they are more proficient with. Other than for teaching, it can also be used for quick language prototyping as BNFC provides the basic infrastructure of abstract syntax with parsing and printing. The frontend provided by BNFC can be refined, for instance by writing better error messages or modifying the pretty printer.

1.2 Problem description

In the current official version of BNFC (2.9) the only common part consists of the code taking care of parsing the command line options specified by the user and parsing the input grammar file. After this initial processing all the rest of the work is delegated to the back-end part of the project, that generates the abstract syntax, the pretty printer and the other components. For all the supported languages there is a backend implementation written from scratch, where each backend has the same structure, generating the same components, but there is no abstraction capturing it. The lack of modularity of BNFC causes a significant amount of duplication that complicates the maintenance of the project and limits its scalability as any general change regarding the backend needs to be replicated in each one of them, making

the project also more error-prone.

1.3 Project description

The aim of this thesis is to implement a new layer of abstraction to provide a common API for the backends of the BNF Converter so that they can work by sharing a common structure to generate their components for all the supported target languages. Having a shared abstraction over the backends allows to express invariants about them using common types, enhancing type safety and ease backend modifications as that they can be compiler-driven. Since the existing code base was old, not always coherently structured and the fact that the use of the new API would have meant some significant changes, my supervisor and I agreed to reimplement the tool from scratch, so the new API is developed in version 3 of BNFC.

1.4 Goals

The main goal of this thesis consists of designing and implementing an API aimed to enforce a common structure over the backends of BNFC that will each implement an instance of it. In particular, the new API will be developed by introducing new Haskell data types and functions abstracting the common functionalities of the backends and defining new interfaces for the abstract syntax, pretty printer and lexer and parser specifications.

Alongside with designing and implementing the new API the other goals consist of reimplementing BNFC improving both its code and output quality. The effort in writing the new source code is oriented towards avoiding duplication, increasing modularity by handling information in more steps, representing it with intermediate formats, and by providing the user with clearer and better structured error messages. Both the project maintenance and extension, with new features or supported languages, will benefit from such modular structure as it will allow more separation between concerns.

1.5 Contributions

This thesis contributions consists of designing an API capturing the backends common structure and reimplement some of the backends to follow that structure. The implementation of the new API should bring about other changes and improvements in the BNF Converter. The haskell pretty printer will use a pretty printing library, and errors will be improved by associating them clearer error messages.

1.6 Structure of the thesis

The thesis will be structured to follow the contributions made to the reimplementation of BNFC and will be mainly focused around presenting the design of the new

API for the backends and the phases that lead to its development. A preliminary background (chapter 2) will introduce the knowledge necessary to understand the work described in the rest of the thesis by illustrating the concepts of the front-end of a compiler, LR parsing and by giving an overview of the parser generators used in the BNFC tool. The central chapters will be dedicated to the description of the backend API (chapter 3) and the implementation of some of its instances (chapter 4). After having presented the implementation details the discussion in chapter 5 will be devoted to reflect on what has been achieved, the thesis will be concluded making the final considerations.

2

Background

The background chapter will use the simple labelled BNF grammar presented in listing 1 to provide a reference for the BNFC grammar formalism [24], further information about the BNFC tool and its formalism can be found in [23] and [27]. After that will follow an overview of the lexer an parser generators (see table 2.1) in which, starting always from the grammar example as input for BNFC, the respective generated specifications will be presented to ultimately derive a common structure. The notion of compiler frontend will also be simply explained, for more detailed theory the reader is referred to Aho et al. [21].

2.1 A simple grammar example

The grammar below serves as a simple, easy to understand example for grammar productions as well as commonly used macros such as coercions, separators and terminators.

```
entrypoints [Stm];
terminator Stm "";

Input. Stm  ::= "input" Ident;
Print. Stm  ::= "print" Exp;

Sum.   Exp  ::= [Exp1];
Times. Exp1 ::= Exp1 "*" Exp2;
Num.   Exp2 ::= Integer;
Var.   Exp2 ::= Ident;

coercions Exp 2;
separator nonempty Exp1 "+";

comment "--";
```

Listing 1: A simple grammar example

The grammar has two categories: statements and expressions. The statements

category `Stm` has two rules, defined with labels `Input` and `Print`. The `terminator` macro expresses that the entries of lists of statements terminate with the token `""`, and is expanded by BNFC into the to list rules in listing 2.

```
[].  [Stm] ::= ;
(:). [Stm] ::= Stm "" [Stm] ;
```

Listing 2: List rules for terminator macro

The expressions category (`Exp`) has four rules, given by the labels `Sum`, `Times`, `Num` and `Var`. There can be list of expressions that are separated by the `+` symbol. This is expanded to the following list rules in listing 3 that, because of the `nonempty` pragma, do not have a case for the empty list.

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

Listing 3: List rules for separator macro

The `coercions` macro generates embeddings between expressions at different precedence levels, so that an expression at a higher precedence is accepted where an expression of lower precedence is needed. See listing 4 for the rules generated in the case of our example.

```
_. Exp  ::= Exp1 ;
_. Exp1 ::= Exp2 ;
_. Exp2 ::= "(" Exp ")" ;
```

Listing 4: List rules for coercion macro

The `entrypoints` pragma allows to specify which of the categories parsers to export, in this case the one for list of expressions.

From the grammar above are derived the token definitions in the below listed character classes (listing 5) and lexing rules (listing 6). Characters include whitespace, letters, both lower or upper case, digits from 0 to 9 and identifiers, which range over alphabet letters, again both lower or upper case, and the symbols `_` and `'`. Lexing rules provide the lexer with patterns to recognize the tokens. The input, print, times, plus, left and right parenthesis tokens are matched with the string containing their corresponding symbol. Integer tokens are non-empty sequences of digits, expressed via the regex `"+"` operation for non-zero repetitions. Similarly, ident tokens correspond to one letter followed by any number of identifiers. Whitespaces and comments are ignored by the lexer.

```
[:whitespace:]          [ \n\r\f\t]
[:letter:]              [a-zA-Z]
[:digit:]               [0-9]
[:idchar:]              [a-zA-Z0-9_']
```

Listing 5: Generated character classes for the example grammar

```
[:whitespace:]+         {}
"--".*                  {}
"input"                 { TokInput   }
"print"                 { TokPrint   }
"*"                     { TokTimes   }
"+"                     { TokPlus    }
"("                     { TokLParen  }
")"                     { TokRParen  }
[:letter:][:idchar]*    { TokIdent   }
[:digit:]+              { TokInteger }
```

Listing 6: Generated lexing rules for the example grammar

2.2 The front end of a compiler

Among the files generated by the BNF Converter there are a lexer and a parser specification for all the lexer and parser generators of the supported target languages, where *lexer* and *parser* are two components of the front end of a compiler. A compiler is a software that reads a program written in a source language and translates it in a semantically equivalent program written in a target language [21]. This translation process happens in two phases: analysis and synthesis. The analysis phase, also referred to as *front end*, identifies the constituent components of the source language and imposes a grammatical structure on them, yielding an intermediate representation of the source code to feed as input of the synthesis part. The synthesis part, also called *back end*, uses the intermediate representation and the symbol table to produce the program in the target language. The BNF Converter targets two components of the front-end of a compiler: *lexer* and *parser*.

2.2.1 Lexer

The lexer [21] component operates the first compilation phase which is the *lexical analysis*. The lexer reads a string of characters representing a source program and divides it into meaningful sequences (*lexemes*) and for each of them produces a *token object*.

The input string goes through a *scanning* or *lexical analysis* phase where the lexer produces a token string from a character stream. The lexemes matching a token

are recognized according to *patterns* [1], regular expressions can be a way to specify patterns for tokens. After the lexing, the sequence of tokens will be passed to the next compilation phase: parsing.

```
"2 + 3 * x"
```

Listing 7: Input example, w.r.t. grammar in listing 1

```
TokInteger(2) TokPlus TokInteger(3) TokTimes TokIdent("x")
```

Listing 8: Result of the lexical analysis of the input example in listing 7.

2.2.2 Parser

The string of tokens produced by the lexer is passed to the parser which implements a *syntax analysis* by creating a syntax tree representing how the tokens are structured together according to the context-free grammar.

All context free languages can be parsed in $\mathcal{O}(n^3)$ time [30], where n is the length of the input string to parse. However, cubic complexity is too high to be used in practice where one would want the time complexity to be somewhat close to a linear. LR grammars can be parsed in quasi-linear time, work well in practice and many are the parser built to handle such grammars. However, unlike context free grammars (CFG), LR grammars are not compositional, so when engineering an LR-grammar one often needs to examine the generated LR-automaton to find problems. BNFC confines itself to pass a CFG specification to the parser generators, it doesn't guarantee that the parser generator will accept without generating errors. The parser generators targeted by BNFC mostly produce *LALR* parsers, except from ANTLR that generates recursive descendent parsers. LALR is a minor variant of *LR(1)* [21].

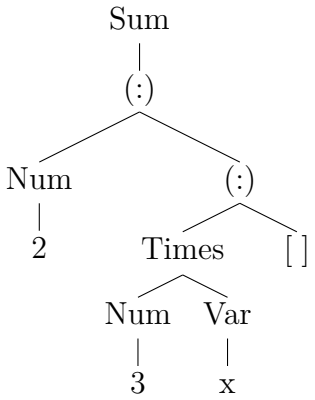

Figure 2.1: Abstract syntax tree for example input in listing 7

[1] description of the form that a token lexeme might have

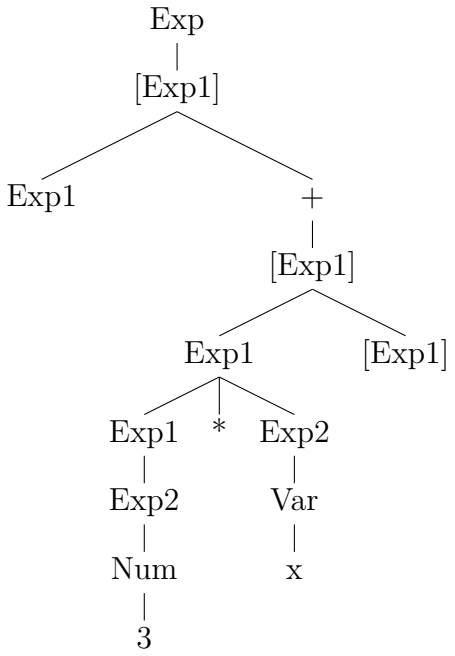

Figure 2.2: Parse tree for example input in listing 7

2.3 Overview of lexer and parser generators

This section is dedicated to analyze and discuss the structure of the grammar specifications that the lexer and parser generators listed in table 2.1 take as input. The specifications are generated by calling BNFC on the example grammar in listing 1, specifying the option corresponding to the examined generator.

Language	Option	Lexer generator	Parser generator
Haskell	–haskell	Alex	Happy
	–haskell-gadt	Alex	Happy
Ocaml	–ocaml	Ocamllex	Ocamlyacc
	–ocaml-menhir	Ocamllex	Menhir
Java	–java	JLex	CUP
	–java-antlr	JLex	ANTLR
C	–c	Flex	BISON
C++	–cpp	Flex	BISON

Table 2.1: List of the used lexer and parser generators according to language

2.3.1 Lexer generators

A lexer generator is an application that produces a lexical analyzer (lexer) for a specified lexer definition.

2.3.1.1 Alex

Alex [1] produces a lexer as a Haskell module starting from a description of the tokens to be recognized from given regular expressions. The Alex lexical specification starts with a few lines of optional Haskell code enclosed between braces, that will be copied verbatim into the generated lexer. This prelude section is followed by another optional section called wrapper that provides functionalities to interact with the Alex generated lexer.

After that there is a section containing macros definitions specifying tokens: macros can either be *character set macros* (listing 9), which begin with a $ symbol, or *regular expression macros*, which begin with a @ symbol. Macro definitions are followed by lexing rules (listing 10). The `eitherResIdent` function yields a token given a string that can be either an identifier or a reserved word.

In the final part of the specification can be found the `Token` data type (listing 11) and wrapper utility functions. The layout of an Alex specification file is summarized as $alex := [\ @\ code\ ]\ [\ wrapper\ ]\ \ macrodef\ \ @id\ ' : -'\ \ rule\ \ [\ @code\ ]\ $ [1].

```
$c = [A-Z\192-\221] # [\215]  -- capital
$s = [a-z\222-\255] # [\247]  -- small
$l = [$c $s]                 -- letter
$d = [0-9]                   -- digit
$i = [$l $d _ ']             -- identifier character
$u = [. \n]                  -- universal: any character
```

Listing 9: Alex character classes

```
@rsyms =    -- symbols and non-identifier-like reserved words
   \* | \( | \) | \+

-- Line comments
"--" [.]* ;

$white+ ;
@rsyms
    { tok (\p s -> PT p (eitherResIdent TV s)) }

$l $i*
    { tok (\p s -> PT p (eitherResIdent TV s)) }

$d+
    { tok (\p s -> PT p (TI s))    }
```

Listing 10: Alex lexing rules

```
data Tok =
   TS !String !Int    -- reserved words and symbols
 | TL !String         -- string literals
 | TI !String         -- integer literals
 | TV !String         -- identifiers
 | TD !String         -- double precision float literals
 | TC !String         -- character literals
  deriving (Eq,Show,Ord)

data Token =
   PT  Posn Tok
 | Err Posn
  deriving (Eq,Show,Ord)
```

Listing 11: Token data type in Alex specification

2.3.1.2 Ocamllex

Ocamllex [11] produces a lexical analyzer for the OCaml language starting from a set of regular expressions and semantic actions. An ocamllex file can start and end with two sections, *header* and *trailer*, containing Ocaml code within braces, both of which can be omitted. Between header and trailer the `let` construct is used to name expressions such as character classes (listing 12) and lexing rules (listing 13). Here `Hashtbl.find symbol_table x` has an analogue function to the one of `eitherResIdent` function in the Alex specification file (section 2.3.1.1).

```
let _letter    = ['a'-'z' 'A'-'Z' '\192' - '\255'] # ['\215' '\247']
let _upper     = ['A'-'Z' '\192'-'\221'] # '\215'
let _lower     = ['a'-'z' '\222'-'\255'] # '\247'
let _digit     = ['0'-'9']
let _idchar    = _letter | _digit | ['_' '\'']
let _universal = _
```

Listing 12: Ocamllex character classes

```
let rsyms = "*" | "(" | ")" | "+"

(* lexing rules *)
rule token =
  parse "--" (_ # '\n')*
                { token lexbuf }
      | rsyms   { let x = lexeme lexbuf in try Hashtbl.find
      ↪  symbol_table x with Not_found -> failwith ("internal lexer
      ↪  error: reserved symbol " ^ x ^ " not found in hashtable") }
      | _letter _idchar*
                { let l = lexeme lexbuf in try Hashtbl.find
                 ↪  resword_table l with Not_found -> TOK_Ident l }
      | _digit+ { TOK_Integer (int_of_string (lexeme lexbuf)) }
      | _digit+ '.' _digit+ ('e' ('-')? _digit+)?
                { TOK_Double (float_of_string (lexeme lexbuf)) }
      | '\n'    { incr_lineno lexbuf; token lexbuf }
      | eof     { TOK_EOF }
```

Listing 13: Ocamllex lexing rules

2.3.1.3 JLex

JLex [9] is a lexical analyzer generator for the Java language. A JLex input file is structured in three sections: *user code*, *JLex directives* and *regular expressions rules*. In the generated file for JLex the user code consists of the declaration of the package and imports, JLex directives include the character classes definitions (listing 14) and lexing rules (listing 15), the latter being expressed by regular expressions.

```
LETTER  = ({CAPITAL}|{SMALL})
CAPITAL = [A-Z\xC0-\xD6\xD8-\xDE]
SMALL   = [a-z\xDF-\xF6\xF8-\xFF]
DIGIT   = [0-9]
IDENT   = ({LETTER}|{DIGIT}|['_])
```

Listing 14: JLex character classes

```
<YYINITIAL>\* { return cf.newSymbol("", sym._SYMB_0, left_loc(),
↪  right_loc()); }
<YYINITIAL>\( { return cf.newSymbol("", sym._SYMB_1, left_loc(),
↪  right_loc()); }
<YYINITIAL>\) { return cf.newSymbol("", sym._SYMB_2, left_loc(),
↪  right_loc()); }
<YYINITIAL>\+ { return cf.newSymbol("", sym._SYMB_3, left_loc(),
↪  right_loc()); }
<YYINITIAL>input { return cf.newSymbol("", sym._SYMB_4, left_loc(),
↪  right_loc()); }
<YYINITIAL>print { return cf.newSymbol("", sym._SYMB_5, left_loc(),
↪  right_loc()); }
<YYINITIAL>"--"[^\n]* { /* skip */ }

<YYINITIAL>{DIGIT}+ { return cf.newSymbol("", sym._INTEGER_,
↪  left_loc(), right_loc(), new Integer(yytext())); }
<YYINITIAL>{LETTER}{IDENT}* { return cf.newSymbol("", sym._IDENT_,
↪  left_loc(), right_loc(), yytext().intern()); }
<YYINITIAL>[ \t\r\n\f] { /* ignore white space. */ }
```

Listing 15: JLex lexing rules

2.3.1.4 FLex

FLex [6] is a lexer generator for the C/C++ languages. The input file for FLex consists of three sections: *definitions*, *rules*, *user code*. In the definitions section character classes can be found (listing 16). The rules section contains the lexing rules (listing 17). The user code in the final part of the file will be copied verbatim into the generated lexer.

```
LETTER  [a-zA-Z]
CAPITAL [A-Z]
SMALL   [a-z]
DIGIT   [0-9]
IDENT   [a-zA-Z0-9'_]
```

Listing 16: FLex character classes

```
<YYINITIAL>"*"              return _SYMB_0;
<YYINITIAL>"("              return _SYMB_1;
<YYINITIAL>")"              return _SYMB_2;
<YYINITIAL>"+"              return _SYMB_3;
<YYINITIAL>"input"          return _SYMB_4;
<YYINITIAL>"print"          return _SYMB_5;

<YYINITIAL>"--"[^\n]* /* skip */; /* BNFC: comment "--" */

<YYINITIAL>{DIGIT}+
    yylval._int = atoi(yytext); return _INTEGER_;
<YYINITIAL>{LETTER}{IDENT}*
    yylval._string = strdup(yytext); return _IDENT_;
<YYINITIAL>[ \t\r\n\f]          /* ignore white space. */;
<YYINITIAL>.               return _ERROR_;
```

Listing 17: Flex lexing rules

2.3.1.5 Common structure of lexer generators

From the previous overview of the lexer generators we can conclude that the considered tools enjoy a common structure regarding the format of the input files used to build the lexical analyzer. We observe that the input files for the different lexer generators contain similar sections: they can all start and/or end with a section containing user code which, in either case, can be omitted, they have a sections dedicated to macros, such as character classes, and directives to lexer generator tool. Finally, they all have a section displaying the lexing rules through which the tokens will be recognized by the lexer.

2.3.2 Parser generators

A parser generator is a tool that given an input specifying a grammar defining a language produces a parser for that language. BNFC uses an array of parser generators specific to the supported programming languages. The following section will give a general overview on the format of the inputs of the parser generators.

2.3.2.1 Happy

Happy [25] is a parser generator specific to the Haskell language which takes a happy grammar file and produces a parser as a compilable Haskell module.

A happy grammar starts with a module header which is a Haskell module and, as all Haskell code, is enclosed within braces. The header section may also contain directive specifying lower or higher precedence of operation, depending on the order they appear in the header, and their associativity. After the header come some declarations stating the name of the parser, a monad declaration that can be used for error handling, the Haskell type used for the tokens and the declarations of

tokens (listing 18). The grammar rules (listing 19) are then listed, where each case of the right hand side of the production is associated to some Haskell code in braces specifying the abstract syntax constructor and the context precedence of its arguments. Each rule is also associated to a type signature that will help documenting the grammar, fix type errors and make the haskell compiler parse faster. The specification ends with the definition of an error handling function.

```
%tokentype {Token}
%token
  '('      { PT _ (TS _ 1) }
  ')'      { PT _ (TS _ 2) }
  '*'      { PT _ (TS _ 3) }
  '+'      { PT _ (TS _ 4) }
  'input'  { PT _ (TS _ 5) }
  'print'  { PT _ (TS _ 6) }
  L_Ident  { PT _ (TV $$)  }
  L_integ  { PT _ (TI $$)  }
```

Listing 18: Happy specification tokens

```
ListStm :: { [AbsGrammar.Stm] }
ListStm : {- empty -} { [] } | Stm ListStm { (:) $1 $2 }

Stm :: { AbsGrammar.Stm }
Stm : 'input' Ident { AbsGrammar.Input $2 }
    | 'print' Exp { AbsGrammar.Print $2 }
```

Listing 19: Happy statements rules

2.3.2.2 Ocamlyacc and Menhir

OCamlyacc [12] and *Menhir* [10] are parser generators specific to the OCaml language which produces a parser from a context free grammar specification. The produced parsers have one parsing function for each entry point of the grammar.

OCamlyacc and Menhir work on a grammar specification that can start with an optional *header* section containing directive or auxiliary functions required by semantic actions, after which comes a declarations section where the all the tokens (listing 20), entrypoints and production rules (listing 21) are declared. The production rules also carry information about context precedence and associativity expressed by the numbers associated to the production argument. The specification ends with an optional *trailer* section. Errors are handled through a *parse-error* function which can be modified by the user. All sections can contain comments, that are enclose between /* and */ when appearing in the declaration and rules section, and between (* and *) when in the header or trailer section.

```
%token KW_input KW_print

%token SYMB1 /* * */
%token SYMB2 /* ( */
%token SYMB3 /* ) */
%token SYMB4 /* + */

%token TOK_EOF
%token <string> TOK_Ident
%token <char> TOK_Char
%token <float> TOK_Double
%token <int> TOK_Integer
%token <string> TOK_String
```

Listing 20: Ocamlyacc/Menhir specification tokens

```
pStm_list : stm_list TOK_EOF { $1 }
  | error { raise (BNFC_Util.Parse_error (Parsing.symbol_start_pos
  ↪  (), Parsing.symbol_end_pos ())) };


stm_list : /* empty */ { []  }
  | stm stm_list { (fun (x,xs) -> x::xs) ($1, $2) }
;

stm : KW_input ident { Input $2 }
  | KW_print exp { Print $2 }
;
```

Listing 21: Ocamlyacc/Menhir statement rules

2.3.2.3 ANTLR

ANTLR [26] generates parsers written in the java language. The structure of the grammar it takes as input consists of declarations followed by a list of rules, containing tokens and productions. The generated ANTLR grammar file has an entrypoints declaration, followed by the other grammar productions (listing 22).

```
listStm returns [ grammar.Absyn.ListStm result ]
  :  /* empty */
    { $result = new grammar.Absyn.ListStm(); }
  | p_2_1=listStm p_2_2=stm
    { $result = $p_2_1.result; $result.addLast($p_2_2.result); }
  ;
stm returns [ grammar.Absyn.Stm result ]
  : Surrogate_id_SYMB_4 p_1_2=IDENT
    { $result = new grammar.Absyn.Input($p_1_2.getText()); }
  | Surrogate_id_SYMB_5 p_2_2=exp
    { $result = new grammar.Absyn.Print($p_2_2.result); }
  ;
```

Listing 22: ANTLR statement rules

2.3.2.4 CUP

CUP [5] is a parser generator for the Java language. In the generated CUP grammar file there is a user code component that contains actions defined by the user and parser code which methods and variables will be placed within the generated parser class, both user and parser code are optional. After the user code comes the symbols list in which are declared the non terminals and the terminals of the grammar (listing 23), followed by the entrypoint(s). The last section of the specification file presents the grammar production rules (listing 24) where each element of the right hand side can be uniquely labelled with a name written after a colon.

```
nonterminal grammar.Absyn.ListStm ListStm;
nonterminal grammar.Absyn.Stm Stm;
nonterminal grammar.Absyn.Exp Exp;
nonterminal grammar.Absyn.Exp Exp1;
nonterminal grammar.Absyn.Exp Exp2;
nonterminal grammar.Absyn.ListExp ListExp1;

terminal _SYMB_0;    //   *
terminal _SYMB_1;    //   (
terminal _SYMB_2;    //   )
terminal _SYMB_3;    //   +
terminal _SYMB_4;    //   input
terminal _SYMB_5;    //   print

terminal Integer _INTEGER_;

terminal String _IDENT_;
```

Listing 23: Terminals and non terminals of cup specification

```
ListStm ::=  /* empty */  {: RESULT = new grammar.Absyn.ListStm(); :}
  | ListStm:p_1 Stm:p_2  {: RESULT = p_1; p_1.addLast(p_2); :}
;
Stm ::= _SYMB_4:p_1 _IDENT_:p_2  {: RESULT = new
↪  grammar.Absyn.Input(p_2); :}
  | _SYMB_5:p_1 Exp:p_2  {: RESULT = new grammar.Absyn.Print(p_2); :}
;
```

Listing 24: Cup statement rules

2.3.2.5 Bison

Bison [3] is a parser generator specific to the C and C++ languages that converts a context free grammar in a LR parser.

The Bison grammar starts with a prologue, enclosed between "%" and "%", containing imports and macros, together with variables and functions used for the actions in the grammar rules. The next section contains declarations of non terminal and terminal symbols (listing 25), then there is the rules section (listing 26). The specification ends with an epilogue, containing functions that didn't need to be used before to generate the parser.

```
%token _ERROR_
%token _SYMB_1    /*   (   */
%token _SYMB_2    /*   )   */
%token _SYMB_0    /*   *   */
%token _SYMB_3    /*   +   */
%token _SYMB_4    /*   input   */
%token _SYMB_5    /*   print   */

%type <liststm_> ListStm
%type <stm_> Stm
%type <exp_> Exp
%type <exp_> Exp1
%type <exp_> Exp2
%type <listexp_> ListExp1
```

Listing 25: Bison specification symbols

```
ListStm : /* empty */ { $$ = 0; YY_RESULT_ListStm_= $$; }
  | ListStm Stm { $$ = make_ListStm($2, $1); YY_RESULT_ListStm_= $$;
  ↪ }
;
Stm : _SYMB_4 _IDENT_ { $$ = make_Input($2); }
  | _SYMB_5 Exp { $$ = make_Print($2); }
;
```

Listing 26: Bison statement rules

2.3.2.6 Common structure of parser generators

All the previously discussed parser generators descend from the *Yacc* tool, which is another parser generator for the C language, this fact, as expected, contributes to input files sharing a common structure. There can be optional sections opening and closing the file between which are enclosed declarations regarding tokens, priority, symbols, and parsing rules.

3

API Description

3.1 Backend

The backends of the BNF Converter share the common goal of producing the same components: a LaTeX description, a txt2tags description, an abstract syntax, a pretty printer, a lexer and parser specification, a test for the generated parser and a makefile. The before mentioned components, except from the TeX and the txt2tags files, are specific to the target language and have been gathered in the common *Backend* type class (see listing 27).

The *Backend* type class has two associated type synonyms: *BackendOptions* and *BackendState*. Both of them are indexed on the *target* language as the options and state of each backend will be defined depending on the target language the backend instance is for. The backend options consist of those command line options that are specific to the target language, while there will still be a data type dedicated to the global options that are shared between all the backends. The backend state is a State monad that will be used to propagate a state during the backend run. It will contain common information needed in the different phases, allowing to avoid computing the same information more than once.

The backend type class is organised to have one method for each of its functionalities. The *parseOpts* method consists of a parser for the backend command line options. The purpose of *initState* is to initialise the state that is threaded through the backend components. Further checks specific to the target language may be performed on the input grammar. If a check fails an error message is returned, otherwise the computation can go on initialising the backend state and running the backend phases. Doing these checks before generating the components and sharing common results by using a state allows less repetition and increases the project modularity. The remaining methods take care of generating the components produce by the BNF Converter: the *abstractSyntax*, *printer*, *lexer*, *parser*, *parserTest* and *makefile* are functions that starting from the representation of the labelled BNF grammar (*LBNF*) generate respectively the abstract syntax, the pretty printer, the lexer specification, the parser specification, the test for the generated parser and the makefile. Their returned type is a state monad where the state is the backend state (*BackendState*) and the value is a list of tuples (*Result* type). The first tuple

item is the file path were the result will be stored and the second is the file content, represented as a string.

The backend components are obtained by calling the *runBackend* function (listing 28) which, unless the checks in the state initialization fail, returns the backend *Result*. This way, the result is stored in an intermediate format, does not have to be written on the disk straight away and can eventually be used for testing. Note that the use of the *State* monad is inherently sequential as the current state value depends on the value that the state had in previous computation. Hence, the order in which the backend methods are called in the *runBackend* function is relevant. The abstract syntax is computed first as it could alter the state with information that the printer, lexer and parser may depend on. For the same reason, other than to follow their natural order in a compiler, the lexer method is run before the parser method.

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

class Backend (target :: TargetLanguage) where
  type BackendOptions target
  type BackendState target
  parseOpts      :: Parser (BackendOptions target)
  initState      :: LBNF -> SharedOptions -> 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 27: Backend type class

```
runBackend ::
  forall target. Backend target =>
  GlobalOptions -> BackendOptions target -> LBNF ->
  Except String Result
runBackend globalOpts backendOpts cf = do
  st <- initState @target cf globalOpts backendOpts
  return $ flip evalState st $ do
    absSpec   <- abstractSyntax @target cf
    printSpec <- printer        @target cf
    lexSpec   <- lexer          @target cf
    parSpec   <- parser         @target cf
    parTest   <- parserTest     @target cf
    mkfile    <-
      if optMakeFile globalOpts
      then makefile @target cf
      else return []
    return $ concat
      [absSpec, printSpec, lexSpec, parSpec, parTest, mkfile]
```

Listing 28: Run backend function

Since the LaTeX and txt2tags descriptions do not depend on the target language, they do not have a dedicated method in the *Backend* typeclass. A *Backend* instance will be instead instantiated for both of them. Unlike the rest of the targets that are programming languages, TeX and txt2tags are markup languages. Nevertheless, they also implement *Backend* instances so that its possible to separate the generation of the language description from the one of the compiler frontend.

Another possible interface for the backend components could have been a record data type. However, that would have required to define the type families for the backend options and state outside of the record data type. Type classes, instead, can also include type families as associated types, making their use in the type class more explicit and allowing better error messages. Type classes also guarantee coherence as there can be at most one backend instance in scope for each supported language. A type class was therefore preferred, even though in this case there are no rules a backend instance needs to obey.

3.2 Abstract syntax

The following section describes the interface for the the abstract syntax of functional languages.

3.2.1 Functional languages

Abstract syntax for functional languages can be modeled by a series of algebraic data types. Grammar categories are represented as data types where the grammar labels

appear as constructors. Data types are also generated for token categories, while user defined functions are mapped to functions. The abstract syntax is generated starting from the LBNF fields holding information regarding the AST rules and the user defined tokens and functions.

AST rules can be simply represented as a map from *Type*s to a map from their corresponding *Label*s to the list of *Type*s in the rule rhs (listing 29), where *Type*s are grammar categories without precedence.

```
Map Type (Map Label [Type])
```

Listing 29: Simple representation of AST rules

$$Exp \rightarrow \begin{array}{lll} Num & \rightarrow & [Integer] \\ Sum & \rightarrow & [\,[Exp]\,] \\ Times & \rightarrow & [Exp, Exp] \\ Var & \rightarrow & [Ident] \end{array}$$

Figure 3.1: AST rules for `Exp` in listing 1 according to representation in listing 29

The representation in listing 29 can be extended with further helpful information for the abstract syntax generation. The chosen representation for the AST rules field (listing 30) in the LBNF data type is again a map from *Type*s to another map. The inner map is different from the one in listing 29. It consists of a map from *Label*s to a tuple. The first tuple item is again the list of non terminals of the rule right hand side. The second tuple item is a Map from the category precedence, encoded as an *Integer*, to the right hand side of the rule *ARHS*, containing both terminals and non terminals, and its position (*WithPosition*).

The *ARHS* type is useful to print the grammar rule from which a data type constructor came from in the form of documentation. The category precedence information is not used for the abstract syntax, but will be needed for the pretty printer. Being able to use the same representation both for the AST and the printer was also a reason why this representation was chosen. The abstract syntax rules also include rules associated to the *internal* pragma. Not being part of the concrete syntax, these rules can not be parsed with, but it is possible to print with them.

```
ASTRules =
  Map Type (Map Label ([Type], Map Integer (WithPosition ARHS)))
```

Listing 30: AST rules

$$
Exp \rightarrow \left[\begin{array}{lll}
Num & \rightarrow & (\ [Integer], \quad 2 \rightarrow 9{:}1\ [Integer]\ ) \\
Sum & \rightarrow & (\ [\ [Exp]\ ], \quad 0 \rightarrow 7{:}1\ [\ [Exp]\ ]\ ) \\
Times & \rightarrow & (\ [Exp, Exp], \quad 1 \rightarrow 8{:}1\ [Exp1, "*", Exp2]\ ) \\
Var & \rightarrow & (\ [Ident], \quad 2 \rightarrow 10{:}1\ [Ident]\ )
\end{array}\right]
$$

Figure 3.2: AST rules for `Exp` in listing 1 according to representation in listing 30

The user defined tokens come from the use of the *token* pragma. Their LBNF field (listing 31) consists of a map from the token name (*CatName*) to the token definition equipped with its position in the grammar file (*WithPosition TokenDef*). The token definition consists of regular expression defining the token and the fact if the token carries or not its position information, that is, if its declaration contains the *position* keyword.

```
TokenDefs = Map CatName (WithPosition TokenDef)

data TokenDef = TokenDef
  { positionToken :: PositionToken
      -- ^ Is it a @position token@?
  , regexToken    :: Regex
      -- ^ The defining regular expression.
  , isIdent       :: Bool
      -- ^ Is it the @Ident@ token?
  } deriving Show
```

Listing 31: Tokens

Functions come from the use of the *define* pragma and their field is also a Map, it associates the label of function rules to the function definition and position.

```
Functions :: Map LabelName (WithPosition Function)
```

Listing 32: Functions

The generated code for each rule and pragma should follow the same order as in the input grammar. The original order is remembered through the `Position` information, which is exploited to sort the maps entries in the state initialisation.

3.3 Pretty printer

The pretty printer consists of implementing a printing function for each grammar category and token. For each grammar category a printing function is defined by

cases, where in each case the rule label is a constructor, taking arguments corresponding to the terminals and non terminals in the rule right hand side. The printing functions for tokens take as argument the token definition. The printing functions produce an annotated document that will be passed to a rendering function. The pretty printer is coded using the fields containing the AST rules (listing 30) and tokens definitions (listing 31) that are also used to produce the abstract syntax.

3.4 Lexer and parser specifications

For the lexer and parser specifications a *Token* data type, showed in listing 11, is used. This data type holds information about the tokens corresponding to the builtin categories, i.e., *Char*, *Double*, *Integer* or *String*, the identifier category (*Ident*) and to user defined token categories. Despite *Ident* being considered a builtin, it has a dedicated case in the token data type. This is because in the BNFC 3 code base it is treated as a user defined token type. The list of *Token* categories used in the grammar will be stored in the backend state and used to generate the lexer and parser specification.

```
data Token = Builtin BuiltinCat | Identifier | UserDefined CatName
```

Listing 33: Token data type

The lexer and parser specifications also require information about the symbols and keywords of the input grammar. Symbols and keywords are the terminals of the grammar, where symbols consist of non alphabetic characters and keywords consist of alphabetic and numeric characters and are in this more similar to identifiers.

The LBNF data type has one field dedicated to symbols (listing 34), one to keywords (listing 35) and one for both symbols and keywords associated to a unique identifier (listing 36). Both the symbols and keywords fields are encoded as maps from the symbol or keyword, which are both synonyms of non empty strings, to list of positions where they occur in the grammar. The symbols and keywords field consists of a Map from names, encoded as non empty strings, to increasing identifiers.

```
Symbols :: Map Symbol (List1 Position)
```

Listing 34: Symbols

```
Keywords :: Map Keyword (List1 Position)
```

Listing 35: Keywords

```
SymbolsKeywords :: Map String1 Int
```

Listing 36: Symbols and keywords

3.4.1 Lexer specification

The lexer specification also makes use of the LBNF fields for single line and block comments, which consist of maps from the comment declaration position to, respectively, line comments or block comments. Line comments are represented with the non empty string starting the line comment, block comments are represented with two non empty strings enclosing the block comment.

```
type LineComments   = Map Position LineComment
newtype LineComment = LineComment String1
```

Listing 37: Line comments

```
type BlockComments = Map Position BlockComment
data BlockComment  = BlockComment String1 String1
```

Listing 38: Block comments

3.4.2 Parser specification

The parser specification uses the parser rules field (listing 39) in the internal representation of the grammar. Parser rules are encoded as a map from categories to a map from right hand sides to labels. Parser rules present a similar, but more simple, structure to the abstract syntax rules (listing 30). Unlike the AST rules, parser rules only contain the *parseable* rules of the grammar. That is, only the grammar rules that are not matched with the *internal* pragma.

```
Map Cat (Map RHS (WithPosition RuleLabel))
```

Listing 39: Parser rules

The parser specification also employs the *entrypoints* field (listing 40) of the internal representation of the grammar. It contains the categories specified with the *entrypoints* pragma, and maps those categories to the position(s) where the entrypoint was declared. If no entrypoint is declared the field is instantiated with the first category declared in the grammar file.

```
Entrypoints :: Map Cat (List1 Position)
```

Listing 40: Entrypoints

3.5 Parser Test

The parser test goal is to test the parser produced with the parser specification generated by the BNF Converter. The BNF Converter generates a parsing function for

each category specified in the *entrypoints* pragma. The parser for the first declared entrypoint (listing 40) will be used to parse example files written in the language of the input grammar. If the files are parsed successfully the resulting abstract syntax is finally printed.

3.6 Makefile

The makefile is produced by exploiting information regarding the options, which are stored in the backend state, and information regarding the input grammar.

4

API instances

This section describes the implemented backend instances: Haskell, LaTeX and Txt2Tags. The backends work on the *LBNF* data type which comes from by the frontend of BNFC.

4.1 Haskell

4.1.1 Options

Haskell options are showed in listing 41. They include, in this order, an optional namespace to be prepended in front of module names, a flag to put the generated components in a directory named like the grammar, an option specifying whether to represent tokens as strings or text in the Haskell backend and four more flags that can respectively make the AST a functor, make the AST data type derive *Data*, *Generic* and *Typeable*, output haskell code using GADTs and generate Agda bindings for the AST.

```
data HaskellBackendOptions = HaskellOpts
  { nameSpace    :: Maybe String
  , inDir        :: Bool
  , tokenText    :: TokenText
  , functor      :: Bool
  , generic      :: Bool
  , gadt         :: Bool
  , agda         :: Bool
  }

data TokenText = StringToken | TextToken
```

Listing 41: Haskell backend options

4.1.2 State

The Haskell state, illustrated in listing 42, contains the options, both global and specific to the language. A field is dedicated to the list of tokens that will be used

to generate the lexer and parser specifications. The other fields hold the abstract syntax rules, the parser rules, user defined functions and tokens.

```
data HaskellBackendState = HaskellSt
  { globalOpt         :: GlobalOptions
  , haskellOpts       :: HaskellBackendOptions
  , lexerParserTokens :: [Token]
  , astRules    :: [(Type, [(Label, ([Type], Map Integer ARHS))])]
  , parserRules :: [(Cat, Map RHS RuleLabel)]
  , functions   :: [(LabelName, Function)]
  , tokens      :: [(CatName, TokenDef)]
  }
```

Listing 42: State for the Haskell backend

Before the actual state initialisation, Haskell-specific checks are performed. Should the checks be successful then the state fields can be initialised, otherwise an exception will be thrown. The Haskell checks currently assess that if a grammar uses the layout pragmas, then it needs to contain the semicolon symbol and eventually also the curly braces. It is also ensured that no layout stop is declared without having also declared a layout start.

During the state initialisation command line options are retrieved. The list of lexer and parser tokens is computed. The abstract syntax rules, tokens definitions, parser rules and user-defined functions are sorted according to their declaration order in the input grammar, so that they can appear in the same order as in the generated files. User-defined functions and parameters are renamed if their names clash with Haskell reserved words.

4.1.3 Abstract syntax

The Haskell abstract syntax consists of a Haskell module in which the grammar rules are mapped to data types. It is produced starting from the internal representation of the input grammar. In particular, the abstract syntax has a data type for each grammar category where the constructors are the labels associated with that category and the constructor arguments correspond to the non terminals of the right hand side (rhs) of the label rule. User-defined tokens are also mapped to data types with one constructor, named the same as the token, and a string as an argument and a tuple of integers representing its position if the token is declared as a position token. Functions coming from the define pragma can be found as functions in the abstract syntax. Furthermore, haddock documentation is printed for each data type constructor, expressing from which rule rhs the constructor originated.

Considering the example in listing 43 and its relative abstract syntax in listing 44 we can see that in the abstract syntax there is a data type for the *Exp* category, having one constructor for each of its labels (*Ehalf, Edouble, Epower, EId*), with arguments

corresponding to the non terminals in the rhs of the labels rules. The function *eSqrt* is generated from the homonym function introduced by the define pragma. Lastly, the newtype *Id* represents the user defined token *Id*.

```
EHalf.    Exp2  ::= "1/2"           ;
EDouble.  Exp2  ::= Double          ;
eSqrt.    Exp1  ::= "sqrt" Exp1     ;
EPower.   Exp   ::= Exp1 "**" Exp   ;
EId.      Exp   ::= Id              ;

coercions Exp 2                     ;

define    eSqrt e = EPower e EHalf  ;

token Id ( letter (letter | digit | '_')* ) ;
```

Listing 43: Grammar example

```
import qualified Prelude as T (Double, String)
import qualified Prelude as C (Eq, Ord, Show, Read, Int, Maybe(..))

import Data.String

data Exp
    = EDouble T.Double
    -- ^ Exp ::= Double
    | EHalf
    -- ^ Exp ::= "1/2"
    | EId Id
    -- ^ Exp ::= Id
    | EPower Exp Exp
    -- ^ Exp ::= Exp "**" Exp
  deriving (C.Eq, C.Ord, C.Show, C.Read)

-- | define eSqrt e = EPower e EHalf
eSqrt :: Exp -> Exp
eSqrt e = EPower e EHalf

newtype Id
    = Id T.String
  deriving (C.Eq, C.Ord, C.Show, C.Read, Data.String.IsString)
```

Listing 44: Grammar example's abstract syntax

4.1.4 Pretty printer

The code generating the Haskell pretty printer is written using the pretty printing library *prettyprinter* [16]. The generated pretty printer is a Haskell module defining a *Print* class with a printing method *prt* (listing 45). It presents an instance of that class for each token and category of the grammar.

```
class Print a where
  prt :: Int -> a -> Doc Ann
```

Listing 45: Print class

The printing method *prt* produces an annotated document (see listing 46) in which the elements of the grammar are annotated with their syntactic function, that is whether if they are a category, a literal, a keyword, a token or a list category.

```
data Ann
  = Keyword
  | Literal Literal
  | Token Token
  | Category Category
  | ListCat ListCat
```

Listing 46: Annotation data type

The obtained annotated document is then converted into a stream and passed to the render function. The render function (listing 48) consists of composing of a function (*render*), that will insert indentation and new lines in the stream, with another function (*renderLazy*) that will highlight syntactical elements with their corresponding colours. Ultimately, the top level printing function (listing 47) is given by calling the renderer on the stream resulting from the streaming function. The printing (listing 47), rendering (listing 49) and streaming (listing 48) functions, together with the *render* and *annToAnsiStyle* functions, are all exported by the module. This aims to make the printer more configurable by the user, as the user is able to replace them with functions that best fit their needs, should they desire to do so.

```
printTree :: a -> String
printTree = renderTree . streamTree
```

Listing 47: Print tree function

```
renderTree :: SimpleDocStream AnsiStyle -> String
renderTree = unpack . renderLazy . render 0 False
```

Listing 48: Render tree function

```
streamTree :: a -> SimpleDocStream AnsiStyle
streamTree a = layoutSmart defaultLayoutOptions
  $ annToAnsiStyle (docTree 0 a)
```

Listing 49: Stream tree function

Let us consider the *Print* instance produced for *Exp* category from the example in listing 43. The instance for the *Exp* category (listing 50) has a definition of the *prt* method for each rule related to the *Exp* category, where the rule label is a constructor with arguments corresponding to the items in the rule rhs. The *prt* method takes as input the precedence level given by the context and the expression to print, it is then recursively called on the rules rhs items. The *prPrec* function compares the context precedence level with the one of the expression, and puts parenthesis around the printed document when the expression has a lower precedence. We can observe that documents generated from categories and keywords are respectively annotated as *CatExp*, which is a case of the *Category* data type, and with *Keyword* (see the *Ann* data type in listing 46). Literals, tokens and list categories are also annotated in the same way in their instances of the *Print* type class.

```
instance Print Abs.Exp where
  prt i (Abs.EDouble d)        = prPrec i 2 $ prt 0 d
  prt i Abs.EHalf              = prPrec i 2 $ annotate Keyword "1/2"
  prt i (Abs.EId id)           = prPrec i 0 $ prt 0 id
  prt i (Abs.EPower exp1 exp2) = prPrec i 0 $ hsep
    [ annotate CatExp (prt 1 exp1)
    , annotate Keyword "**"
    , annotate CatExp (prt 0 exp2)
    ]
```

Listing 50: Print instance for the *Exp* category for grammar in listing 43

4.1.5 Lexer specification

The Alex lexer specification starts with a prelude sections containing language and options pragmas, module name and imports.

After that, character macros and regular expression macros are declared. In the regular expression macro for reserved words and symbols are listed the grammar symbols, and the grammar keywords that contain Unicode characters, as Alex will only recognize keywords of ASCII characters. Symbols and keywords are retrieved from the corresponding fields of the LBNF (see listing 34 and listing 35).

The lexing rules for comments are produced by printing the string and strings of single line (listing 37) and block comments (listing 38) respectively. In case of block comments a multi-line regular expression that starts and finished with the block comment delimiters strings is generated. The LBNF field for token definitions

(listing 31) is used to obtain the lexing rules for the user defined tokens. This is done by printing the token name and the token regular expression. Lexing rules corresponding to the builtin categories used in the grammar are then printed. The LBNF field keeping track of which builtin categories are used is exploited to know which builtin rules to print.

After the lexing rules a data type for tokens is declared. The list of *Token*s (listing 11) in the state is used to define the cases of that data type as it will have one constructor for every builtin, one for the identifier and one for every user defined tokens. Builtins and the identifier cases have a specific constructor name, while the constructor name for user defined tokens is given by the name of the token (*CatName*). The constructors all take a string as an argument.

Similarly, the list of *Token*s in the state is also used to define by cases functions taking the token data type as input. The lexer specification also contains functions that do not require any specific data type or structure. In these cases the strings corresponding the functions will be directly printed.

Finally, the LBNF field containing symbols and keywords with their identifier (listing 36) is used to produce a binary search tree ordered by the identifiers of the symbols and keywords.

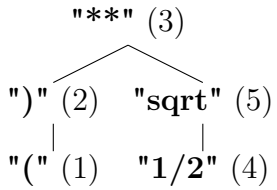

Figure 4.1: Symbols and keywords binary tree for grammar in listing 43

4.1.6 Layout

The Haskell language uses layouts, i.e., program elements can be grouped by indentation. The BNF Converter currently supports layout syntax as an experimental feature for the Haskell backend. When the *layout* pragma is used a Haskell module called *Layout* is generated. It provides functions to modify the stream of tokens coming from the lexer by inserting further tokens to recognize a layout block. Layout blocks are enclosed within braces, with its elements separated by semicolons.

The Haskell *Layout* module contains a list of the layout start and stop symbol names together with their identifiers. The layout start and stop symbols are retrieved by their corresponding fields in the LBNF data type, while their identifiers are looked up in the symbols and keywords map (listing 36).

4.1.7 Parser specification

The Happy parser specification starts with a Haskell module header containing language and options pragmas, module name and imports. This section holds instructions to export a parsing function for each entrypoint of the grammar, together with a parse error type and lexing functions.

The parser specification continues with the declaration of the parsing functions that Happy will generate, the type of the tokens that the parser will accept, the function to call in case of a parse error and the list of grammar tokens. The names of the parsing functions correspond to the names of the categories in the *entrypoints* field (listing 40) of the grammar.

The declared tokens are given by the union of the grammar's symbols and keywords and their priorities (listing 36) and by the tokens list in the state (listing 11).

The declaration section is followed by a section of rules. First come rules regarding *Token*s: builtins, identifier and user defined tokens. These rules are produced by a function that is defined by cases given by the list of tokens in the state. Then a production rule is defined for each parser rule (listing 39). As can been observed in listing 51, the rule name is given by the category name and the rule cases constructors correspond to the labels name. The constructors arguments are values consisting of a $ symbols followed by the corresponding indexes of the rhs non-terminals. The specification ends with a footer where the parse error and lexing function are defined.

```
Exp2 :: { Abs.Exp }
Exp2
  : '(' Exp ')'   { $2 }
  | '1/2'         { Abs.EHalf }
  | Double        { Abs.EDouble $1 }

Exp1 :: { Abs.Exp }
Exp1
  : 'sqrt' Exp1   { Abs.eSqrt $2 }
  | Exp2          { $1 }

Exp  :: { Abs.Exp }
Exp
  : Id            { Abs.EId $1 }
  | Exp1          { $1 }
  | Exp1 '**' Exp { Abs.EPower $1 $3 }
```

Listing 51: Happy rules for the *Exp* category for grammar in listing 43

4.1.8 Parser Test

The parser test consists of a Haskell module calling the generated parser on a number of example files. The parser function used to test the parser is the one for the first declared entrypoint. It is obtained by converting the entrypoints map to a list, sorting the entrypoints by their declaration order in the grammar, and taking the head of the list. The layout fields of the internal representation of the grammar are checked to see if the grammar uses layouts. If so, the layout resolver will be called before calling the generated parser.

4.1.9 Makefile

The Haskell makefile consists of a series of rules. It contains rules to build the lexer and parser, that will call Alex and Happy on the lexer and parser specification respectively. The *make all* rule will compile the components produced by the BNF Converter. The *clean* and *distclean* will remove the generated files.

The paths to files to be compiled and removed are based on the Haskell options in the Haskell backend state, which can specify to gather the components in a directory or to prepend a namespace to the modules names. The layout fields of the internal representation of the grammar are checked to see if layouts are used and, in case, the *Layout* module will be included in the list of files to be removed.

4.2 LaTeX

The LaTeX description generator of the language is also an instance of the *Backend* type class. Nevertheless, it does not have an implementation for all of the class methods. It will implement the state initialisation, the abstract syntax and the makefile methods.

Since the LaTeX backend has no specific options, its state will only be initialised to contain the global options.

The abstract syntax generator will produce the tex file describing the language. The tex file begins with commands and macros regarding the document. It then contains two sections, illustrating the grammar terminals and rules respectively. The terminals section documents which builtins, user defined tokens, keywords, symbols and comments appear in the grammar. The used builtins are retrieved from the corresponding LBNF field, for each of them will be printed the corresponding TeX description. The names and regular expressions of user defined tokens (listing 31) will be printed. Keywords (listing 35) and symbols (listing 34) are documented by listing their corresponding strings. Similarly, single line (listing 37) and block comments (listing 38) are listed together with their start and finish symbols. To write the rules sections it is necessary to process the abstract syntax rules to become a list of categories tupled with their list of right hand sides. The rules are displayed by associating the categories to their lists of right hand sides items, both terminal and non terminals, putting separator between the right hand sides.

The LaTeX makefile presents a rule to produce a pdf file by calling the executable *pdflatex* on the produced tex file and rules to clean the generated files. The only information used to code the LaTeX makefile is the name of the input grammar, which is retrieved by the global options. This is needed to write the rules that will produce the pdf, as the tex file will have the same name as grammar.

4.3 Txt2Tags

Similarly to LaTeX, the txt2tags description of the language also is a backend instance which will only implement the state initialisation, the abstract syntax and the makefile methods.

The Txt2Tags backend produces a t2t file that will be the input for *txt2tags* executable to produce a document of a certain target. Therefore, this backend state will have one option for the *txt2tags* target, which default value is the *html* target. The state will be initialised to contain the target option and the global options.

The abstract syntax generator will produce the t2t file describing the language. It is coded the same way as to the tex file for the LaTeX backend, the only difference lays in the syntax. The t2t file section documenting the grammar terminals is written by exploiting the information in their relative LBNF fields. The rules section is again produced by printing categories together with right hand sides.

The Txt2Tags makefile has a rule calling the *txt2tags* executable on the produced file, and other rules to remove the generated files. The grammar name is again learnt through the global options and used to refer to the generated t2t file.

5

Discussion

5.1 Related work

BNFC is a compiler construction tool that, starting from a Labelled BNF grammar, generates a specification for the lexer and parser components of the front end of a compiler. There are several lexer and parser generators that work on a grammar, for instance all the ones targeted by BNFC. However, they all are specific to a programming language and therefore do not share BNFC's multilingual nature. Among other similar multilingual tools that from a grammar generate a parser there are: *Grammatical Framework*, *ANTLR*, *syntax* and *tree-sitter*.

5.1.1 Grammatical Framework

The Grammatical Framework (GF) [28] [7] is a grammar formalism used for multilingual grammar applications. It provides a special-purpose, non Turing-complete, functional language (the GF language), a compiler for the language and a generic grammar processor [8]. GF programs are compiled to a multilingual grammar called *parallel multiple context free grammar* (PMCFG) [22]. It includes an abstract syntax definition and as many concrete syntaxes definitions as the supported languages. The abstract syntax contains a system of syntax trees and the concrete syntaxes implement a reversible map from the abstract trees to nested tuple of strings and integers. Given a multilingual grammar, GF generates a linearization function, that looks up a syntax tree in the concrete syntaxes map and retrieves the corresponding concrete syntax tuple, and a parsing function that achieves the opposite by returning an abstract syntax tree from a concrete syntax tuple.

BNFC and GF work on different grammar formats. GF ultimately handles a multilingual grammar (PMCFG), it has its own parser based on such grammar which time and space complexity are polynomial in the length of the input, with an exponent determined by the grammar [22]. BNFC handles context free grammars and does not produce a parser directly, it instead produces a specification to be fed to a parser generator, with the generated parsers mostly being LALR parsers.

Despite GF and BNFC sharing the ability to produce parsers for many target languages, one cannot substitute one for the other because of their different application domains. GF focuses on natural languages and can handle translations between

them by first parsing a string in a concrete syntax into an abstract syntax tree and then linearizing the tree to a string in the desired concrete syntax. BNFC instead targets the development of programming languages by proving the frontend of their compilers, easing prototyping.

5.1.2 ANTLR

ANTLR [26, 2] is a parser generator that given a grammar produces code recognizing the language defined by the grammar. The tool is able to produce a lexer and a LL(k) parser (Left to right, Leftmost derivation) in different target languages. The target language is specified through the command line options.

Both BNFC and ANTLR work with languages defined by context free grammars. However, differently from the BNFC tool, which grammar is written in Labelled BNF notation (*LBNF*), ANTLR's grammar is expressed by Extended BNF (*EBNF*) notation. ANTLR also allows to embed in the grammar actions written in the target language, they will then appear in the generated lexer and parser rules. Since the actions consist of target language code, ANTLR is not language agnostic like BNFC.

5.1.3 Syntax

Syntax [14] is a parser generator implementing an array of LR parsing algorithms, together with LL(1) parsing. The fact that it separates the algorithms for parsing table[1] calculation from the parser code generation makes it language agnostic. It is able to generated parsers in several target languages, where each target language corresponds to a plugin, making the tool easy to extend.

Syntax supports more low level grammar formats than BNFC. In particular, it accepts context free grammars written either in a JSON-like notation or in a Yacc/Bison-style notation. In such grammars is possible to specify the precedences and associativities of the grammars symbols and operators. Similarly to ANTLR, syntax allows to include code in the grammar file that will then appear at the beginning of the generated parser file. Although in this case the code is arbitrary, while in the case of ANTLR it has to be written in the target language. Like ANTLR, syntax current targets confine to languages belonging to the imperative and objected oriented paradigms, while BNFC also covers functional languages including Agda, which also provides verification facilities.

5.1.4 Tree-sitter

Tree-sitter [20] provides a parser generator. It receives as input a context free grammar written in JavaScript, which will then be converted to JSON format and go under a series of transformations to be finally split between its non terminals and terminals. Tree-sitter generates an incremental parser for the language expressed by the grammar written in the C language, together with bindings for the JavaScript

[1] a table encoding the grammar that given a state and a non terminal shows whether to apply a shift action, how to apply a reduce action and also how to compute the next state

and Rust languages. An incremental parser is a parser that is able to parse a program incrementally, i.e. if a program is modified it does not require to re parse the entire program. In the tree-sitter API text ranges are used to indicate what portions of the source program should be re parsed.

The tool comes with a series of other bindings for other programming languages, so that the generated parser can be integrated into projects written in the languages supported by the bindings. Tree-sitter also includes JavaScript grammars describing a wider set of programming languages and their relative parsers generated by tree-sitter. Ultimately, even though parsers are only generated in the C language, tree-sitter qualifies as a multilingual tool in that its parsers can be integrated in applications coded in some other languages.

5.2 Evaluation

The new backend API has been evaluated with tests, some of which involve golden testing and round trip testing.

5.2.1 Golden testing

The golden tests in BNFC 3 are based on the *tasty* test framework [19] combined with *tasty-silver* [18], which is a test runner supporting golden tests. Golden tests are performed by running BNFC 3 on example grammars, which will yield the list of generated components (see the *Result* data type in listing 27). The component that is been tested is looked up in the list and compared with the content of the corresponding golden file stored in the project. The test is successful when the produced out equals to the expected value in the golden file.

Golden testing has been used to test the Haskell lexer and parser specification and abstract syntax. Haskell golden tests also check the generated files when the *functor* and *gadt* are specified. These options respectively make the abstract syntax functorial and produce Haskell code that uses *GADT*s. The produced LaTeX and txt2tags descriptions have also been checked with golden testing.

5.2.2 Round trip testing

The Haskell pretty printer has been tested with round trip testing. The first step of these tests is parsing example files written in the languages of some example grammars. The parser takes as argument the string holding the file content and returns an abstract syntax object. As second step, the pretty printer is called on the abstract syntax object coming from the first parsing. As a result, a string is printed. In the third and last step, the printed string is parsed a second time. This testing can be summarized as parsing, printing and parsing again.

5.2.3 Testing the Haskell makefile and parser test

The makefile and parser test are tested together, as one of makefile's goals is to compile the parser test to produce its executable. These tests require to ability to execute shell commands, such as changing directory and calling makefile rules. This is achieved by using the *shelly* [17] library, which allows to insert shell like commands in the tests code.

Firstly, it is necessary to move to the makefile directory so that its *make* rule can be invoked. After the *make* rule invocation the parser test executable is available and it is used to try to parse examples after. Finally, the *make distclean* rule is called so that all generated file can be deleted. The *shelly* library assesses the given commands exit codes to be zero, so in case of a non-zero exit code an exception will be thrown.

5.3 Differences to BNFC 2.9

The section provides an overview of the aspects in which the version 2.9 and 3 of BNFC differ from each other.

5.3.1 Code base structure

The first difference is that BNFC 3 uses a common interface for its backends that are encoded as instances of the *Backend* type class. Since the *Backend* methods yield a list of file paths and file contents, the output of a backend is available as an intermediate result, instead of being written straight away.

The BNFC 3 code base decreases code duplication in comparison with BNFC 2.9. This is due to having a shared state, which allows to compute exactly once the needed information, and also to perform the specific checks exactly once, before initialising the state. The project is consequently less error prone, as it is not necessary to concern, and possibly risking to forget, about repeating the checks or the information processing in more contexts in the code base. BNFC 3 also presents an increased modularity and separation of concerns. Every target language has a dedicated folder containing dedicated modules for the generated components, options, state, state initialisation and eventually utilities function.

5.3.2 Types

In BNFC 3 the data type representing the labelled BNF grammar (*LBNF*) is more structured than the one in BNFC 2.9. It presents a field for each pragma (*token*, *comment*, *layout*, *entrypoint*) and rule (abstract syntax rules, parser rules, defined functions rules). While the grammar data type in BNFC 2.9 contains a list of pragmas, with the pragma data type having a case for each pragma, and a list of rules, containg all the grammar rules, where different rules are encoded as different fields of the rule data type. Dedicating a separate field to every pragma and set of rules, and also to the other grammar elements, contributes to make the *LBNF* data

type more explicit, increase separation of concern and therefore simplify the code base.

Furthermore, the types in BNFC 3 express invariants and consequently enhance type safety. For instance, names are encoded as non empty list of characters rather than strings, which ensures that names are indeed not empty. In the *LBNF* fields map data structures are used, meaning that duplicates are not allowed. In addition, BNFC 3 defines specific data types with specific constructors to describe the grammar elements, so that different elements can be distinguished also on a type level. In contrast, in BNFC 2.9 there is a tendency to define the grammar elements as synonyms of strings. The latter approach constitutes a risk with respect to type safety, since there is the possibility to exchange a data type for another.

In BNFC 2.9 there is also an inconsistent use of the string type and the document type of a pretty printing library. It is not an issue in bnfc3 where a pretty printing library [16] is always used.

5.3.3 Builtin types

The BNF Converter recognizes the following builtin types: *Char*, *String*, *Integer*, *Double* and *Ident*, where the *Ident* builtin stands for an identifier, that is a name staring with a letter followed by any combination of letters, digits, underscore and apostrophe symbols. In BNFC 3 builtins category are represented with their own data type, which does not include the *Ident* builtin because identifiers are encoded as strings and treated like user defined tokens. In particular, the token definition data type presents a boolean flag indicating whether the token is given by the use of *Ident*. Unlike BNFC 2.9, BNFC 3 allows the user to overwrite the builtins by defining categories or tokens with the same names. This entails that the used builtins now need to be qualified.

5.3.4 Options

The options of BNFC 3 have been refactored to use the *optparse-applicative* library [15]. Options consist of global options, which have their own parser, and a command establishing which backend to produce. There is one command for each backend, specifying a sub parser based on its target language's specific options, which need to be listed after the command.

The usage of BNFC 3 is illustrated in the following help message:

```
Usage: bnfc3 [--version] [--numeric-version]
[--license] [-v|--verbose] [--dry-run] [-f|--force]
[-o|--outdir OUTDIR] [-m|--makefile] GRAMMARFILE COMMAND
```

5.3.5 Haskell printer

Contrary to the generated printer of BNFC 2.9, which is based on the manipulation of strings, the one of BNFC 3 relies on the *prettyprinter* [16] pretty printing library

to produce an annotated document. The annotated document is rendered by calling rendering functions that are all exported and that the user can eventually modify. Exporting these functions and performing the rendering in more steps enhance the printer configurability.

A further difference concerns the instances of the *Print* class for list categories. In BNFC 2.9 there were two overlapping printing methods, one for categories and one for list categories. In the print instance of a category there would be a case for each of its rules followed, in case the grammar presents lists of that categories, by cases for the lists of elements of the category. In BNFC 3 there is instead only one printing method (*prt*) with categories and list categories implementing separate instances of the *Print* class. The cases of list categories instances consist of the empty list, the single element list and the list with more than one element. The empty list case is omitted when the list category is coupled with the *nonempty* pragma.

5.4 Future work

5.4.1 Backend instances

The future work consists of finishing implementing the backend instances for the remaining target languages. In order to do that it is necessary to provide a common interface for the abstract syntax of object oriented language. Alongside with that, a common interface for the lexer and parser specifications could also be implemented.

The project can eventually be extended to support new target languages. A suggestion could be to substitute the Txt2Tags backend with a backend producing a specification in other format, for instance markdown. This way the produced specification could be fed to another document converter, *pandoc* [13] for instance, which supports many formats.

5.4.2 Testing

When the remaining backend instances will be implemented they will need to be tested. The suggested approach would be to use building tools, as in the tests for the Haskell backend. Thus, one does not need to have all the generators and need libraries installed during testing.

5.5 Conclusion

In this thesis I contributed to the reimplementation of the BNF Converter. My work was mainly focused on designing a common API for the BNFC backends. Backends are now encoded as instances of the same type typeclass, the *Backend* type class, so that their methods are now bounded to the same type signatures. The backend methods employ a shared state to store useful information computed during the generation of the components. The components are coded exploiting common structures and data types.

The produced code base presents better modularity and separation of concerns, less duplication and a common structure for its backends. Being more structured, the BNFC tool is easier to be understood, also by external contributors, maintained and extended. Furthermore, the use of dedicated and more expressive data types and structures strengthens the type safety and increases the level of checking of the project.

To conclude, with this project I personally learned the important role that types and modularity play in easing development and reaching satisfactory results. I hope that the effort in this reimplementation will help the maintenance and growth of the project, together with guaranteeing its continuity. Being BNFC an open source project, I also hope that the project reimplementation will have a beneficial impact on the open source community.

Bibliography

[1] Alex user guide. https://www.haskell.org/alex/doc/alex.pdf. Accessed: 2021-04-06.

[2] ANTLR github. https://github.com/antlr/antlr4/tree/aeaa445b4d4f1c143814af6ec7438f38e66b6441.

[3] Bison manual. https://www.gnu.org/software/bison/manual/bison.pdf. Accessed: 2021-03-29.

[4] BNFC reference. https://bnfc.readthedocs.io/en/v2.9.3/lbnf.html/.

[5] CUP manual. https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html. Accessed: 2021-03-29.

[6] FLex user guide. https://www.cs.fsu.edu/~engelen/doc/reflex/html/index.html. Accessed: 2021-04-06.

[7] GF Reference Manual. http://www.grammaticalframework.org/doc/gf-refman.html/. Accessed: 2021-11-22.

[8] Grammatical Framework github. https://github.com/GrammaticalFramework/gf-core/tree/e4b2f281d97f97317df37b5e296cea371c334f37.

[9] JLex user guide. https://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html. Accessed: 2021-04-07.

[10] Menhir manual. http://gallium.inria.fr/~fpottier/menhir/manual.html/. Accessed: 2021-03-29.

[11] Ocaml manual. https://ocaml.org/releases/4.10/htmlman/lexyacc.html/. Accessed: 2021-04-07.

[12] Ocaml manual. https://ocaml.org/releases/4.09/htmlman/. Accessed: 2021-03-29.

[13] Pandoc website. https://pandoc.org/. Accessed: 2021-11-22.

[14] Syntax github. https://github.com/DmitrySoshnikov/syntax/tree/d751c887f7f2d5be22d2c5b061eaa53860c331ff.

[15] The optparse-applicative. `https://hackage.haskell.org/package/optparse-applicative-0.16.1.0`.

[16] The prettyprinter library. `https://hackage.haskell.org/package/prettyprinter-1.7.1`.

[17] The Shelly library. `https://hackage.haskell.org/package/shelly-1.9.0`.

[18] The tasty-silver library. `https://hackage.haskell.org/package/tasty-silver-3.3.1`.

[19] The tasty test framework. `https://hackage.haskell.org/package/tasty-1.4.2`.

[20] Tree-sitter github. `https://github.com/tree-sitter/tree-sitter/tree/65746afeff569478ea75f71927090c316debb96f`.

[21] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley 2nd edition, 2007.

[22] Krasimir Angelov. Incremental parsing with parallel multiple context-free grammars. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 69–76, 2009.

[23] Markus Forsberg and Aarne Ranta. Labelled BNF: a high-level formalism for defining well-behaved programming languages. In *Proceedings of the Estonian Academy of Sciences*, volume 52, pages 356–393. Estonian Academy Publishers, 2003.

[24] Markus Forsberg and Aarne Ranta. The Labelled BNF grammar formalism. *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg*, 2005.

[25] Simon Marlow and Andy Gill. Happy user guide. *Glasgow University, December*, 1997.

[26] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[27] Michael Pellauer, Markus Forsberg, and Aarne Ranta. Bnf converter: Multilingual front-end generation from labelled BNF grammars. Technical Report 2004-09, Computing Science at Chalmers University of Technology and Göteborg University, 2004.

[28] Aarne Ranta. *Grammatical framework: Programming with multilingual grammars*, volume 173. CSLI Publications, Center for the Study of Language and Information Stanford, 2011.

[29] Aarne Ranta. *Implementing Programming Languages*. Individual author and College Publications, 2012.

[30] Daniel H Younger. Recognition and parsing of context-free languages in time n3. *Information and control*, 10(2):189–208, 1967.