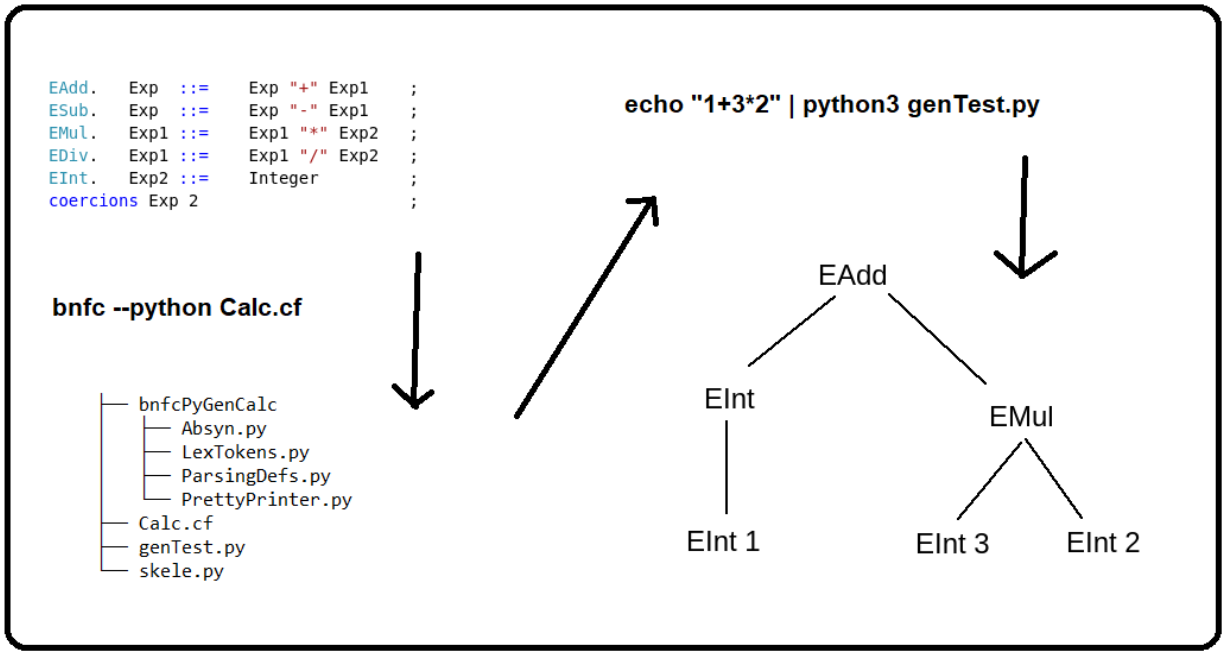


Python BNF Converter

Extending the languages supported by BNFC with Python

Master's thesis in Computer Science - Algorithms, Languages and Logic

Björn Werner

Gothenburg, Sweden 2024

Abstract

Given a specified grammar for some language, the BNFC tool can generate code for lexing and parsing input in that language. The project consists of extending the supported languages of BNFC to also include Python. To verify the usefulness of the implementation, a series of tests and exercises were carried out for a grammar of a subset of C++, comparing the functionality of the generated Python code to other generated parsers by BNFC, as well as coding a typechecker, interpreter and a compiler.

Contents

List of Acronyms

BNF	Backus-Naur-Form
LBNF	Labelled BNF
BNFC	BNF Converter
CF	Context-free grammar
AST	Abstract syntax tree
PLY	Python Lex Yacc

1

Introduction

1.1 Background

Backus-Naur Form (BNF) is a metasyntaxical notation for context-free (CF) grammars and can be used to specify how something should be written, for example something as simple as a the destination information format on a postal card. Something more complex like a programming language can be defined in this manner too, like how assignments to variables is to be written, or how a function definition must look like.

The BNF *Converter* is a back-end tool that, given some grammar specified in a BNF format, generates a set of files herein called a front-end, which one can use to implement:

- Typecheckers
- Interpreters
- Compilers
- Or something else

Here follows a practical example from the BNFC homepage [12].

Given a labelled BNF for arithmetical expressions, specified in a file called *Calc.cf*:

```
EAdd.  Exp  ::=  Exp  "+"  Exp1  ;
ESub.  Exp  ::=  Exp  "-"  Exp1  ;
EMul.  Exp1 ::=  Exp1 "*"  Exp2  ;
EDiv.  Exp1 ::=  Exp1 "/"  Exp2  ;
EInt.  Exp2 ::=  Integer         ;
coercions Exp 2                  ;
```

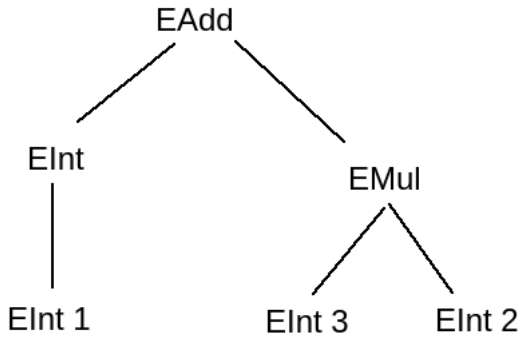
One can call upon BNFC to, in this case, generate a front-end for java, and additionally upon make - which compiles the generated java files for ready use.

```
bnfc --java -m Calc.cf && make
```

Now there exists a directory, named *calc* after *Calc.cf*, providing a front-end and an additional test program. By using echo, and pipe the output to a test programme, one can test the generated lexer and parser:

```
$ echo "1 + 3 * 2" | java calc/Test

Parse Succesful!

[Abstract Syntax]

(EAdd (EInt 1) (EMul (EInt 3) (EInt 2)))

[Linearized Tree]

1 + 3 * 2
```

The output is the abstract syntax tree that is parsed from the input.



Everything up until now was generated by the BNFC back-end, but now that we have the abstract syntax tree - one can create additional tools to process the abstract syntax tree in various ways:

- A typechecker to ensure that the parameters in all operations are of valid types, and to type-annotate the AST in order for an interpreter or compiler to make decisions based on the type of an expression [18][p. 58].
- Write an interpreter that evaluates the abstract syntax tree - like a small calculator that computes the expression. When designing a new programming language an interpreter is a quick way to get the language running [18, p. 81]
- If our BNF describes a new programming language - write a compiler, that takes the abstract syntax tree for some specific program we want to compile, and generates said program in some other target language like Jasmin [14] (the assembly language for the java virtual machine).

This motivates why BNFC is useful - but the generated skeleton files in Java can only directly be used by other tools created in Java. Hence there is a benefit to expand the languages supported by BNFC. The picture below shows which languages are

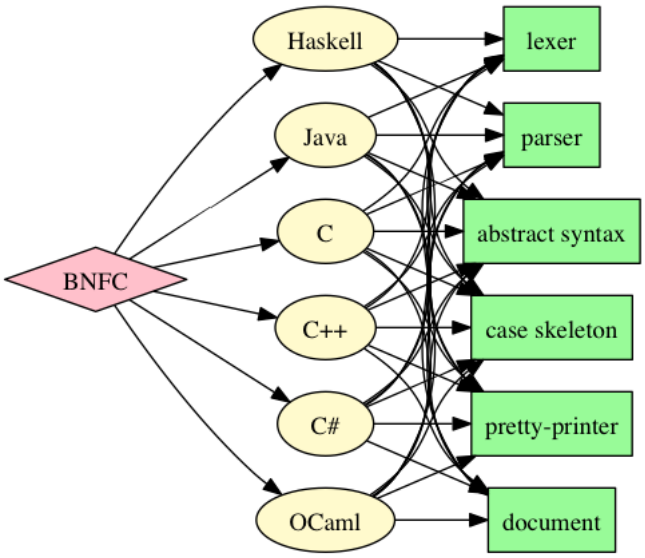


Figure 1.1: The first BNF Converters were made at Chalmers, in 2002. Since then six different programming languages have been implemented with BNFC support [17].

currently supported.

1.2 Aim

The aim of the project is to add Python support to BNFC, so the generated front-end consist of Python files. In essence so that a user would be able to run the following command to generate a frontend, which allows Python to be used to process abstract syntax trees specified in the target grammar:

```
bnfc --python Calc.cf
```

1.3 Goals

In order to add Python support to BNFC several components need to be implemented:

Lexer - which tokenizes an input string.

Parser - which reads the tokenized input, and stores them into a data structure that can be recursively evaluated (below, AST).

Abstract Syntax Tree - the data structure in which the structure is stored, that shows how the input is grouped.

Pretty printer - a way to print an abstract syntax tree in a human-readable format.

Traversal - provides a template code used to traverse the AST for subsequent implementation (e.g. type-checking), by using:

- Data classes - a decorator introduced in Python 3.7 to generate special class methods [19].
- Structural pattern matching, a programming pattern introduced with Python 3.10 to mimic pattern-matching [9].

Layout syntax - which is a way to structure what belongs to a block based on the indentation level [18].

An additional test program should also be created, that prints the abstract syntax tree along with the linearized output, as in the java example in the background section.

1.4 Limitations

Since the currently BNFC supported languages by themselves already form a template for how the Python implementation should behave - the limitations of the project are already pre-defined. No functionality outside the current functionality in BNFC is intended to be implemented.

1.5 Related work

There exists a myriad of parser generators, which also sometimes is described as compiler-compilers. YACC being one of them, standing for Yet-Another-Compiler-Compiler, as the subject was apparently popular in the 1960's [7, p. 257] [16]. A list of various existing parsing generators can be found in Wikipedia [21] each having their own: supported languages, parsing algorithms, input grammar notation, whether the grammar is defined separately or part of the code, whether a lexer is also generated, or what the licencing is.

Parsing generators that support python include: ANTLR [1], APG [20], GOLD [2], Lark [3], PLY [8], PlyPlus [4], SLY [6], SP [5] among others.

PLY - being the main technology of this report, uses a parsing algorithm called LALR(1) [8], just like some other parser generators that are used by BNFC does, as Yacc, Bison, CUP or Happy [18, p. 53].

2

Theory

2.1 Lexer

In lexical analysis, a lexer [7] tokenizes input according to defined grammar. This functionality may also be known as a scanner [15] or a tokenizer [11] depending on the litterature.

If we have a grammar consisting of integers and the addition symbol "+", we can define them with regular expressions. With the regular expressions, it is possible to convert an input string consisting of "1 + 23" into a tokenized form. By attaching the category for each element in the input we obtain a series of pairs:

```
(Integer, "1")
(Plus, "+")
(Integer, "23")
```

A lexer using the grammar rules above will however reject an input string such as "x + 1", since we have not defined any token that can be expressed with the character "x", either as a fixed token (as with "+" above), or as a generic character (as with the generic digit regular expression above).

An example of how identifiers could be expressed with regular expressions.

```
Identifier: letter(letter|digit)+
```

Note that if two token types would correspond to the same input, the grammar is ambiguous. Therefore in the example above, we enforce the first element for an identifier to be a letter, to prevent overlap with integers.

The tokenized form can now be used for other tools, such as parsers.

2.2 Parsers

A parser takes a sequence of tokens, and tries to build an abstract syntax tree. There exists different algorithms on how to perform the parsing, such as LL(k), LR(k) [16], and LALR which is an optimized variant of LR, at the cost of expressiveness [18].

LL(k) stands for left-to-right parsing, with leftmost derivations, and lookahead k [18]. Leftmost derivations state what end, in this case left, of an expression the parser tries to reduce first according to the rules of the grammar. LR(k) instead uses rightmost derivations. The lookahead k value indicates the number of unshifted tokens the parsing algorithm is to use, in order to determine how to proceed with the parsing. If k is not mentioned, it is by default 1.

For a grammar defined by the following rules which are using prefix notation, it is enough to only inspecting the leftmost token in the production on the right hand side, to know what operation is to be performed. Hence using a LL parser will suffice.

```
Exp ::= "+" Exp Exp
Exp ::= "-" Exp Exp
Exp ::= Integer
```

An example input could be "+ 1 - 2 3", which first could be tokenized by the lexer into (Plus, "+"), (Integer, 1), (Minus, "-"), (Integer, 2), (Integer, 3). The LL parser looks at the first token, being a "+" which is different from the alternatives "-" and integer, and hence knows that two expressions must follow.

The next token, being an integer, is afterwards parsed. Given that the rules state that nothing additional follows an expression consisting of an integer, the parser can then conclude that it has obtained the first expression for the "+" above.

The parsing continues and succeeds because at any step only one possible rule is applicable.

An example of a set of rules where the above LL parser would not succeed is for a grammar where operators use infix notation:

```
Exp ::= Exp "+" Exp
Exp ::= Integer
```

In this case, if the parser is implemented by attempting to use the first rule listed, it would loop infinitely as it tries to interpret the first token as an "Exp" [18, p. 49].

This problem can be solved with a LR parser, which stands for left-to-right with rightmost derivations. Instead of attempting to reduce the "Exp" completely in the example above before shifting anything else, the LR parser sees that the following token is a "+", and decides to continue to shift. After shifting the "+" the parser knows that another expression will follow, so the second Exp is shifted as well, and if no other token follows the second Exp, the LR parser starts the rightmost derivations, attempting to reduce the last occurrence of "Exp + Exp" into an "Exp".

Here follows an example of how LR parsing works, using the following grammar:

```
EAdd. Exp ::= Exp "+" Exp
EMul. Exp ::= Exp "*" Exp
EInt. Exp ::= Integer
```

The left column is the stack, which consists of the content that is being parsed. The middle column shows the input, and the right column describes what action that is to be performed next. Possible actions are "reducing" with some rule, "shifting" in new tokens onto the stack, "accept" the final category as a valid starting category, or "reject" if the contents of the stack can not be reduced any further.

Stack	Input	Action
	1 + 2 * 3	shift
Integer	+ 2 * 3	reduce with rule EInt
Exp	+ 2 * 3	shift 2x
Exp "+" Integer	* 3	reduce with rule EInt
Exp "+" Exp	* 3	shift/reduce conflict! shifting 2x
Exp "+" Exp "*" Integer		reduce with rule EInt
Exp "+" Exp "*" Exp		reduce with rule EInt
Exp "+" Exp "*" Exp		reduce with rule EMul
Exp "+" Exp		reduce with rule EAdd
Exp		accept

Table 2.1: LR-parsing of an expression.

A problem with infix operators is that they introduce shift/reduce conflicts. On the fifth row in the example above there are two possible actions, to either reduce the contents on the stack using the rule "EAdd", or continue shifting. The conventional procedure is to shift in such cases [18][p. 55]. Another type of conflict is a reduce/reduce conflict, which we will observe the two examples below:

Grammar with a reduce/reduce conflict:

```
NaturalNumber. Exp ::= Integer;
IntegerNumber. Exp ::= Integer;
```

Grammar with an implicit reduce/reduce conflict:

```
SVar. Stm ::= Ident;
SExp. Stm ::= Exp;
EId. Exp ::= Ident;
```

It is not possible to decide which rules to use to reduce an integer or an identifier in the examples above, as there are more possible reductions than one. These conflicts must be resolved by a rewrite of the grammar [18, p. 54]

2.3 Generating lexers and parsers

If a grammar conforms to the constraints of the lexer and parser architecture, it is possible to generate the lexer and the parser instead of writing them by hand. To this end, numerable tools have been developed, such as the Unix tools Lex and Yacc.

2.4 PLY

PLY [8] (short for Python Lexer Yacc) is a lexer, and a parser using LALR, implemented in Python. It is self contained within two separate files, **lex.py** and **yacc.py**. PLY utilizes the *inspect* module, which is part of the Standard Python Library [13], to perform metaprogramming such as obtaining information from variables and functions, starting with either "t_" or "p_" to build the lexer and the parser respectively.

An example for how a lexer for *Calc.cf* is defined, starting with tokens and regex for corresponding tokens:

```
tokens = ('PLUS', 'TIMES', 'Integer')

t_PLUS = r'\+'
t_TIMES = r'\*'

def t_Integer(t):
    r'\d+'
    return t
```

PLY uses the variable "tokens" and all other variables and functions starting with "t_", from which it builds the lexer. The fixed tokens for "+" and "*" are defined with fixed regex matchers, while an integer, regex matchers can instead be defined using a function. When using a function, PLY uses the regular expression defined underneath the name of the function, in this case "d+" ("d" short for digit and "+" indicating possible repetitions). The type of the parameter "t" is a "LexToken", with the two attributes "type", standing for what token it has, and "value", consisting of the characters from the input.

One can perform side effects such as converting the tokens characters, being a string into a python integer, like in the example below, or set the type to something else than "Integer".

An example where the token value is converted:

```
def t_Integer(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Together with a function that defines how errors are handled, the lexer in PLY can be built.

A tokenized input of "1+2" can afterwards be:

```
LexToken('Integer', 1)
LexToken('PLUS', '+')
LexToken('Integer', 2)
```

In order to have error handling, a specific function for the lexer must be defined:

```
def t_error(t):
    print(t[0] + ' was unable to be tokenized')
```

If desired, the lexing can continue, by skipping or by some other user defined behaviour, or the lexing may be halted completely.

By adding lexing of say identifiers, it is possible to tokenize expressions with variables, such as "x+1" or "variable5 + 1". The below method uses the regex "
w" which matches with any letter, digit or underscore.

```
def t_Ident(t):
    r'\w+'
    return t
```

In order to tokenize separate reserved words in statements, such as "int" in "int x", a separate keyword called "reserved" is used, similar to tokens. In order to match reserved keywords a map may be used in combination with the matching function, where the dictionary keys are the characters the keyword consists of, and the dictionary values are the token types. If the value of the token that is being matched already exists in the "reservedMap" dictionary, the type of the token should be updated. If the value of the token can not be found in the dictionary, the type will instead default to what was originally believed to be the type, being "Ident" in the example below:

```
reservedMap = {'int': 'INT'}
reserved = ('INT',)
tokens = reserved + ('Ident',)

def t_Ident(t):
   r'\w+'
   t.type = reserved_map.get(t.value, 'Ident')
   return t
```

The regex for a reserved word should not be defined using the previous "t_PLUS=r'+'" declaration, as that would allow concatenated expressions with "int", like "intint", instead of "int" being a separate word.

In order to ignore specific characters, a separate variable is defined. It is inherently assumed in the above example with reserved keywords that white-spaces are ignored, meaning that the white-spaces themselves are not tokenized. This behaviour is conventional [7][p. 54], as otherwise the parser would need to consider arbitrary amounts of tokenized white-spaces.

```
t_ignore = ' \t'
```

The functionality described above encapsulates what is used for the project in this

report. Additionally PLY has functionality that will not be described, such as: @TOKEN" decorators, optimized mode, debugging functionality, lexing states, lexer cloning, internal lexing states and conditional lexing.

2.4.1 Parsing in PLY's Yacc

Given a lexer, as built as in the previous section, one can define parsing definitions that PLY will use to build a parser, in order to parse tokenized input. Once the parsing definitions are defined, along with how to handle parsing errors, it is possible to build the parser.

An example of how addition of expressions can be defined for the grammar *Calc.cf*:

```
def p_Add(p):
    """
    Exp : Exp PLUS Exp
    """
    p[0] = ('Add', p[1], p[3])
```

The type of the parameter "p" is a "YaccProduction", which can be thought of as a list of what is being parsed. The parser will recursively parse the non-terminals, here being both "Exp", while the terminal "+", here using its token type as described in the previous section for the lexer, will be filtered out.

What to do with the parameters to form the result, stored in the first element of p, is up to the user. In this example a tuple is set to be what is constructed and stored. In order to signify that an addition is performed between two expressions the string "Add" is set as the first element of the tuple.

Notice the similarity of the target and production with how addition can be described in a BNF as in the introduction.

```
Add. Exp ::= Exp "+" Exp ;
Mul. Exp ::= Exp "*" Exp ;
```

The above BNF example lacks information regarding the precedence of operators. This means that "1+2*3" could be parsed as either $(1+2)*3$ or $1+(2*3)$ according to the BNF rules in the example above.

In BNF it is possible to overcome this by defining additional rules, coercions, which is how the project in this report combats the problem, but it is nevertheless possible to solve this in PLY by defining precedence for the operators.

Below is an example where the precedence of the operators has been defined, from lowest (first) to highest (last) for the operators used above. The additional keyword "left" here indicates that the infix operator is left-associative, meaning that in "1+2+3", "1+2" is evaluated first.

```
precedence = (
    (left, 'PLUS')
```

```
    (left, 'TIMES')
)
```

The variant that avoids this with coercion rules looks like the following in BNF:

```
Add. Exp ::= Exp "+" Exp1 ;
Mul. Exp1 ::= Exp1 "*" Exp2 ;

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

Here the number behind Exp indicates its priority. Directly translated to PLY parsing methods - this becomes:

```
def p_Add(p):
    """
    Exp : Exp PLUS Exp1
    """
    p[0] = ('Add', p[1], p[3])

def p_Mul(p):
    """
    Exp1 : Exp1 TIMES Exp2
    """
    p[0] = ('Mul', p[1], p[3])

def p_Exp1(p):
   """
   Exp : Exp1
   """
   p[0] = p[1]

def p_Exp2(p):
   """
   Exp1 : Exp2
   """
   p[0] = p[1]

def p_Exp(p):
   """
   Exp2 : LPAREN Exp RPAREN
   """
   p[0] = p[2]
```

In the last parsing definition the parentheses "(" and ")" are both added to the lexer, so it is possible to write expressions such as "(1+2)*3". Additionally the arithmetical operators use left associativity.

For grammars with more base-categories, for example Expressions and Statements, it may be desired to define what the starting category should be. Either for debugging reasons to parse one specific category, or because one may want to create a multifaceted program that can parser several different entrypoints. By default the starting category becomes the target category of the first defined rule in the grammar.

An example of a BNF with several categories, of which two are starting points, may be:

```
entrypoints Stm, Exp;

Assign. Stm ::= Ident "=" Exp ;

Add. Exp ::= Exp "+" Exp1 ;
Mul. Exp1 ::= Exp1 "*" Exp2 ;
EId. Exp2 ::= Ident ;
EInt. Exp2 ::= Integer ;

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

The way to set what the starting point should be in PLY is done via the "start" keyword. This keyword can be set when building a parser.

```
yacc(start='Stm')
```

Supporting several entrypoints can be accomplished by using several parsers, by attempting to parse with each parser iteratively, until one succeeds.

```
parsers = []
for start in entrypoints:
   parser = yacc(start=start)
   parsers.append(parser)
```

Finally, in order to use the lexer and parser in PLY on some input, the parser is initialized with a lexer as defined in the previous section.

```
ast = parser.parse(input, lexer)
```

The result of the parsing is in this case an AST, as we defined the result of each parsing rule to save a tuple with a description of the operation along with the expressions.

If the input is "1 + 2 * 3", the output becomes the AST:

```
    ('Add', ('Integer', 1), ('Mul', ('Integer', 2), ('Integer', 3)))
```

Instead of producing an AST, one might however define the parsing rules as to perform the mathematical calculation instead, in order to compute the result of "1 + 2 * 3".

Observe how the result can be changed to be the computation. Since python does not enforce strict typing it's entirely up to the user to choose what the result should be.

```
def p_Add(p):
    """
    Exp : Exp PLUS Exp1
    """
    p[0] = p[1] + p[3]

def p_Mul(p):
    """
    Exp1 : Exp1 TIMES Exp2
    """
    p[0] = p[1] * p[3]

def p_Integer(p):
    """
    Exp2 : Integer
    """
    p[0] = int(p[1])
```

For these rules the parser would instead produce the integer 7.

For grammars with more categories, such as statements and expressions, an entrypoint is needed, otherwise the first defined rule will serve as the default entrypoint [8]. Given some defined entrypoint, the parser attempts to reduce the input to said entrypoint. If several different entrypoints are desired, it can be done by generating several parsers, each with their own entrypoint.

2.4.2 Additional functionality for the parser

PLY enables combining parser rules, for which the target category is a shared one which can be done via a multiline-pattern in the docstring. This reduces the number of methods, but some extra logic has to be added to differentiate the possible production rules.

Below is an example where the rules for addition and times have been combined into one parsing function:

```
def p_BinaryOperation(p):
    """
    Exp : Exp PLUS Exp
        | Exp TIMES Exp
    """
    if p[2] == '+':
```

```
        p[0] = ('Add', p[1], p[3])
    elif p[2] == '*':
        p[0] = ('Times', p[1], p[3])
```

Instead of defining tokens with the corresponding regex it is possible to use a more direct approach by defining them as literals.

```
literals = ['+', '*']
```

This allows one to use the literals directly when defining the docstring in the parser function definitions. The length of the literals can at maximum be one character however.

```
def p_Add(p):
    """
    Exp : Exp '+' Exp
    """
    p[0] = ('Add', p[1], p[3])
```

When the parser is built a parser.out file can be created. This holds debugging information as well as information on possible conflicts found in the rules of the grammar. In order for the parser.out file to be generated, the debug-flag can be set when building the parser.

```
parser = yacc(debug=True)
```

Finally it is also possible to define empty productions. They are usually usable as the empty-list case for parsing lists in this project:

```
def p_EmptyListArg(p):
    """
    ListArg:
    """
    p[0] = []

def p_OneListArg(p):
    """
    ListArg: Arg
    """
    p[0] = [p[1]]

def p_ListArg(p):
    """
    ListArg: Arg ';' ListArg
    """
    p[0] = [p[1]] + p[3]
```

Additionally PLY has functionalities that will not be described, such as embedded actions, an optimization mode, debugging tools, among other things.

3

Methods (implementation)

The first half of this chapter describes how the PLY frontend is generated and the second half describes how the verification testsuite is implemented.

All generated Python files are a result of running BNFC on some grammar. Each section below describes a corresponding generated Python file.

A generated frontend for *Calc.cf* has a structure as illustrated on the front-page:

bnfcGenCalc/

    LexTokens.py

    Absyn.py

    ParsingDefs.py

    PrettyPrinter.py

genTest.py

skele.py

3.1 Generated Lexer

LexTokens.py contains the lexical information PLY needs to build its lexer. The output from the lexer will later be handed to the parser, which sets certain requirements on the output of the lexer. This information the lexer needs includes:

- Reserved tokens, like "int" and "if".
- Symbol tokens, like "+".
- Literals, like strings or integers.
- Regex rules.
- Characters that should be ignored, like whitespaces.
- How errors should be handled.

Since the parser of PLY does not allow concatenated special characters in the tokenized symbols, like "++", it is more practical to convert all symbols into combina-

tions of letters or numbers. A "+" is then converted into its unicode codepoint as a number.

The reserved tokens are converted into unicode representations as well, in case they contain special characters. Since a reserved token may overlap with say, a string, a check is done in the lexer method, to see if the token is in fact a reserved word. Then the type of the lexed token is set to its unicode representation. If no reserved token matches, it must be a string.

Below is an example:

```
reserved = ( 'int', )

reserved_map = {
    'int' : 'R_105_110_116',
}

tokens = reserved + ('S_43', 'String', )

t_S_43 = r'\+'

def t_String(t):
    r'"[^"]+"'
    t.type = reserved_map.get(t.value, 'String')
    return t

t_ignored = " \t\n"
```

3.2 Generated classes for the abstract syntax

The generated abstract syntax in **Absyn.py** consists of Python classes. One category of abstract classes which are meant to be used for indicating what type variables are, and the other category consisting of dataclasses, with member variables. The "@dataclass" decorator makes Python automatically create common class methods, such as the constructor `"__init__"` and the representation method `"__repr__"`, making it easier to print an instanced object of the class.

```
# Abstract classes
class Exp:
    pass

# Dataclasses building up the AST
@dataclass
class EAdd:
    exp_1: Exp
    exp_2: Exp
    ann_type: AnnType0 = field(default_factory=AnnType0)
```

```
@dataclass
class EInt:
    integer_: int
    ann_type: AnnType0 = field(default_factory=AnnType0)
```

Additionally, the ann_type variable is a placeholder for information that can be added to the syntax tree. For the purpose of this project, it has been a variable for annotating the type of nodes in the AST. In order for every instanced class of EAdd to not refer to the same ann_type object, the dataclass keywords field and default_factory are used to create individual instances of AnnType0. The somewhat odd class name AnnType0, attempts to evade a naming conflict with rules or categories in a grammar with the same name, which however, is still possible.

The class for AnnType0 has a set and a get function, where the set function raises an error if the type has been requested to change from anything else than the initial value "None", as a type should not be subject to change during typechecking. The class is defined as:

```
class AnnType0:
  def __init__(self):
    self.__v = None

  def s(self, val):
    if not self.__v == None:
        if self.__v != val:
            raise Exception('already has type: ' + str(self.__v) + ' and
                tried to set to ' + str(val))
    self.__v = val

  def g(self):
    return self.__v

  def __str__(self):
    return str(self.__v.__class__)

  def __repr__(self):
    return str(self.__v.__class__)
```

3.3 Generated Parser

ParsingDefs.py contains the generated parsing rules to build the parser. These are derived directly from the LBNF in the BNF file, like in the calculator example in the introduction. In order for PLY to build the parser every rule has to be formatted in a certain way, see the example below. The production is on the right hand side, and the result is on the left side. Any single parsing rule consists of, for example: the rule's name (EAdd), the value category (Exp), and the production consisting of the argument categories with the tokens required for the rule. This information has

to be in the docstring of the function.

Underneath the docstring is the defined behaviour for how PLY's parser should process the parsed token. The variable *p* can be thought of a list of the elements specified in the docstring, where the result is saved in `p[0]`. The result is produced using the dataclasses defined previously, where the two arguments to EAdd, an Exp and Exp1, comes from the array *p*. The `p[2]` consists of the plus token, and is therefore of no interest because the parser already decided to reduce with the rule EAdd.

In the example `S_43` is the unicode-mapped token for "+".

```
def p_EAdd(p):
    """
    Exp : Exp S_43 Exp1
    """
    p[0] = EAdd(p[1], p[3])

def p_EInt(p):
    """
    Exp2 : Integer
    """
    p[0] = EInt(p[1])
```

3.4 Generated PrettyPrinter

The generated **PrettyPrinter.py** file provides the two following printing options:

Print the AST, using the representation which is implicitly defined with the dataclasses.

Printing the "linearized tree", which is the AST rendered according to the whitespace conventions of C. This means characters as "{" behave as block enclosing literals, starting on a new line and thereafter with indentation. A grammar that significantly differs from C - where say the "{" is used as list separators for example instead of the usual comma "," - may therefore look strange in the linearized tree output.

3.5 Generated Skeleton code

The purpose of the generated **skele.py** file is to provide template code that is easy to start to use the abstract syntax and contains structural pattern matching using the match-case syntax in Python3.10. It is a form of tree-traversal - allowing the user to specify what should be done for some node that is being traversed.

The skeleton code provides two forms. One where all base-categories are within the same matcher, called *Generalized Algebraic Datatype* (GADT), and another where the base-categories are split into separate matchers.

An example is a BNF grammar of a fragment of C, where two base-categories are statements (Stm) and expressions (Exp) is:

```
# Categories combined into one matcher
def skeleMatcher(ast: object):
   match ast:
      case SExp(exp_, ann_type):
         # Exp ";"
         raise Exception('SExp not implemented')
    ...
    case EAdd(exp_1, addop_, exp_2, ann_type):
         # Exp4 AddOp Exp5
         raise Exception('EAdd not implemented')
    ...

# Categories split into separate matchers (Stm and Exp).
def matcherStm(stm_: Stm):
    match stm_:
      case SExp(exp_, ann_type):
         # Exp ";"
         raise Exception('SExp not implemented')
      case SDecls(type_, listid_, ann_type):
         # Type [Id] ";"
         raise Exception('SDecls not implemented')


def matcherExp(exp_: Exp):
   match exp_:
      case EMul(exp_1, mulop_, exp_2, ann_type):
         # Exp5 MulOp Exp6
         raise Exception('EMul not implemented')
      case EAdd(exp_1, addop_, exp_2, ann_type):
         # Exp4 AddOp Exp5
         raise Exception('EAdd not implemented')
```

In each "case" the attributes (like "exp_1" or "ann_type") of the any object is made accessible via the structural pattern matching.

3.6 Generated test file

A test file, called genTest.py is generated to easily test whether a given file can be parsed. If it is successfully parsed, the output includes the AST and the linearized output.

En example output is:

```
$echo "1 + 2 * 3" | python3.10 genTest.py
1 + 2 * 3
```

```
Building parser for entry: Exp

Trying to parse from Exp
Parse Successful!

[Abstract Syntax]
(EAdd (EInt 1) (EMul (EInt 2) (EInt 3)))

[Linearized Tree]
1 + 2 * 3
```

If a file is not parsed correctly, the lexer complains about what it could not tokenize by printing the line number, describes the error, accompanied by the specific place (and character) it got stuck on.

Example where the lexer can not tokenize all of the input:

```
$ echo "1 + 2 ? 3" | python3.10 genTest.py
1 + 2 ? 3

Building parser for entry: Exp

Trying to parse from Exp
Illegal character line 1: ? ascii: 63
```

If the input could be correctly tokenized, but failed the parsing, the parser prints out a syntax error.

Example where a syntax error is reported:

```
$ echo "1 + 2 + + 3" | python3.10 genTest.py
1 + 2 + + 3

Building parser for entry: Exp

Trying to parse from Exp
line: 1 lexpos: 8 Syntax error at '+'
Parse failed
```

3.7 How are the above files generated by BNFC?

In BNFC the backend has been extended with the python generating Haskell files. Originally the are modifications to the C generating implementation, as the C-backend looked like the least complicated and the easiest to modify.

```
/Backend/Python
    CFtoFlexPython.hs
```

```
      CFtoPythonAbs.hs
      CFtoPrettyPrinter.hs
      CFtoSkeleCode.hs
      PyHelpers.hs
      RegToFlex.hs
   Python.hs
```

Every generator primarily uses BNFC's CF.hs, which provides an API to extract information from a parsed LBNF file.

3.8 The primary data types in CF.hs

Roughly speaking, CF.hs provides the needed API to implement a backend for some target language. All the generating files use CF.hs primarily.

The file CF.hs describes how CF grammars, specified in BNF, are represented with a data type:

```
data CFG function = CFG
{ cfgPragmas      :: [Pragma]
, cfgUsedCats     :: Set Cat   -- Categories used by the parser.
, cfgLiterals     :: [Literal] -- @Char, String, Ident, Integer,
   Double@.
                               --   @String@s are quoted strings,
                               --   and @Ident@s are unquoted.
, cfgSymbols      :: [Symbol]  -- Symbols in the grammar.
, cfgKeywords     :: [KeyWord] -- Reserved words, e.g. @if@, @while@.
, cfgReversibleCats :: [Cat]   -- Categories that can be made
   left-recursive.
, cfgRules        :: [Rul function]
, cfgSignature    :: Signature -- ^ Types of rule labels, computed
   from 'cfgRules'.
} deriving (Functor)
```

A rule is defined as:

```
  data Rul function = Rule
{ funRule :: function
    -- ^ The function (semantic action) of a rule.
    --   In order to be able to generate data types this must be a
       constructor
    --   (or an identity function).
, valRCat :: RCat
    -- ^ The value category, i.e., the defined non-terminal.
, rhsRule :: SentForm
    -- ^ The sentential form, i.e.,
    --   the list of (non)terminals in the right-hand-side of a rule.
, internal :: InternalRule
```

```
    -- ^ Internal rule only for the AST and printing.
  } deriving (Eq, Functor)
```

A "sentform" is defined as the production of a rule, and is defined as:

```
type SentForm = [Either Cat String]
```

In essence, a rule defined as:

```
EAdd. Exp ::= Exp "+" Exp1
```

will have the sentform "[Left Exp, Right "+", Left Exp1]". The "+" will also be listed in the symbols of the grammar.

Given this information, as well as the rule's name (EAdd), it is possible to:

- From the symbols, define the tokens for the lexer in PLY.
- From the rules and categories, to define the Python classes for the abstract syntax, as well as the parsing definitions for PLY, by combining the token information and the abstract syntax classes.

3.9 CFtoFlexPython

Generates the LexTokens.py file, as well as supports the other generators with the character-to-unicode mapping.

In order to define rules for the lexer, as described in the theory chapter regarding tokens; reserved tokens; and special-character tokens, a character-to-unicode transformation is made, where a "+" for example is rewritten as "_43". For keywords this procedure yields longer obfuscated sequences of numbers, which will be illustrated below.

In essence, the following parts of CF, are used in order to create the components for the lexer:

```
Symbols -> t_S_43 = r'\*'
Keywords -> reserved_map dictionary
Literals -> def t_Id(t):
```

A small grammar with the reserved keywords "int" and "bool", along with the operators "+" and "*", and the literals "Integer" and "Id" - yields the following code for the lexer for PLY:

```
reserved_map = {
   'bool' : 'R_98_111_111_108',
   'int' : 'R_105_110_116',
}
```

```
reserved = (
   'R_98_111_111_108',
   'R_105_110_116',
)

tokens = reserved + ('S_42','S_43','Integer','Id',)

t_S_42 = r'\*'
t_S_43 = r'\+'

def t_Integer(t):
   r'\d+'
   t.type = reserved_map.get(t.value, 'Integer')
   return t

def t_Id(t):
   r'[A-Za-z](\_|(\d|[A-Za-z]))*'
   t.type = reserved_map.get(t.value, 'Id')
   return t
```

The reserved tokens have priority when being matched by attempting to use the dictionary method "get", which if it does not find the key (for example "int"), in reserved_map, the value of the lexed token will default to the functions matching category, here being "Id" for t_Id, which contains the regular expression that will match with "int".

3.10 CFtoPythonAbs

Generates Absyn.py, as well as the ParserDefs.py with the assistance of the character-to-unicode mapping.

Each value category in a grammar, such as "Stm" or "Exp", is used to generate abstract classes that are never instantiated, but only used to signify what category a member variable belongs to.

Each rule is then used to create both a dataclass for the abstract syntax, and to create a parser definition for PLY.

Let's look at the following example rule. Note that precedence levels of non-terminals have been normalized, meaning stripped of their precedence number once they are defined as member variables:

```
EAdd. Exp ::= Exp "+" Exp1
```

The rule generates the abstract syntax dataclass, where member variables with overlapping names, such as two "exp_" are enumerated:

```
@dataclass
```

```
class EAdd:
   exp_1: Exp
   exp_2: Exp
   ann_type: AnnType0 = field(default_factory=AnnType0)
```

This dataclass is then possible to match in a match-case, as illustrated in section 3.5 on the skeleton code. In order to produce the parsing definitions in **ParserDefs.py**, a rule's value category is also used.

The generated parser definition for the rule EAdd above, becomes with the additional character-to-unicode transformation:

```
def p_EAdd(p):
   """
   Exp : Exp S_43 Exp
   """
   p[0] = EAdd(p[1], p[3])
```

The parameter indices "p[1]" and "p[3]" are obtained from what places the corresponding non-terminal appear in the production.

If the rule is a coercion rule, where the rule names are "_" in the grammar file, the name for the parsing definitions is replaced with the target value category:

```
def p_Exp1(p):
   """
   Exp1 : Exp2
   """
   p[0] = p[1]
```

For categories that are defined with separators or terminators, parsing definition for lists are created. The produced parsing definitions depend on if the list uses a delimiter, if it is a termination delimiter or a separator delimiter, and if the list is allowed to be empty,

For example a list of expressions, [Exp], with a comma as separator:

```
separator Exp "," ;
```

Yields the following three parsing definitions after renaming the non-terminals, meaning "[Exp]" to "ListExp", and attaching a "Nil", "One", or "Cons" to their names:

```
def p_NilListExp(p):
   """
   ListExp :
   """
   p[0] = []
```

```
def p_OneListExp(p):
   """
   ListExp : Exp
   """
   p[0] = [p[1]]


def p_ConsListExp(p):
   """
   ListExp : Exp S_44 ListExp
   """
   p[0] = [p[1]] + p[3]
```

If the list was forced to be a non-empty list, the first definition, which allows empty lists, would not be created. Furthermore, the last line for concatenating lists is not optimal as it creates lots of new lists, but it is straightforward.

3.11 CFtoPythonPretty

The pretty printer and the skeleton code are generated in a similar way as how the classes are generated for the abstract syntax in the previous section. In the case of the pretty printer, the main difference is that the generation of the pretty printer uses mainly pre-written code for the rendering of the AST and of the linearized tree. Notably, the pretty printer instead of using the transformed character-to-unicode labels, which the parser utilizes, uses the original characters directly in the destructors to produce the original input.

3.12 CFtoSkelePy

The file CFtoSkelePy.hs generates the file skele.py, which is described in section 3.5 on Generated Skeleton code. There are two different kinds of matchers that are to be created, one with all in the same matcher, akin to Haskell's GADT option, and a separate set of matchers where every base category uses its own matcher.

For both matchers, any coercion or list rules are filtered out, yielding only the rules which would produce classes. For the first matcher, the entire set of the remaining rules are used, to create the destructors all in one matcher. For the second matcher, the rules are split up into their respective target non-terminals (Stm, Exp, etc..), creating separate matchers.

3.13 PyHelpers

The file PyHelpers.hs contains shared helper functions for the other modules, such as the renaming transformations for character-to-unicode or lists.

3.14 RegToFlex

The file RegToFelx.hs contains functions to convert BNFC's types like String, Integer, or user-created custom tokens defined within a grammar, to regular expressions. This file is primarily used by CFtoFlexPython.hs, and does not generate any parts of the frontend on its own. The file was originally stolen from the C backend due to its almost full similarity for the needed regular expressions - hence the naming "Flex" remains to attribute its origin.

3.15 Python.hs

The file Python.hs contains the entrypoint for the Python backend, which is called by BNFC. This module calls upon the modules described above in order to generate the frontend. The code for the lexer is produced first, so the parser can make use of the tokens generated for the lexer. This file is also responsible for creating the **genTest.py** programme, which consists mainly of predetermined python code. Aside from the list of entrypoints and the name of the grammar file, nothing from CF.hs is used.

3.16 Testsuites

The grammar used defines a small subset of C++, and is the same for all the labs in the book *Implementing Programming Languages* [18]. The example source-code files that are used were acquired from the PLT course [10]. Several testsuites were created to verify the implementation:

- A first testsuite was created to test whether the good examples are successfully parsed while the bad (incorrect) examples are rejected.
- A second testsuite was implemented for comparing the output of the Python frontend against the output of the C frontend, for both the AST and the linearized tree.
- A third testsuite was implemented for the typechecker, determining if the good examples typechecked, and that the bad examples did not typecheck. The typechecker additionally type-annotates the AST for the interpreter and compiler, and is therefore also used in the the interpreter and compiler below to pre-process the AST yielded from the parser.
- A fourth testsuite was created for the interpreter, to determine if the interpreter produced the correct output.
- A fifth testsuite was implemented to test whether the compiler produced working Jasmine programs, that produced the correct output when they are run with Java.
- A sixth testsuite, not automated but run by hand, consists of generating a Python frontend for each of the grammars: Alfa, C, C++, Cubilatt, BNFC's

define pragma, GF, Haskell-core, Java, Javalette, LBNF, OCL, and Prolog. The steps taken for each grammar consist firstly of verifying whether the frontend could be generated, and secondly that the generated frontend could be used to successfully parse example input of the corresponding grammar.

3.17 Example Python code for the labs

The book *Implementing Programming Languages* [18], describes several labs, three of which being the tasks of constructing a typechecker, an interpreter and a compiler. Below follows code snippets for each of them, to illustrate possible implementations.

3.17.1 The typechecker

In order to check whether a statement typechecks, all the operations of the expressions inside the statement should typecheck. For example, a conditional expression for a while-loop should have a boolean type, and otherwise throw an error. The statement inside the while-loop also needs to be typechecked, which is done in a new context, also known as scope. Creating or removing a scope does not need any arguments or to return any variable, as Python allows global variables.

```
def checkStm(s: Stm):
    match s:
        case SWhile(exp_, stm_, ann_type):
            if not checkExp(exp_, BOOL):
                raise TypeException("Type not correct for while condition")
            newScope()
            checkStm(stm_)
            deleteScope()
    ...
```

In order to type-annotate expressions in the AST, which is useful for the interpreter and the compiler, the ann_type member variable can be set when inferring the type of expressions. When looking up the type of a variable that has been declared earlier, the tree node can now directly be type annotated:

```
def inferExp(e: Exp):
    match e:
        case EId(id_, ann_type):
            type_ = lookupVar(id_)
            ann_type.s( type_ )
            return type_
    ...
```

3.17.2 The interpreter

The interpreter executes the AST, in order to produce some output. Below an and-expression is evaluated. If the first expression evaluates to be false, the second expression is not evaluated at all, which affects how the environment develops.

```
def evalExp(env, e):
    match e:
      case EAnd(exp_1, exp_2):
         (env, v1) = evalExp(env, exp_1)
         if v1 == False:
            return (env, False)
         else:
            (env, v2) = evalExp(env, exp_2)
            return (env, v2)
    ...
```

3.17.3 The compiler

The compiler produces Jasmine code from a type-annotated AST. Values are kept momentarily on a stack to be used in the near future, and variables can be loaded and saved to memory addresses.

The below example shows how loading the value of a variable can be compiled.

```
def compileExp(exp_):
    match exp_:
        case EId(id_, ann_type):
            # Id
            (addr, _) = lookupVar(id_)
            match ann_type.g():
                case (Type_int()|Type_bool()):
                    emit('iload ' + str(addr))
                    stackVal.inc()
                case Type_double():
                    emit('dload ' + str(addr))
                    stackVal.inc(2)
    ...
```

The memory address for the variable is retrieved, to then load the value on the memory address. By utilizing the type-annotation from the typechecker, the compiler knows which load operation to emit into the Jasmine code, and how much space the loaded value will take on the stack. The first load alternative, "iload" is used to load integers from some address, while the second "dload" is used to load a double from some address. As doubles are twice the size of ints in Jasmine, they take up twice the space on the stack.

4

Results

The results consists of the verification against the testsuites. Further discussion regarding the results resides in the following chapter discussions.

4.1 Verification

The verification of the implementation consists of the 6 testsuites described in method. The first 5 uses the PLT examples, and the last one uses the BNFC example grammars.

4.2 Verification using the PLT examples

The below subsections describes the usage of the testsuites from the PLT course to verify the python implementations of the typechecker, interpreter and compiler.

4.2.1 Testing the generated lexing and parsing files

The first testsuite tests whether the Python BNFC implementation can generate the Python files to tokenize and parse a small subset of the C++ programming language (from lab 1 in the *Programming Language Technology* course [10]). The testsuite consists of both "good" examples and "bad" examples. The good examples to test for true positives, and the bad ones to test for false positives.

Results:

- Good programs: passed 157 of 157 tests passed.
- Bad programs: 125 of 125 tests successfully failed.

4.2.2 Verification of the AST output and linearization

A testsuite was designed using the C files from lab 1 in the PLT course, to compare if the generated Python frontend produced the same AST and linearized tree as the C frontend.

The output is the same for 156 of 157 test cases, where the difference stems from that Python escapes white-space characters in strings when printing, such as tab or

newline, while the C frontend does not escape the white-space characters, resulting in tabs and line breaks in the output.

Results:

- Same output: (156/157)

4.2.3 Verification of the typechecker

The second lab involves implementing a typechecker. A testsuite using the test examples from the lab was constructed, testing whether the output of the typechecker was a pass or fail. The code examples are divided in four categories: Good, good with subtyping, bad and bad runtime. Only the "bad" category should fail - yielding an exception when attempting to typecheck one of the bad examples.

Results:

- True positives: (111/111)
- subs: True positives: (34/34)
- False positives: (0/68)
- Bad runtime positives: (4/4)

4.2.4 Verification of the interpreter

The testsuite for a Python implementation of the interpreter tests whether the interpreter given an input program of a subset of C, and optionally additional input that is intended to be used in "readInt" or "readDouble", produces the correct output. The produced output is compared with the **correct** output, which resides within a separate set of files in the testsuite that is only used for correction.

There are three categories of tests:

- The first "good" category contains various C programs.
- The second category contains C programs that involve subtyping (meaning implicit casting, for example adding $1 + 2.5$.
- The third category contains programs that should fail during runtime (for example using uninitialized variables, like declaring "int x" and using x without assigning any value to x)

Results:

- Good - same output: (111/111)
- Subtyping - same output: (34/34)
- Bad Runtime - finishes: (0/4)

4.2.5 Verification of the compiler

The testsuite for the compiler involves a series of steps:

- The Python implemented compiler is handed a C program, and produces a programme in the Jasmine assembly format.
- The Jasmin-program is afterwards compiled via Jasmin, into a java.class programme.
- Java is used to run the java.class file, producing some output.
- Finally the output produced is compared with the correct output, which resides within a separate set of files.

The results are as follows:

- Good - same output: (111/111)
- Subtyping - same output: (34/34)

4.3 Verification using the BNFC examples

Below is the results after manually testing each example grammar within the BNFC repository.

There were very few examples for each CF grammar however, and it is not verified whether the outputted abstract syntax tree of those are indeed correct.

4.3.1 Verification with the Python backend

Essentially 10/12 tests can generate the python files from the LBNF file, but only 9/10 can parse the examples.

- Alfa:

  Can't generate due to layout rules.

- C:

  Can generate both subset of C and C.

  - Subset of C example works.

  C examples work (except small.c for the three tested backends):

  - core.c
  - runtime.c

- C++:
  - Can generate.
  - Example works.

- Cubicaltt:
  - Can't generate due to missing layout.
- define:
  - Can generate.
  - Example works.
- GF:
  - Can generate.
  - Example works.
- Haskell-core:
  - Generation - Warning names not unique:

    Core.cf:7:1: Module

    Core.cf:62:1: Vbind

    Core.cf:71:1: ATbind

    This can be an error in some backends.
  - Examples work.
- Java:
  - Can generate.
  - No examples to test.
- Javalette:
  - Can generate.
  - Example works.
- LBNF:
  - Can generate.
  - Example works.
- OCL:
  - Can generate.
  - Example worked. but several warnings.
- Prolog:
  - Can generate.
  - Both small examples work.

4.3.2 Verification with the C backend

For comparison, the C backend:

- Generates 10/12 (fails the two grammars that use layout).
- Can run 9/10 examples (missing for Java).

4.3.3 Verification with the Haskell backend

Haskell backend:

- Generates 12/12 (but 1 with warnings)
- Can run 11/12 examples (missing for Java).

4.4 Benchmarking

Using the BNFC example grammar C.cf, along with the example file core.c, the execution time for lexing and parsing was compared for the Python frontend and the Java frontend.

The core.c file, which is over 7000 lines long, was concatenated to itself up to 25 times its original size, measuring up to 193 800 lines of code. Using the Unix command "time" to measure, the Java backend took about 1.5 seconds to both tokenize and parse, as well as print out the abstract syntax tree and the linearized tree. The trees were not verified however, so it is uncertain whether the parsing only parsed a chunk of the 200 thousand lines, and chopped off the rest.

The Python frontend on the other hand, took 51 seconds, to parse and print the AST and the linearized tree. Further repeated tests indicated:

- It took about 10.0 seconds to parse:
  - Of which 6 seconds due to the parsing.
  - Of which 4 seconds to build the AST using the sub-optimal concatenation of lists.
- It took about 40 seconds to print the linearization, perhaps because of a similar issue as with the list constructors in the parser, as the linearization attempts to create a list of all the characters from the AST, before rendering according to a C-style language.

For longer example files, where core.c was concatenated to 30 times its original size, the Java frontend failed to parse beyond some certain point.

4.5 PLY output for grammars with conflicts

There are reduce/reduce conflicts and shift/reduce conflicts.

4.5.1 Reduce/reduce conflicts

The following example is meant to produce an indirect reduce/reduce conflict:

```
entrypoints Exp;

EMul. Exp1 ::= Exp1 "*" Exp2 ;
EAdd. Exp ::= Exp "+" Exp1 ;

-- Reduce/reduce conflict
EVar. Exp2 ::= Var ;
EName. Var ::= Ident ;
EId. Exp2 ::= Ident ;

coercions Exp 2 ;
```

BNFC generates the Python frontend without any warnings. When attempting to run the generated test file the following output is exhibited.

```
$ echo "a + b" | python3.10 genTest.py
a + b

Building parser for entry: Exp
Generating LALR tables
WARNING: 4 reduce/reduce conflicts
WARNING: reduce/reduce conflict in state 5 resolved using rule (Var ->
    Ident)
WARNING: rejected rule (Exp2 -> Ident) in state 5
WARNING: Rule (Exp2 -> Ident) is never reduced

Trying to parse from Exp
Parse Successful!

[Abstract Syntax]
(EAdd (EVar (EName "a")) (EVar (EName "b")))

[Linearized Tree]
a + b
```

The warnings explicitly indicate the rules that produced the conflict. It may be difficult to decipher the referenced state 5 in the debug file parser.out, but it does indicate how the issue leads to the total number of 4 reduce/reduce conflicts.

Below is state 5 in the parser.out file. The exclamation marks indicate conflicts, and the unicode converted tokens S_42 and S_43 stands for "+" and "*" respectively.

```
state 5

    (5) Exp2 -> Ident .
    (4) Var -> Ident .
```

```
! reduce/reduce conflict for S_42 resolved using rule 4 (Var -> Ident .)
! reduce/reduce conflict for S_43 resolved using rule 4 (Var -> Ident .)
! reduce/reduce conflict for $end resolved using rule 4 (Var -> Ident .)
! reduce/reduce conflict for S_41 resolved using rule 4 (Var -> Ident .)
  S_42          reduce using rule 4 (Var -> Ident .)
  S_43          reduce using rule 4 (Var -> Ident .)
  $end          reduce using rule 4 (Var -> Ident .)
  S_41          reduce using rule 4 (Var -> Ident .)

! S_42          [ reduce using rule 5 (Exp2 -> Ident .) ]
! S_43          [ reduce using rule 5 (Exp2 -> Ident .) ]
! $end          [ reduce using rule 5 (Exp2 -> Ident .) ]
! S_41          [ reduce using rule 5 (Exp2 -> Ident .) ]
```

For conflicts with direct reduce/reduce conflicts, PLY is not able to build the parser at all. The below example shows a direct reduce/reduce conflict between the rules EName and EId:

```
EName. Exp2 ::= Ident ;
EId. Exp2 ::= Ident ;
```

4.5.2 Shift/reduce conflicts

By convention a shift/reduce conflict is resolved by shifting [18, p. 55].

This also holds for PLY, as we will see for the following example containing a shift/reduce conflict between the rules EAdd and EMul:

```
EMul. Exp ::= Exp "*" Exp ;
EAdd. Exp ::= Exp "+" Exp ;

EInt. Exp ::= Integer ;
```

The above grammar yields the following output when running the test program. Notice that we are made aware of 4 shift/reduce conflicts, and that instead of reducing "1 * 2" the parser has continued to shift in "+ 3". With rightmost derivations, "2 + 3" is reduced first, and afterwards the multiplication is reduced, which can be seen in the abstract syntax output.

```
$ echo "1 * 2 + 3" | python3.10 genTest.py
1 * 2 + 3

Building parser for entry: Exp
Generating LALR tables
WARNING: 4 shift/reduce conflicts

Trying to parse from Exp
```

```
Parse Successful!

[Abstract Syntax]
(EMul (EInt 1) (EAdd (EInt 2) (EInt 3)))

[Linearized Tree]
1 * 2 + 3
```

The second time the test program is run, since the parser is re-using the cached parsing table in parsetab.py, no warning is exhibited.

5

Discussion

5.1 The development process

Regarding the development, it was mainly an iterative process. Indeed, the easiest way was to first create prototype functions that extracted the necessary information and produced the desired output, in the form of instructions for PLY. There were many similar functions, which could be refactored into common ones, only changing some specific behaviour, but that refactoring has been avoided in order to prevent side-effects that may be difficult to debug. Hence, the functions to write classes, coercions, and lists are all separate, even though they could have been incorporated into a general function. Once the behaviour of every piecewise function has been tested and verified, these functions could be compared for the common parts, and thereafter refactored into more general functions. (not yet done)

5.2 Issues and caveats

5.2.1 Mutliple entrypoints

PLY generates a parser for every entrypoint, so when multiple entrypoints are used, not all categories may be reachable from the entrypoint. This generally generates multiple warnings for each of those entrypoints.

The example below shows the behaviour. Note that there are no warnings after building a parser for "Prog", but when building a parser for "Stm". The warnings further escalate for "Exp". The warnings are printed to stderr and generate no exceptions. After finishing building the parsers, parsing from "Prog" yields a successful parsing.

```
Building parser for entry: Prog
Building parser for entry: Stm
WARNING: .../ParsingDefs.py:31: Rule 'Prog' defined, but not used
WARNING: There is 1 unused rule
WARNING: Symbol 'Prog' is unreachable
Building parser for entry: Exp
WARNING: /ParsingDefs.py:31: Rule 'Prog' defined, but not used
WARNING: There is 1 unused rule
WARNING: Symbol 'Prog' is unreachable
```

```
WARNING: Symbol 'Typ' is unreachable
WARNING: Symbol 'ListStm' is unreachable
WARNING: Symbol 'Stm' is unreachable

Trying to parse from Prog
Parse Successful!
```

One possibility to solve this is to create a dependency tree for all the categories, and omit all parsing methods using unreachable categories when building for some specific entrypoint. Another possibility is to generate an additional set of rules to the grammar, of which purpose it is to contribute with the following effect:

```
EntryProg. Start ::= Prog
EntryStm. Start ::= Stm
EntryExp. Start ::= Exp
```

Then it is enough to generate a parser with the entrypoint set to "Start", instead of having multiple entrypoints. However, innovative naming schemes would need to be used, in order to prevent overlapping of starting rule names or of the starting category, with existing rule names or categories respectively.

5.2.2 Conflict warnings

In parsing there can exist reduce/reduce and shift/reduce conflicts. When building a parser using the other backends, the user is notified about such conflicts. That does not appear to be the case for PLY. When setting the debug flag when building the parser in PLY, an additional parser.out file is created that contains cached parser tables as well as other info.

The minimal grammar with shift/reduce conflicts:

```
EAdd. Exp ::= Exp "+" Exp ;
EMul. Exp ::= Exp "*" Exp ;
EInt. Exp ::= Integer ;
```

This yields the following parsing rules in the *parsing.out* file:

```
Rule 0    S -> Exp
Rule 1    Exp -> Exp S_43 Exp
Rule 2    Exp -> Exp S_42 Exp
Rule 3    Exp -> Integer
```

In the example below we are starting from state 0, which is the entrypoint. Since there are two possible tokens that could be in the tokenized input, being "Exp" or "Integer", there are two possible actions to take. Either shifting in an integer and go to state 2, or an "Exp" and go to state 1:

```
state 0
```

```
   (0) S -> . Exp
   (1) Exp -> . Exp S_43 Exp
   (2) Exp -> . Exp S_42 Exp
   (3) Exp -> . Integer

   Integer       shift and go to state 2

   Exp           shift and go to state 1
```

Conflicts are signified by the parser if there are more than one possible actions, for a given token. This may be difficult to interpret - especially with the character-to-unicode transformation.

```
state 5

   (1) Exp -> Exp S_43 Exp .
   (1) Exp -> Exp . S_43 Exp
   (2) Exp -> Exp . S_42 Exp

 ! shift/reduce conflict for S_43 resolved as shift
 ! shift/reduce conflict for S_42 resolved as shift
   $end          reduce using rule 1 (Exp -> Exp S_43 Exp .)
   S_43          shift and go to state 3
   S_42          shift and go to state 4

 ! S_43          [ reduce using rule 1 (Exp -> Exp S_43 Exp .) ]
 ! S_42          [ reduce using rule 1 (Exp -> Exp S_43 Exp .) ]
```

5.2.3 No layout

Currently there exist layout pragmas in BNFC, that are not incorporated into Python backend implementation. It is the case that the layout pragmas are only used by the Haskell backend, but it would be beneficial if the layout pragmas could be added in order to parse grammars that use layout syntax.

5.2.4 Defining coercions explicitly instead of using the coercion pragma

Currently the pretty printers for the different frontends exhibit an odd behaviour when the priority of the coercions are defined in a nonconventional order.

One example of how Calc.cf can be defined in a conventional manner, with "Exp" being the lowest precedence and "Exp2" the highest, is:

```
EAdd. Exp ::= Exp "+" Exp1;
EInt. Exp2 ::= Integer ;
```

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

Which yields the expected linearization:

```
$ echo "1 + 2" | ./TestCalc

Parse Successful!

[Abstract Syntax]
(EAdd (EInt 1) (EInt 2))

[Linearized Tree]
1 + 2
```

However - if we manually exchange all "Exp1" with "Exp2" - the program should still remain the same. All of our changes should only change the semantics after all.

Here is the grammar after we exchange every "Exp1" with a "Exp2" and vice versa.

```
EAdd. Exp ::= Exp "+" Exp2 ;
EInt. Exp1 ::= Integer ;

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

The linearized output is different in this case however:

```
$ echo "1 + 2" | ./TestCalc

Parse Successful!

[Abstract Syntax]
(EAdd (EInt 1) (EInt 2))

[Linearized Tree]
1 + (2)
```

A small but telling difference. What happens is that rule EInt (which is Exp1) now is "expected" to have to loop around, since EAdd in its second parameter uses an Exp2 now instead of an Exp1.

This is however a mistake! Because we changed the priority between Exp1 and Exp2. The pretty printers (at least in the C, Haskell and Python backends) assumes that the priority of the coercions is defined as normal.

5.2.5 The linearization always uses parentheses

Additionally using any other character, such as brackets instead of parentheses, does not affect the output of the linearization. The linearization still uses parentheses to loop around precedence in coercions.

The BNF below is the normal Calc, but with parentheses replaced with brackets.

```
EAdd. Exp ::= Exp "+" Exp1;
EInt. Exp2 ::= Integer ;

_. Exp2 ::= "<" Exp ">" ;
_. Exp1 ::= Exp2 ;
_. Exp ::= Exp1 ;
```

This yields the output:

```
$ echo "1 + <2 + 3>" | ./TestCalc

Parse Successful!

[Abstract Syntax]
(EAdd (EInt 1) (EAdd (EInt 2) (EInt 3)))

[Linearized Tree]
1 + (2 + 3)
```

5.2.6 Using non-special characters when defining rules

If one defines the production of rules with characters where there usually should be special characters, for example replacing the parantheses with the letter "a":

```
Add. Exp ::= Exp "+" Exp1
Int. Exp1 ::= Integer

_. Exp1 ::= "a" Exp "a"
_. Exp ::= Exp1
```

PLY is not able to parse an input such as "a1+2a + 3". With spaces inbetween it is okay, like "a 1+2 a + 3". Similar behaviour can be seen in the Haskell backend, while the C backend successfully parses the first mentioned input.

It appears that the character "a" gets placed into the reserved keywords, like "int", by the inner backend of BNFC while it reads the CF file above. Reserved keywords are meant to not be parsed concatenated with a following expression, like "int1+2". This indicates that the problem may not lie with the code-generating backends, though it is puzzling that the C backend behaves differently.

5.2.7 Multiple separators for the same category

In the generated Haskell- and C frontend it is possible to define multiple separators for the same category.

```
EAdd. Exp ::= Exp "+" Exp1 ;
EMul. Exp1 ::= Exp1 "*" Exp2 ;
Arr1. Exp2 ::= "[" [Exp1] "]";
EInt. Exp2 ::= Integer ;

separator Exp1 "," ;
separator Exp1 ";" ;

coercions Exp 2;
```

This yields the C frontend output, where both "," and ";" are possible separators for Exp1:

```
$ echo "[1, 2; 3] + 2" | ./TestCalc

Parse Successful!

[Abstract Syntax]
(EAdd (Arr1 [(EInt 1), (EInt 2), (EInt 3)]) (EInt 2))

[Linearized Tree]
[1, 2, 3] + 2
```

For the Python frontend it generates overlapping parser methods however, yielding an error when attempting to running the python programme due to an oversight, where the possibility was not reconsidered to have multiple separators for the same category.

6

Conclusion

The Python backend of BNFC was implemented while relying on PLY for the frontend. There are some issues as mentioned in the discussion chapter, where currently the character-to-unicode transformation makes it difficult to interpret the debug-file is the most pressing one. In order for users to use the Python backend with PLY it is imperative that conflicts in a grammar are well presented. This might be solved before the project is done - perhaps by translating the debug parsetab.py file using a pretty printer, that replaces all the unicode expressions and updates the indentation.

It was possible however to implement the typechecker, interpreter and compiler using the Python frontend. It was also able to produce the same abstract syntax trees and linearization tree as the C frontend, which is some merit for the Python frontend's usability.

Future possible additions for the Python backend may be to implement the layout syntax, or perhaps generate more suitable code for PLY, such as utilizing the optimization mode, or clever ways of building parsers for several entrypoints.

References

[1] Antlr. https://www.antlr.org/, Retrieved 2024-05-18.

[2] Gold parsing system. http://www.goldparser.org/, Retrieved 2024-05-18.

[3] Lark. https://github.com/lark-parser/lark, Retrieved 2024-05-18.

[4] Plyplus. https://github.com/erezsh/plyplus, Retrieved 2024-05-18.

[5] Simple parser. https://github.com/CDSoft/sp?tab=readme-ov-file, Retrieved 2024-05-18.

[6] Sly. https://github.com/dabeaz/sly, Retrieved 2024-05-18.

[7] Jeffrey D. Ullman Alfred V.Aho, Ravi Sethi. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[8] David M. Beazley. Ply (software) official documentation. https://www.dabeaz.com/ply/ply.html, Retrieved 2024-05-15.

[9] Guido van Rossum Brandt Bucher. Structural pattern matching. https://peps.python.org/pep-0634/.

[10] Chalmers. Programming language technology. https://www.chalmers.se/en/education/your-studies/find-course-and-programme-syllabi/course-syllabus/DAT151/?acYear=2023%2F2024.

[11] James Alan Farrell. Anatomy of a compiler. http://www.cs.man.ac.uk/~pjj/farrell/comp3.html, 1995.

[12] Centre for Language Technology. The bnf converter. http://bnfc.digitalgrammars.com/.

[13] Python Software Foundation. inspect. https://docs.python.org/3/library/inspect.html, Retrieved 2024-05-18.

[14] Daniel Reynaud Jonathan Meyer. Jasmin home page. https://jasmin.sourceforge.net/.

[15] Linda Torczon Keith D. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers, 2004.

[16] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[17] Aarne Ranta. Python backend to bnf converter – slide during the master thesis mingle in 2023.

[18] Aarne Ranta. *Implementing Programming Languages*, volume 16. College Publications, 2012.

[19] Eric V. Smith. Data classes. `https://peps.python.org/pep-0557/`.

[20] Lowell D. Thomas. Sabnf. `https://sabnf.com/`, Retrieved 2024-05-18.

[21] Wikipedia. Comparison of parser generators. `https://en.wikipedia.org/wiki/Comparison_of_parser_generators`, Retrieved 2024-05-18.