An Appendix to Pierre Nugues's Python Book

Aarne Ranta
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
aarne.ranta@cse.gu.se

Abstract

The third edition of Pierre Nugues's book uses Python to introduce several state-of-the-art techniques in data-driven NLP. This paper describes a few examples of symbolic methods using the Python bindings of Grammatical Framework, GF.

1 Introduction

Pierre Nugues gave me a draft of his book *An Introduction to Languge Processing in Perl and Prolog* (Nugues, 2006) before its publication. I liked the book very much. It covered much of the same ground as the most popular textbook in the field, (Jurafsky and Martin, 2000). But it was shorter (althouch still long) and more precise about the mathematical foundations. It also had much more to say about other languages than English. I chose it as the recommended book in my teaching.

One thing about the book that I did not use were the code examples. Perl and Prolog were still the state of the art at the time, but I preferred Haskell, a functional programming language with static typing and powerful abstraction mechanisms. Fortunately, combining Nugues's book with programming in Haskell was not a problem, because all the important concepts were explained independently of the code examples.

The third edition of the book, from 2024, changed the title: *Python for Natural Language Processing* (Nugues, 2024). Also much of the content has changed: it is now entirely about statistics and machine learning, and it covers the latest techniques such as large language models and transformers. The mathematics behind these methods is still carefully explained in separation from the code, but the code is very useful as it uses state-of-the art Python libraries that enable realistic coding exercises and experiments. I am very much looking forward to studying this book in more depth to learn new things.

Haskell is still my preferred language for research projects, but Python is easier to deploy in areas where it has library support. Python has also become a lingua franca for students and researchers in many areas, which makes it the language of choice in projects that want to attract community contributions. Last but not least, Python has over the years adopted features of functional programming, often inspired by Haskell: lambda abstractions, higher-order functions, comprehensions, and structural pattern matching.

Even at my university, Python has become the programming language number one in teaching. In the last five years, I have taught several editions of three different courses that use Python, to thousands of students. In this connection, I have written several pieces of code where I think Python works well. One of these tasks is among those that Pierre has left out from the third edition: syntax. Hence a humble — or actually rather pretentious — proposal arises to complement Pierre's book with some grammar-based applications in Python. Complete code examples can be found in `https://github.com/GrammaticalFramework/comp-syntax-gu-mlt/tree/main/python`.

2 Background: GF

Perhaps the most widely used tool for syntax in Python is NLTK, Natural Language ToolKit (Bird et al., 2009). Nugues (2024) mentions NLTK three times, about other things than syntax. NLTK's CFG class supports context-free grammars, whereas the CCG class supports combinatory categorial grammars.

Grammatical Framword (GF, Ranta, 2011) is a grammar formalism based on a Logical Frame-

From Electrophoresis to Wikidata: Festschrift in honor of Pierre Nugues, pages 25–29.
Published by University of Tartu Library, May 2025.

work (LF), which is a generic name for computer systems based on constructive type theory (Martin-Löf, 1984). GF uses LF to define **abstract syntax**, which is a free algebra of **abstract syntax trees**. It extends LF with a layer of **concrete syntax**, which is a mapping from abstract syntax trees to strings in some actual language. These mappings are by design **reversible**, which makes GF grammars usable for both **linearization** (from trees to strings) and **parsing** (from strings to trees).

An abstract syntax in GF can be equipped with several concrete syntaxes, which results in a **multilingual grammar**. Combining parsing with one concrete syntax and linearization with another one results in **translation** betwee the languages. To make this possible, GF has an expressive power slightly above CCG but still in the mildly context-sensitive class equivalent to PMCFG (Parallel Multiple Context-Free Grammar, Seki et al., 1991; Ljunglöf, 2004).

To give an example, consider the following rules of context-free grammar. They are written in the BNF (Backus-Naur Form) notation, which is also recognized by GF as a special case of full GF grammars. Each rule can be given a name; otherwise, GF creates names automatically, but they are clumsier to use.

```
Pred.  S  ::= NP VP
Compl. VP ::= TV NP
```

This grammar is in GF separated into an abstract and a concrete syntax that look as follows:

```
-- abstract
fun Pred : NP -> VP -> S
fun Compl : TV -> NP -> VP

-- concrete
lin Pred np vp = np ++ vp
lin Compl tv np = tv ++ np
```

This grammar corresponds to the SVO word order (Subject-Verb-Object). However, just changing one rule in the concrete syntax defines the SOV order:

```
lin Compl tv np = np ++ tv
```

The VSO order requires a more radical change: splitting the VP into a **record** with a verb part and an object part:

```
lin Pred np vp =
  vp.verb ++ np ++ vp.obj
lin Compl tv np =
  {verb = tv ; obj = np}
```

This is an example where the "multiple" part of PMCFG is used: instead of a single string, a VP consists of two strings, which can be taken apart as a **discontinuous constituent**. Another example is **inflection tables**. For instance, French verbs are modelled by tables that contain 53 strings:

```
table {
  VF Ind Pres Sg P1 => "aime" ;
  VF Ind Pres Sg P2 => "aimes" ;
  ...
  VF PastPart Pl Fem => "aimées"
  }
```

This table can be used as the linearization of the same abstract object as English "love", which is a table with just five forms.

Even with the expressive power of a formalism such as GF, writing grammars requires both time and expertise. The following assets help solve these problems: a **Resource Grammar Library** (RGL), which makes the grammar rules of over 40 languages usable via a high-level API (Application Programming Interface), and **embedded grammars**, program libraries that enable the access to GF grammars in main-stream programming languages such as C, Java, and Python. Embedded grammars use a runtime format of GF, called PGF (Portable Grammar Format, Angelov et al., 2009), which can be manipulated with a runtime system written in C (Angelov, 2011). The C runtime can be imported in other programming languages via their foreign function interfaces — a technique that is extensively used in Python.

Another direction in GF has been compound systems with data-driven techniques such as neural dependency parsing. Some of these projects are outlined in (Ranta et al., 2020).

3 The PGF Library in Python

To enable PGF in Python, you just need to install the pgf library:

```
pip3 install pgf
```

To test this, one can get started with a ready-made PGF file from `https://www.grammaticalframework.org/~aarne/ResourceDemo.pgf.gz`. Uncompress it and create a file `trans.py` with the following Python script, which is a simple translator loop receiving input in English and

converting it to all other languages. We have left out all "unnecessary" parts such as error handling.

```
import pgf

gr = pgf.readPGF('ResourceDemo.pgf')
inlang = gr.languages['ResourceDemoEng']

while True:
    string = input('> ')
    parses = inlang.parse(string)
    _, tree = parses.__next__()
    print(tree)
    for _, cnc in gr.languages.items():
        print(cnc.linearize(tree))
```

The script loads a PGF file with the function `pgf.readPGF()`. After that, it sets the input language to English; the keys in the `languages` dictionary are concrete syntax module names, and the values are the concrete syntaxes themselves. An infinite loop reads user input that it tries to parse in English, with the `parse()` method of the concrete syntax. A successful parse returns an iterator, here `parses`, which consists of pairs of probabilities and abstract syntax trees. The parses are returned lazily in the order of decreasing probability (which can be defined for each GF grammar separately). Here, we just print the linearizations of the first tree in each language using its `linearize()` method. Running the script looks as follows:

```
$ python3 trans.py
> this grammar knows forty languages
...
hierdie grammatika ken veertig tale
aquesta gramàtica sap quaranta llengües
denne grammatik kender fyrre sprog
see grammatika tunnab nelikümmend keelt
gramatika honek berrogei hizkuntzak ditu
tämä kielioppi tuntee neljäkymmentä kieltä
cette grammaire connaît quarante langues
diese Grammatik kennt vierzig Sprachen
...
```

Obviously, many more things can be done with the few methods shown in the above script. The full API of the PGF library is given in `https://www.grammaticalframework.org/doc/runtime-api.html#python`, explaining functionalities such as morphological analysis and tree visualization. All of these are accessible from Python without writing any GF code and even without installing the GF compiler. The compiler is, however, needed if you want to build your own .pgf files. The compiler is run on a set of .gf files as follows:

```
gf -make MyEng.gf MyFre.gf ...
```

with the list of all those concrete syntaxes that implement the abstract syntax `My`. The resulting file `My.pgf` can be used in Python programs in the way shown in the script above.

Translation is the most straightforward application of GF grammars. But it is no longer as popular as it used to be, after all advances in neural machine translation. Two application that can more clearly benefit from the underlying tree structures are **semantics** and **natural language generation**. Let us briefly cover those applications in the following sections.

4 Semantics

A widely used, traditional way of defining semantics is by **pattern matching on abstract syntax trees** (Van Eijck and Unger, 2010). It originates in Montague semantics (Montague, 1974), where it works on syntactic structures similar to those used in the GF Resource Grammar Library. Such general syntax-based semantics is probably not a very common NLP task these days. But the same principles can be applied for more specific tasks, such as query languages, which compute answers to questions by similar semantic rules.

Using a GF grammar for semantics makes it possible to share the semantics among different concrete languages, because the semantic functions operate on the abstract syntax. Let us illustrate this with a fragment of Montague's famous PTQ fragment (Proper Treatment of Quantification in Ordinary English). The most important rules are the ones having to do with predication and complementation, especially in the presence of quantifiers. We can cover an interesting part of Montague's PTQ with the following abstract syntax functions in GF:

```
Pred  : NP -> VP -> S
Compl : TV -> NP -> VP
Intr  : IV -> VP
Every : CN -> NP
Named : PN -> NP
```

For each of the involved categories, there is a **semantic type**, which in a functional notation gives the following types of semantic interpretation functions:

```
semS  : S -> bool
semPN : PN -> ind
semVP : VP -> ind -> bool
semIV : IV -> ind -> bool
semTV : TV -> ind -> ind -> bool
semCN : CN -> set
semNP : NP -> (ind -> bool) -> bool
```

Noun phrase (NP) rules are of special interest: because NP includes both proper names (PN) and quantified phrases (Every), its semantic type cannot be just `ind` of individuals. Thus, in the rule for `Pred`, it is the (semantics of) the NP that is applied to VP, not the other way round. This is natural for quantifier phrases but means that PN has to be "raised" into a function of expected type; this was one of the most famous tricks in Montague's semantics.

In Python, the semantic types can be expressed as type hints of semantic functions, which operate recursively on abstract syntax trees for each category, as shown in the code below. The only new function needed from the PGF library is the method `unpack()`, which returns a pair of a function and a list of arguments. By using the `match` statements of Python, we can write pattern matching very much like in Haskell (Van Eijck and Unger, 2010).

```
def semNP(tree: pgf.Expr) ->
      Callable[
        [Callable [[ind], bool]],
        bool]:
    match tree.unpack():
      case ('Named', [pn]):
        return lambda P: P (semPN(pn))
      case ('Every', [cn]):
        return lambda P: all(P(x)
                    for x in semCN(cn))

def semS(tree: pgf.Expr) -> bool:
    match tree.unpack():
        case ('Pred', [np, vp]):
            return semNP(np)(semVP(vp))
```

5 Natural Language Generation

Natural Language Generation (NLG) is probably the most popular application of GF grammars. Rule-based NLG in the style of Reiter and Dale (2000) is still an efficient and technique method to convert data into text. It does not "hallucinate", and it can be implemented simultaneously for a large set of languages via abstract syntax, guaranteeing that all generated languages express exactly the same content.

Abstract Wikipedia (Vrandečić, 2021; Ranta, 2023; Angelov et al., 2025) is an ongoing project where GF and Python are used for NLG. Abstract Wikipedia is based on Wikidata, a fact database developed to support Wikipedia (Vrandečić and Krötzsch, 2014). Queries to this database can be stored in JSON files, which are easy to manipulate in Python. For example, Wikidata about cities can contain entries of the following kind:

```
"Q2167": {
  "labels": {
      "en": "Lund",
     "fr": "Lund",  ...
      },
  "country": "Q34",
  "population": "98308",
  "university": "Q218506",
  "domain": "city"
  }
```

The identifiers starting with "Q" are unique identifiers of Wikidata objects, also usable as abstract syntax functions in GF. Every object may have "labels" in different languages, and concrete syntax linearizations can be derived from them. Any of these pieces of information can be missing, which is represented as a `null` value in the object. The following code generates one-line descriptions of cities from `city_data`.

```
def city_descr(city, prop=None):
    country = city_data[city][country]
    if country and property:
      tree = f'Descr prop (Loc {country})'
    elif country:
      tree = f'Descr city (Loc {country})'
    else:
      tree = 'city'
    return pgf.readExpr(tree)
```

Descriptions are generated as abstract syntax trees, which can be linearized to every supported language. A simple way to build such trees is to use f-strings for them, with slots for the country of location and the most important properties. This produces strings, from which trees can be constructed with the `readExpr()` function. The resulting description of Lund says that it is a university town in Sweden, because both `country` and `university` fields have non-null values:

```
Descr UniversityTown (Loc Q34)
```

This tree linearizes to strings such as

a university town in Sweden
une ville universitaire en Suède
yliopistokaupunki Ruotsissa

Such descriptions can typically be found as the first sentences of Wikipedia articles. The generation of complete articles is discussed in Angelov et al. (2025).

References

K. Angelov. 2011. *The Mechanics of the Grammatical Framework*. Ph.D. thesis, Chalmers University of Technology.

K. Angelov, B. Bringert, and A. Ranta. 2009. PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information*, 19:201–228.

Krasimir Angelov, Andrea Carrión del Fresno, Ekaterina Voloshina, and Aarne Ranta. 2025. Leveraging grammatical framework and wordnet for natural language generation from wikidata. In *Distributed Computing and Artificial Intelligence, Special Sessions I, 21st International Conference*, pages 173–184, Cham. Springer Nature Switzerland.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit*. O'Reilly.

D. Jurafsky and J. Martin. 2000. *Speech and Language Processing*. Prentice Hall.

P. Ljunglöf. 2004. *The Expressivity and Complexity of Grammatical Framework*. Ph.D. thesis, Dept. of Computing Science, Chalmers University of Technology and Gothenburg University.

P. Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis, Napoli.

R. Montague. 1974. *Formal Philosophy*. Yale University Press, New Haven. Collected papers edited by Richmond Thomason.

Pierre Nugues. 2006. *An Introduction to Languge Processing in Perl and Prolog*. Springer.

Pierre Nugues. 2024. *Python for Natural Language Processing*. Springer.

Aarne Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford.

Aarne Ranta. 2023. Multilingual text generation for abstract wikipedia in grammatical framework: Prospects and challenges. In *Logic and Algorithms in Computational Linguistics 2021 (LACompLing2021)*, pages 125–149, Cham. Springer Nature Switzerland.

Aarne Ranta, Krasimir Angelov, Normunds Gruzitis, and Prasanth Kolachina. 2020. Abstract Syntax as Interlingua: Scaling Up the Grammatical Framework from Controlled Languages to Robust Pipelines. *Computational Linguistics*, 46(2):425–486.

Ehud Reiter and Robert Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.

H. Seki, T. Matsumura, M. Fujii, and T. Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229.

Jan Van Eijck and Christina Unger. 2010. *Computational semantics with functional programming*. Cambridge University Press.

Denny Vrandečić. 2021. Building a Multilingual Wikipedia. *Communications of the ACM*, 64(4):38–41.

Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85.