Natural Language Generation with the Grammatical Framework

Aarne Ranta

July 28, 2021

Contents

Chapter 1

Introduction

1.1 What is NLG?

Natural Language Generation (NLG) is a programming task where data is converted to natural language. An example is a table that lists countries and their populations:

country	**population**
Argentina	44938712
United States	331449281

From this data, a very simple NLG system can produce the sentences

> *The population of Argentina is 44938712.*
> *The population of United States is 331449281.*

These sentences essentially list the individual facts in the data, row by row from the table. This simple system can still be useful, for instance as a device to feed the data to speech synthesis.

A step beyond the simplest conceivable system is not just to list the data point by point, but also express interesting observations or summaries based on the data. Thus a slightly more advanced NLG system could also produce

> *United States has over seven times more inhabitants than Argentina.*

which combines two facts in a hopefully interesting way. Methods for selecting what to say about a data are traditionally a central interest of NLG research.

After selecting *what* to say, an NLG system has to define *how* to say it. The simplest method is to use **templates**, which are sentences or texts with "holes" to which the data is inserted. Thus a template for populations of countries might be

```
The population of _ is _.
```

and for comparisons of two countries,

```
_ has _ times more inhabitants than _.
```

Sooner or later, the template method may turn out insufficient, because the words belonging to the template may be different for different data values. A typical example is the number of the noun, which should be as in

You have 2 new messages.
You have 1 new message.

Witnessing a wide-spread use of templates, it is still very common to see examples such as

You have 1 new messages.
You have 1 new message(s).

The problem is often harder for other languages than English. Even the first example, populations of countries, which works fine in English, creates a problem in languages where country names have to be inflected. Thus for instance in Swedish, we need the genitive form of the country. A simple-minded template would add a genitive *s* to the name:

```
_s befolkning är _.
```

This would work for many countries, but not for those whose name already ends with an *s*, such as *Mauritius*: no *s* would then be added. This problem becomes worse in languages like Finnish, where country names are inflected in intricate ways.

To avoid the template problem, an NLG system must be aware of **grammar** so that it can select proper forms of words. Even the order of words may have to vary as a function of what data is described. Building in correct grammar into NLG is a nontrivial task, but it can be helped by software

tools and libraries. In this document, we will introduce the solution provided by Grammatical Framework (GF), which has been used in NLG for over 40 languages.

Another recent trend in NLG is **language models**, such as BERT and GPT-3; the idea was originally conceived by Shannon in the 1940. They are algorithms that can produce text automatically by continuing a given text in the "most probable" way, based on a vast collection of already seen texts. While these systems can produce impressive and natural-looking texts, a closer inspection often reveals them to be nonsense. The problem is that language models only model the language, without relation to data outside the language. Another problem is that models of appropriate size are very costly to build (in terms of money and even of climate impact). What is more, many languages of interest simply do not have enough data available to build such models.

We will consider language models as a way to help GF-based NLG to find the most natural renderings of data. But we will stress the utility of rule-based, fully controllable methods to generate language that is faithful to the content we want NLG to express.

1.2 What is NLG good for?

NLG is used for text production because of its **low price** and **high speed**. Once an NLG system for sport events or financial data or weather reports is in place, it can produce new documents for free and in seconds, as opposed to human reporters working against payment for hours. Such a system can also guarantee **correctness**, as it is free from accidental errors in converting data to text. A related aspect is **uniformity**: NLG can create documents in a specific format without any deviations, which means that if the documents are used for instance for decision making, their readers can easily find the information they want.

With uniformity comes also the risk of **monotony**, which means that the texts might not be entertaining to read. This affects the cost/benefit balance of NLG, because readers might not be willing to pay as much for NLG-produced texts as for natural ones. The initial **cost** of NLG can of course be high in itself, since building a system is expert work that pays off only when the volume of produced texts is high enough; this has been witnessed by companies that have investigated potential uses of NLG in their business.

The cost can be particularly high for **languages other than English**, where building the system can be more demanding (because of complex grammar and lack of existing resources). Most existing NLG targets English, and so for instance languages of developing countries — whose speakers might need it the most because of lack of human-written text — seldom have access to NLG systems.

The balance between advantages and problems in NLG changes if we move from monolingual to **multilingual** NLG. Think about a technology with the following characteristics:

- it can generate many languages from the same data;
- adding a new language is much cheaper than starting with the first language.

Such an NLG system could be used for the **dissemination of information** simultaneously to many languages. It would enable

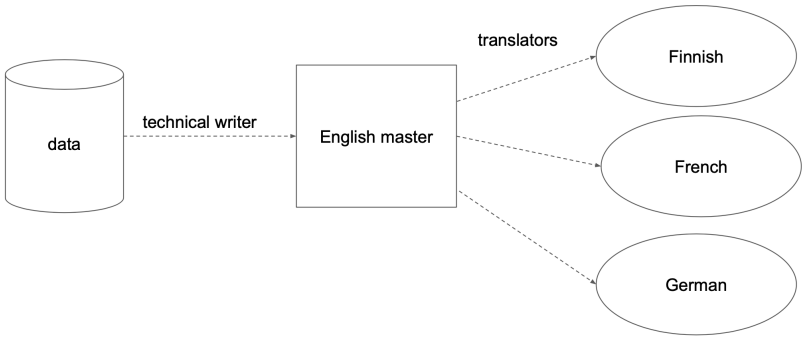
- authorities to keep all linguistic groups in a region informed in a uniform way,
- companies to market their products at many markets simultaneously,
- ideal organizations to disseminate information world-wide.

This kind of system is precisely the target of this tutorial. In the 23 years of GF, previous experience has been gained from numerous NLG applications, for example,

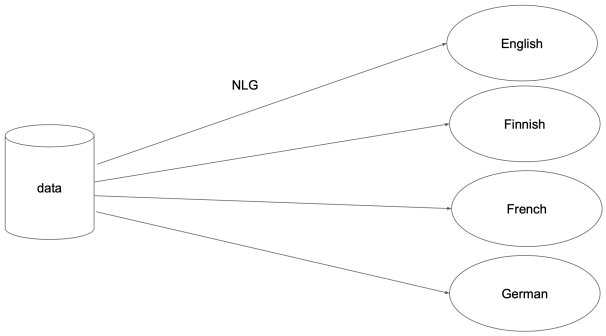
- mathematics education material (WebALT),
- descriptions of paintings in museums (MOLTO),
- healthcare documents in underresourced languages (Meraka Institute, South Africa),
- product descriptions in e-commerce (Digital Grammars, Textual),
- descriptions of buildings from accessibility perspective (Digital Grammars, Tillgänglighetsdatabasen),
- software and system specifications (KeY, Altran).

From the users' point of view, multilingual NLG is often introduced as a replacement of **translation**. The starting point is a workflow where

- a technical writer (or a copywriter) produces a **master document** (usually in English) based on some data,
- translators translate the master document into different languages.



To reduce the cost and improve the speed and consistency of this completely manual process, an often used method is **machine translation**: only the master document needs a human author, whereas the translations are done automatically. From the NLG perspective, this is a very roundabout procedure: why write an English document to describe the data and translate the English text, when one could generate different languages (including English) directly from the data?



Machine translation can in fact be seen as an application, or extension, of NLG. NLG goes from data to texts. Translation adds to this a component that goes from text to data. The data is, by definition, something **structured** and **formal**. Unlike natural language, it is **unambiguous**. The NLG part of translation (from data to text) is therefore **deterministic**. The text-to-data part, in contrast, requires **ambiguity resolution**, which is the most demanding part of translation: no algorithm has so far succeeded to perform it in a completely reliable way. Therefore NLG can be not only a cheaper solution to multilingual dissemination, but also yield better quality.

1.2.1 Abstract Wikipedia

The Wikipedia is an on-line freely available encyclopedia on the internet. It has been built by millions of authors, together producing over 20 million articles in over 300 languages. Those languages, however, vary immensely in coverage and quality:

- English has much more articles than other languages,
- many articles available in other languages are not available in English,
- the quality and length of corresponding articles in different languages varies.

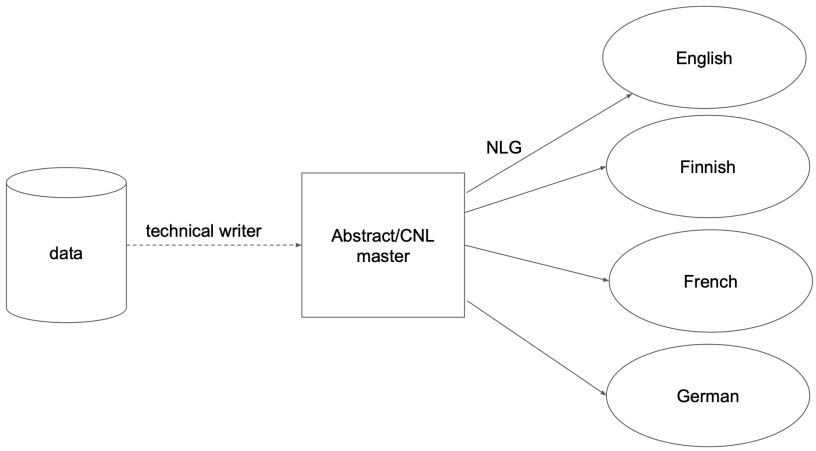
Abstract Wikipedia is an initiative from the Wikimedia foundation to solve this problem by the use of NLG. The basis of NLG is data stored in **Wikidata**, which is a database containing basic facts (in the form of RDF triples, such as (France,capital,Paris)). Wikidata is already used for generating some parts of Wikipedia articles, for instance, in the fact boxes. The vision of Abstract Wikipedia is to generalize this in terms of both language coverage and richer content.

The following tutorial is structured with the Abstract Wikipedia in mind. We proceed bottom-up as follows:

- atomic facts: direct renderings of RDF triples,
- more idiomatic ways to express atomic facts,
- texts consisting of combining facts in more fluent ways than just listing them,
- selecting facts and inferring interesting generalizations from them.

All of this can be made into a fully automatic process, for which we will show code that combines GF and Python. However, the last step, selecting interesting content, cannot be fully automated. Therefore the ultimate outcome is an **interactive NLG system**, where a human is in the loop filtering the data and restructuring it in the way that can result in relevant and fluent Wikipedia articles.

Having a human in the loop resembles the set-up with a technical writer of a master document. But in this case, the master document is not English but a formal representation of an **abstract syntax**: a data structure that combines source content with a specification of syntactic structure. It is a kind of **second-order data** (where Wikidata itself is first-order), which still has the virtue of enabling deterministic NLG, at the same time as its correctness can be verified in relation to the first-order data.



Notice that the abstract syntax does not need to be a formal notation. It can look exactly like natural language, as long as it is a **controlled natural language** (**CNL**). A CNL is a formally defined subset of natural language, which has an unambiguous syntax and semantics. With the methods to be presented here, a CNL comes in as a natural input mode for interactive NLG.

The scale of Abstract Wikipedia is enormous, just thinking about the combination of 20 million articles and 300 languages. For GF, this means scaling up from 40 languages to 300, and from a few special areas of knowledge to everything under the sun. There is also a new technological challenge: how to enable Wikipedia authors to produce master documents in abstract syntax with an effort not much higher than writing them in English.

All in all, we can see the Abstract Wikipedia as a "man on the moon" project of NLG. It is vastly more demanding than anything done before — but at the same time, we know enough about the science and technology that is needed to be convinced that it can be done. It is just a matter of resources and effort.

1.3 Who is this document for?

This document is a hands-on guide on Natural Language Generation and GF. The focus is on multilingual NLG, where GF is perhaps the only systematic and scalable approach supported by mature software and language resources.

This document is meant to be self-contained: it can be followed by anyone with a basic knowledge of programming. In particular, no previous knowledge of GF is assumed. Such knowledge is of course useful for an in-depth

understanding of the scope and limitations of GF, and it can be gained from an introduction course (such as a summer school), the GF book (Ranta 2011), or an on-line tutorial. But here we take some shortcuts and show only those parts of GF that are needed for the tasks at hand.

The shortcuts are made this possible by the **GF Resource Grammar Library** (**RGL**), which takes care of the linguistic details needed in multilingual NLG. Many features of GF are only needed when implementing those details. Hence, when the RGL is used, the GF application programmer needs only to master a subset of the GF programming language.

Besides GF, we will use Python. To understand the Python code, the reader is assumed to have previous knowledge of Python. However, the algorithms written in Python will also be explained independently of the code, and it is possible to just run the code without going into details. Similar code could be written in Java or Haskell or C, since GF is interoperable with all these languages.

A special target group is those who want to contribute to the Abstract Wikipedia. Much of the data in our examples is collected from Wikidata, and the order of presentation follows the strategy outlined above where we proceed from atomic facts to second-order data.

Chapter 2

A gradual introduction to NLG in GF

Let us start the work with a data set that is simple and well-known, yet rich enough to illustrate the main issues of NLG. The set is a table with information about countries from Wikidata (in the category "sovereign states", `wd:Q3624078` in Wikidata). The original query can be found in

- https://w.wiki/3fV$

The results have been edited slightly, so that, in particular, every country appears only once in the result. Here is a fragment of the table:

country	**capital**	**area**	**population**	**continent**	**currency**
Afghanistan	Kabul	652230	36643815	Asia	Afghan afghani
Albania	Tirana	28748	3020209	Europe	Albanian lek
Algeria	Algiers	2381741	41318142	Africa	Algerian dinar
Andorra	Andorra la Vella	468	76177	Europe	euro
Angola	Luanda	1246700	29784193	Africa	kwanza
Argentina	Buenos Aires	2780400	44938712	South America	Argentine peso

We will proceed bottom-up from the simplest kind of NLG to more involved:

- Stage 1: atomic facts with templates to which data is inserted, e.g.

  the capital of Argentina is Buenos Aires
  the population of Argentina is 44938712
  the continent of Argentina is South America

- Stage 2: atomic facts with grammar, enabling other languages than English, with translations and proper inflections of names, e.g. Finnish

Suomi, Suomen ("Finland", "of Finland"), as well as idiomatic expressions for atomic facts, e.g. *Argentina has 44938712 inhabitants, Argentina is in South America.*

> *Argentina has 44938712 inhabitants*
> *Argentina is in South America*

- Stage 3: combinations of facts into fluent texts

  > *Argentina is a South American country with 44000000 inhabitants. Its area is 2780000 square kilometres. The capital of Argentina is Buenos Aires and its currency is Argentine peso.*

- Stage 4: selection, aggregation, and summarization of data, e.g.

  > *Brazil is the larges country in South America, with a population of over 210 million and an area of over 8 million square kilometres.*

At each stage, we provide both GF and Python code. The essential parts of the code are shown and explained in the text, while the entire code can be found in GitHub,

> https://github.com/aarneranta/NLG-examples

in the subdirectories

- `doc/facts1` for Stage 1,
- `doc/facts2` for Stage 2,
- `doc/facts3` for Stages 3 and 4

Each stage concludes with exercises, where readers are encouraged to apply the methods to their own languages and their own data.

2.1 Atomic facts

The first kind of things we want to express is **atomic facts** assigning **values** of **attributes** to **objects**. In our table, the objects of primary interest are the countries in the first column. Each of the later columns assigns a different attribute to this object:

- The capital of Argentina is Buenos Aires.

- The area of Argentina is 2780400 square kilometres.
- The population of Argentina is 44938712.
- The continent of Argentina is South America.
- The currency of Argentina is Argentine peso.

All facts have the same syntactic structure:

- The **Attribute** of **Object** is **Value**.

where **Attribute** is a noun, **Object** is a proper name, and **Value** is a number or a proper name.

Repeating the same syntactic structure in all sentences is not a way to build a natural-looking text. But it is already natural language and thereby serves some of the purposes of NLG. It constitutes a baseline from which the text can be improved with various NLG techniques. It is a **canonical representation** of the data contained in the cells of a table, where the sentences stand in one-to-one correspondance with the data itself.

2.2 Stage 1: Baseline NLG for atomic facts

We will now start writing code for an NLG system and use atomic facts as the first step. We will show parts of the full code, which can be found in the directory `facts1/`. The code consists of three kinds of files:

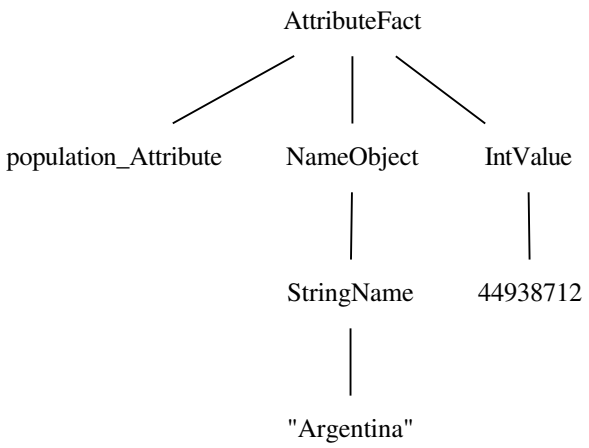
- GF files defining a grammar of facts
- Python files defining how data is converted to grammatical structures
- TSV (tab-separated values) files containing the data itself

The focus will be on the GF files, which would not require any changes if the Python files were replaced by equivalent C, Haskell, or Java files or the TSV files with other formats such as XML or Json. But we stick to Python and TSV so that we can quickly build a runnable system that the readers can test and develop further.

The minimum number of GF files is two: one for an **abstract syntax** and one for a **concrete syntax**. The abstract syntax defines a set of fact **trees** (**abstract syntax trees**), which are language-neutral representations of linguistic structures — for example,

```
AttributeFact
  population_Attribute
  (NameObject (StringName "Argentina"))
  (IntValue 44938712)
```

(in GF code), equivalently This tree can also be shown in a graphical representation,



in a graphical representation. The concrete syntax shows how trees are **linearized**, i.e. converted to **strings** in a natural language — for example,

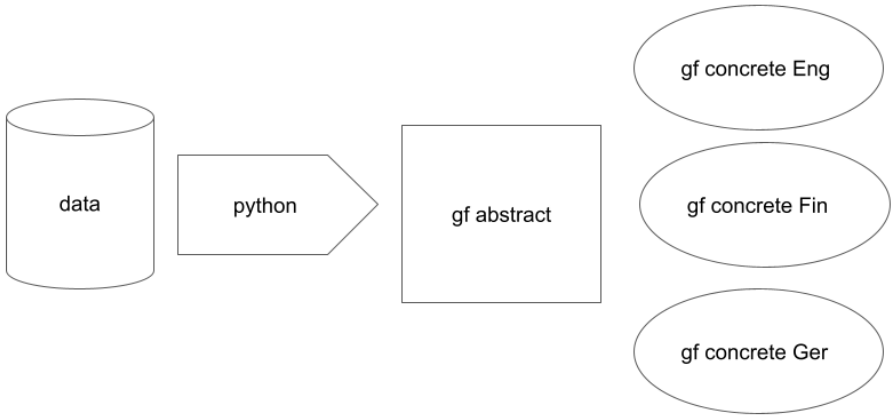
the population of Argentina is 44938712

The strength of GF lies in the possibility to have several concrete syntaxes, each corresponding to a different language:

Argentinas befolkning är 44938712
Argentiinan väkiluku on 44938712

The conversion of data to text is performed in Python (or other **host language**) code, which maps data to abstract syntax trees, rather than concrete strings. Thus a system can be extended with new languages by only writing new concrete syntax modules in GF, without touching the Python.

The following diagram shows the structure of an NLG systems we will build:



2.2.1 Abstract syntax

The GF file `Facts.gf` defines an abstract syntax for atomic facts. The complete code is here:

```
abstract Facts = {

cat
  Fact ;
  Object ;
  Attribute ;
  Value ;
  Name ;

fun
  AttributeFact : Attribute -> Object -> Value -> Fact ;

  capital_Attribute : Attribute ;
  area_Attribute : Attribute ;
  population_Attribute : Attribute ;
  continent_Attribute : Attribute ;
  currency_Attribute : Attribute ;

  NameObject : Name -> Object ;
```

```
  NameValue : Name -> Value ;
  IntValue : Int -> Value ;
  StringName : String -> Name ;
}
```

The parts of this file are

- a **module header** indicating that this is an abstract syntax module named `Facts`,
- a **module body**, in curly braces, consisting of two kinds of **judgements** ("rules"):
  - `cat`, **categories** of trees,
  - `fun`, **functions** for building trees from zero or more given trees.

The arrow notation in `fun` judgements is used for **function types** to separate the **argument types** from the **value type**. Thus the first `fun` judgement

```
AttributeFact : Attribute -> Object -> Value -> Fact
```

says that

- `AttributeFact` is a function that takes a `Attribute`, an `Object` and a `Value` as its arguments an produces a `Fact` as its value.

The limiting case are **constant functions**, which have no argument types — here the five ones from `capital_Attribute` to `currency_Attribute`.

The functions

```
IntValue : Int -> Value ;
StringName : String -> Name ;
```

have as their argument types the **built-in categories** `Int` and `String`. These categories are not declared in the `cat` judgments and have no user-defined functions to build trees in them: their objects are given as **integer literals** (such as `44938712`) and **string literals** (`"Argentina"`), respectively.

For other categories than the built-in ones, trees are built by combining smaller trees with functions. Thus the representation of the sentence

- *the population of Argentina is 44938712*

is a tree of type `Fact` built by the function `AttributeFact` as follows (repeating what was shown above):

```
AttributeFact
  population_Attribute
  (NameObject (StringName "Argentina"))
  (IntValue 44938712)
```

Notice the GF syntax for function applications,

```
f a b c
```

similar to Haskell, rather than

```
f(a,b,c)
```

as in Python and Java.

2.2.2 Concrete syntax

A concrete syntax specifies how abstract syntax trees are linearized to strings in some language. Here is the complete code from the file `FactsEng.gf`,

```
concrete FactsEng of Facts = {

lincat
  Fact = Str ;
  Object = Str ;
  Attribute = Str ;
  Value = Str ;
  Name = Str ;

lin
  AttributeFact attr obj val = "the" ++ attr ++ "of" ++ obj ++ "is" ++ val ;

  capital_Attribute = "capital" ;
  area_Attribute = "area" ;
  population_Attribute = "population" ;
  continent_Attribute = "continent" ;
  currency_Attribute = "currency" ;

  NameObject name = name ;
  NameValue name = name ;
  IntValue int = int.s ;
  StringName str = str.s ;
}
```

This concrete syntax file consists of

- a module header saying that it is a concrete syntax named `FactsEng` of the abstract syntax `Facts`,
- a module body containing two kinds of judgement:
  - `lincat` specifying the **linearization type** of each category,
  - `lin` specifying the **linearization function** of each function.

The simplest linearization type is `Str`, **string**. This type is sufficient for expressing **context-free grammars** in GF. It is too limited for full-scale NLG, but provides an easy way to introduce GF. We will extend it to richer types later.

The linearization functions are defined with expressions that combine **variables** with **string literals**. Thus the first rule

```
  AttributeFact attr obj val = "the" ++ attr ++ "of" ++ obj ++ "is" ++ val
```

uses the variables `prop`, `obj`, and `val`, which stand for the arguments of the function, and the literals `"the"`, `"of"`, and `"is"`. The method of combination is **concatenation**, denoted by `++` (again borrowed from Haskell).

The built-in types `Int` and `String` have more complex linearization types than `Str`. Hence their objects cannot be used directly when a `Str` is expected, but extracted as a **projection** `.s` — a notation that will be fully explained later.

A slight variant of `FactsGer` defines a concrete syntax for German:

```
concrete FactsGer of Facts = {

-- lincat as in FactsEng

lin
  AttributeFact attr obj val = attr ++ "von" ++ obj ++ "ist" ++ val ;

  capital_Attribute = "die Hauptstadt" ;
  area_Attribute = "die Fläche" ;
  population_Attribute = "die Einwohnerzahl" ;
  continent_Attribute = "der Kontinent" ;
  currency_Attribute = "die Währung" ;

-- the remaining lin as in FactsEng
}
```

More languages can be added as an easy exercise (see below). The GitHub directory contains (at the time of writing), also Finnish.

A potentially useful hack, common in templates-based NLG, is shown in `FactsFin`:

```
  AttributeFact attr obj val = "maan" ++ obj ++ attr ++ "on" ++ val ;
```

When assigning an attribute to country name, the name must be inflected in the genitive. But there is no simple way to do this, so the wrapper word *maa* ("country") is added amd turned into the genitive *maan*. The result is an equivalent of English *of the country X*. It is grammatically correct but extremely clumsy Finnish, immediately revealing that the text is machine-generated. This trick is used frequently for instance in Facebook, to report someneone's location: *Urho on paikassa Helsinki* instead of *Urho on Helsingissä*.

2.2.3 Testing grammars in the GF shell

The GF shell, just like the Python shell, is a line-based tool that enables testing GF code while developing it. The following session shows some useful commands in the shell:

```
# start gf in the OS shell, see the welcome message
$ gf

         *  *  *
      *           *
    *               *
   *
   *
   *        * * * * * *
   *        *         *
    *       * * * *  *
      *     *      *
         *  *  *

This is GF version 3.10.4.
```

```
# importing (i) a GF file makes a language available
> i FactsEng.gf
linking ... OK

Languages: FactsEng
1 msec

# generate random (gr) abstract syntax tree
Facts> gr
AttributeFact area_Attribute (NameObject (StringName "Foo")) (IntValue 999)

# generate random tree and pipe (|) it to linearize (l)
Facts> gr | l
the population of Foo is Foo

# parse (p) a string into a tree
Facts> p "the capital of France is Paris"
AttributeFact capital_Attribute (NameObject (StringName "France"))
  (NameValue (StringName "Paris"))

# import another concrete syntax for the same abstract
Facts> i FactsGer.gf
linking ... OK

Languages: FactsEng FactsGer

# translate by parsing in one language and linearizing into another one
Facts> p -lang=Eng "the capital of France is Paris" | l -lang=Ger
die Hauptstadt von France ist Paris

# quit (q) GF and return to the OS shell
Facts> q
See you.
```

2.2.4 Python code converting data to text

GF grammars can be used in Python via the library called `pgf`. **Portable Grammar Format** (**PGF**) is the binary "machine language" of GF.

The first step in using GF from Python is to compile the GF source code into PGF:

```
gf -make FactsEng.gf FactsFin.gf FactsGer.gf
```

The result is a single file, `Facts.pgf`, which can be read by Python via the classes and functions in the `pgf` library.

Here is an example of a python3 session illustrating the functionalities we need to star with:

```
>>> import pgf

# read a PGF object from file
>>> gr = pgf.readPGF('Facts.pgf')

# show the names of the available concrete syntaxes
>>> print(list(gr.languages.keys()))
['FactsEng', 'FactsFin', 'FactsGer']

# extract the concrete syntax object for English
>>> eng = gr.languages['FactsEng']

# build a tree, "Expression" in technical jargon
>>> attr = pgf.Expr('area_Attribute',[])

# linearize the tree in English
>>> eng.linearize(attr)
area

# build some more trees incrementally
>>> obj = pgf.readExpr('NameObject (StringName "France")')
>>> val = pgf.readExpr('123')
>>> fact = pgf.Expr('AttributeFact',[attr,obj,val])

# show the resulting tree in GF notation
>>> print(fact)
AttributeFact area_Attribute (NameObject (StringName "France")) 123
```

```
# linearize the resulting tree
>>> print(eng.linearize(fact))
the area of France is 123
```

The following classes and methods are used in this first version of Python NLG:

- `PGF`, class of objects obtained from `.pgf` files (no constructor used here):
  - `readPGF(<filename>)`, function that returns a `PGF` object;
- `Expr`, class of abstract syntax trees ("expressions" in the technical jargon):
  - `Expr(fun,[Expr])`, its constructor that reads a function name (a string) and a list of subtrees,
  - `readExpr(str)`, function that reads a tree from a string (in the GF notation);
- `Concr`, class of concrete syntaxes obtained from PGF languages:
  - `PGF.languages`, class < variable with a dictionary mapping module names to `Concr` objects,
  - Concr.linearize(expr), linearization function for trees.

Some other functions, such as a parsing method, will be introduced later as needed. Complete documentation can be found in

http://www.grammaticalframework.org/doc/runtime-api.html#python

The Python program `facts1/facts.py` reads a TSV file and a GF grammar, and prints out a linearization of every atomic fact in all of the languages covered by the grammar. The complete code is here:

```
import pgf

# read country data from a tsv file into a named tuple
from collections import namedtuple
def get_countries(filename):
    countries = []
    Country = namedtuple('Country',
                'country capital area population continent currency')
    file = open(filename)
    for line in file:
        fields = Country(*line.split('\t'))
        countries.append(fields)
```

```
    return countries

#for each tuple, build a list of GF trees, one for each atomic fact
  object = pgf.Expr('NameObject', [mkName(c.country)])
  return [
    pgf.Expr('AttributeFact',[pgf.Expr(attr,[]),object,val])
      for (attr,val) in [
        ('capital_Attribute',    pgf.Expr('NameValue',[mkName(c.capital)])),
        ('area_Attribute',       pgf.Expr('IntValue', [mkInt(c.area)])),
        ('population_Attribute', pgf.Expr('IntValue', [mkInt(c.population)])),
        ('continent_Attribute',  pgf.Expr('NameValue',[mkName(c.continent)])),
        ('currency_Attribute',   pgf.Expr('NameValue',[mkName(c.currency)]))
        ]
    ]

# building an integer literal from a string
def mkInt(s):
    return pgf.readExpr(str(s))

# building a Name tree from a string
def mkName(s):
    return mkApp('StringName',[pgf.readExpr(str('"' + s + '"'))])

# putting it all together
def main():
    gr = pgf.readPGF('Facts.pgf')   # read the compiled grammar
    countries = get_countries('../data/countries.tsv')
    langs = list(gr.languages.values())
    for lang in langs:
        text = []
        for c in countries:
            for t in country_facts(c):
                text.append(lang.linearize(t))
        print('\n'.join(text))

main()
```

Running this Python code results in 1940 sentences, stating 5 atomic facts

about 194 countries in two languages:

```
$ python3 facts.py | more
the capital of Afghanistan is Kabul
the area of Afghanistan is 652230
the population of Afghanistan is 36643815
the continent of Afghanistan is Asia
the currency of Afghanistan is Afghan afghani
...
maan Peru pääkaupunki on Lima
maan Peru pinta-ala on 1285216
maan Peru asukasluku on 29381884
maan Peru maanosa on South America
maan Peru valuutta on Peruvian sol
...
die Hauptstadt von Zimbabwe ist Harare
die Fläche von Zimbabwe ist 390757
die Einwohnerzahl von Zimbabwe ist 16529904
der Kontinent von Zimbabwe ist Africa
die Währung von Zimbabwe ist United States dollar
```

2.2.5 Exercises

Add a language

An easy exercise to start with is to add a module for a new own language:

- copy and modify `FactsEng.gf`
- compile the pgf file with this new module included
- run the `facts.py` script

Don't worry if it looks hopeless to get the grammar right: we will fix this by using more power of GF.

Change the domain of data

The `Facts` grammar is completely general for any kind of data that can be presented in the format "the Attribute of Object is Value", except for the names of the attributes and wrapper words such as `maan` in `FactsFin.gf`. You can hence apply it to some other tabular data, with the following modifications:

- in `facts.py`,
  - change `get_countries()` to reflect the number and meaning of the fields in your data,
  - change `country_facts()` to select the facts you want to display and the proper grammar functions for them;
- in `Facts*.gf`,
  - change the set of Attribute functions and their linearizations,
  - if necessary, change the wrapper words used in different languages.

Get Wikidata

The data format "the Attribute of Object is Value" is a special case of **semantic triples**, such as those used in RDF (**Resource Description Framework**). RDF if a standard format for storing data on the web and is used, for example, in Wikidata, which is a data resource for the Wikipedia. Thus in Wikidata, the fact that Cairo is the capital of Egypt is represented as the triple

```
wd:Q85 wdt:P1376 wd:Q79
```

where `wd:Q85` is an identifier for Cairo, `wd:Q79` for Egypt, and `wdt:Q85` for the relation "is the capital of".

Wikidata is accessible via its query interface

https://query.wikidata.org/

In this interface, one can write queries in the SPARQL language (**SPARQL Protocol and RDF Query Language**) and get answers in a tabular form. For example, the following SPARQL query generates a table of countries and their capitals:

```
select ?city ?country ?cityLabel ?countryLabel {
  ?country wdt:P31 wd:Q6256 .
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P1376 ?country .
  ?city rdfs:label ?cityLabel .
  filter(lang(?cityLabel) = 'en') .
  ?country rdfs:label ?countryLabel .
  filter(lang(?countryLabel) = 'en')
}
```

The result is displayed as a table in the web browser, but it can also be downloaded in the TSV format. This gives a direct way to apply the methods seen by now to Wikidata. What is more, the query also suggests a way to obtain translations of names, by varying the language code (e.g. from `en` to `fi`).

We will return to Wikidata many times later, but you can get a first feeling of it by writing some query, downloading the result as TSV, and using the result in the way explained in the previous exercise.

2.3 Stage 2: Improving NLG for atomic facts

We will now improve the quality of NLG by making full use of GF. The code for this can be found in the directory `facts2/`. The main improvents are

- going from static templates to grammatical structures,
- thereby enabling more fluent language,
- translating the names to different languages,
- restructuring the code so that it is easier to extend and reuse.

These improvements are based on two new concepts of GF:

- the **Resource Grammar Library** (RGL), giving an easy access to grammar rules,
- the **module system** of GF, enabling a modular structure of the code.

2.3.1 Refactoring abstract syntax

The GF code in `facts1/` is monolithic: there is just one GF file for the abstract syntax and one for each language. These files contain both generic concepts about facts and concepts specific to countries. If we want to apply the grammar on some other domain, such as Nobel prize winners, we have to make a copy of the code and change some parts of it. A better way to do this is to divide the code into a generic and domain-specific part:

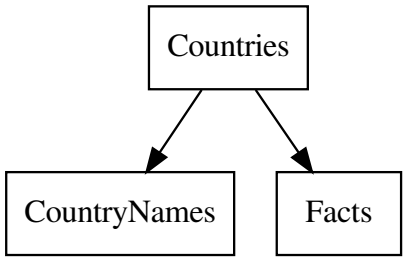
- `Facts.gf`, defining general categories and syntactic structures,
- `Countries.gf`, defining domain-specific concepts, such as the attributes,

In addition, we will now introduce a third kind of a module,

- `CountryNames.gf`, defining the names of countries, cities, and currencies.

These three modules are organized hierarchically, so that `Countries` is the **top module**, which **inherits** from the other two modules. The hierarchy

can be visualized in GF by using the command `dg`; see "help dg" in the GF shell for details about its use. The resulting graph looks as follows:



The generic `Facts` module is now a proper part of the earlier `Facts` module:

```
abstract Facts = {

cat
  Fact ;
  Object ;
  Attribute ;
  Value ;
  Name ;

fun
  AttributeFact : Attribute -> Object -> Value -> Fact ;
  NameObject : Name -> Object ;
  NameValue : Name -> Value ;
  IntValue : Int -> Value ;
}
```

Notice that, in additions to the country-specific attributes, we have left out the `StringName` function that builds names from string literals. The reason is that we now want to introduce names as abstract syntax identifiers, so that we can linearize them in language-specific ways. This brings us to the module `CountryNames`:

```
abstract CountryNames = {

cat CName ;
```

```
-- generated from Wikidata
fun Africa_CName : CName ;
fun Asia_CName : CName ;
fun Central_America_CName : CName ;
fun Europe_CName : CName ;
fun North_America_CName : CName ;
fun South_America_CName : CName ;
--- followed by hundreds more names
}
```

This module contains

- domain-specific name categories, here just `CName`, but we could also have separate categories for continents, countries, cities, and currencies,
- names extracted from data, here for all continents, countries, cities, and currencies

The rationale of domain-specific categories will become obvious later. One reason is that different categories can have different properties. For example, a country name can include an expression for a citizen of that country, which is not possible for some other types of names.

Extracting names from data can be done in different ways. For example, the Wikimedia identifiers (such as `wd:Q79` for Egypt) would work perfectly as abstract function names: they are unique, and they have well-defined mappings to names ("labels") in different languages. Here we have, however, used the English names to make it easier to inspect the grammar. The important thing in both cases is to make sure the function names are valid GF identifiers. This means, among other things, surrounding them by single quotes if they contain special characters. The function

```
extract_names.mkFun(name,category)
```

in `facts2/extract_names.py` is a way to guarantee this (in almost all cases). The same module also has the function `name_rules()`, which forms a pair of GF `fun` and `lin` rules for a name in a wanted language, given a TSV file with aligned names.

The topmost module `Countries` inherits both `Facts` and `CountryNames`, to which it adds some code of its own:

```
abstract Countries = Facts, CountryNames ** {
fun
-- using CNames
  cName : CName -> Name ;

-- basic properties
  capital_Attribute : Attribute ;
  area_Attribute : Attribute ;
  population_Attribute : Attribute ;
  continent_Attribute : Attribute ;
  currency_Attribute : Attribute ;

-- specialized expressions for properties
  populationFact : CName -> Int -> Fact ;
  continentFact : CName -> CName -> Fact ;
}
```

The main parts are

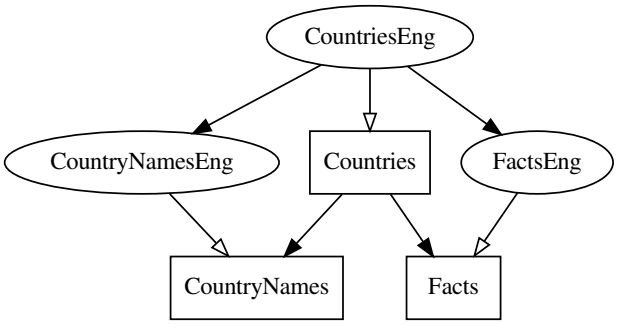
- a header that lists the inherited modules and adds (with operator `**`) a body of new judgements,
- functions for using domain-specific names in the domain-independent category `Name`,
- domain-specific attributes,
- other domain-specific functions, which enable idiomatic expressions of facts.

The last part is a main ingredient in making NLG more natural. Here wi illustrate it with just two alternative ways for expressing facts:

- *Argentina has 41343201 inhabitants* (`populationFact`)
- *Argentina is in South America* (`continentFact`)

2.3.2 Refactoring concrete syntax

The concrete syntax is usually, although not necessarily, divided into modules parallel to the abstract syntax. The module structure then looks as follows:



The graph generated by `dg` uses

- rectangles for abstract modules,
- ellipses for concrete modules,
- solid arrow heads for inheritance,
- hollow arrow heads for the concrete-abstract relation.

Just to refactor `FactsEng` in this way would not require more explanation. But we will now introduce a new concept: **resource modules** with reusable operations. Here is how they are used in `FactsEng`:

```
concrete FactsEng of Facts = open SyntaxEng, SymbolicEng in {

lincat
  Fact = Cl ;
  Object = NP ;
  Attribute = CN ;
  Value = NP ;
  Name = NP ;

lin
  AttributeFact attr obj val =
    mkCl (mkNP the_Det (mkCN attr (mkAdv possess_Prep obj))) val ;
  NameObject name = name ;
  NameValue name = name ;
  IntValue int = symb int ;
}
```

The only new notation in this module is the `open` directive on the first line. It makes the contents of two modules, `SyntaxEng` and `SymbolicEng`, usable in the module body. But the contents are not inherited — in fact, this would

not even make sense, because the opened modules are not concrete syntaxes but of a third kind, **resource modules**.

We do not need to write any resource modules ourselves at this point, but just use those available in the RGL (GF Resource Grammar Library). The three most important modules are, for each language with its language code as suffix,

- `Syntax`, which contains syntactic categories and functions,
- `Paradigms`, which contains operations for inflecting words,
- `Symbolic`, which enables the use of numeric and other literals in syntactic structures.

The RGL is installed in a standard, although platform-specific, place in the file system, where the GF compiler finds it via the environment variable `GF_LIB_PATH`. If you can import `FactsEng` without error messages in GF, the installation works as it should, and you do not need to worry about it.

The next thing is to study the contents of the library via its API,

- https://www.grammaticalframework.org/lib/doc/synopsis/

Instead of giving a systematic RGL tutorial (which can be found elsewhere), let us just look at the parts relevant to the current task. The first objective is to define linearization types in terms of RGL categories. The ones used in `FactsEng` are

- `Cl`, **clause**, sentences expressing simple predications,
- `NP`, **noun phrase**, phrases usable as subjects and objects in clauses,
- `CN`, **common noun**, phrases from which noun phrases can be built adding determiners.

The difference between `NP` and `CN` may be tricky for the non-linguist. To give an example, *country* is a `CN`, from which one can form `NP`s such as *the country*, *these countries*. Thus a `CN` has both singular and plural forms, whereas an `NP` already has a fixed number, either singular or plural.

The authors of the RGL have to define in detail

- how the forms of `CN`s are built: *country-countries* vs. *continent-continents*;
- what forms are selected in `NP`s: *this country* vs. *these countries*;
- how `NP`s are used in `Cl`s: *this country* ***is*** *in Europe* vs. *these countries* ***are*** *in Europe*;

Since this linguistic ground work has been done once and for all in the RGL, NLG engineers do not need to repeat it. But we do need to develop an idea about what combinations of RGL functions are possible. The linearization of `AttributeFact` is quite a mouthful to start with:

```
mkCl (mkNP the_Det (mkCN attr (mkAdv possess_Prep obj))) val
```

Let us work out an example to see how it fits this pattern:

the capital of Italy is Rome

The outermost part of the GF expression is `mkCl` applied to two `NP`s, presented as follows in the API:

```
mkCl : NP -> NP -> Cl  -- she is the woman
```

The example following the type of the function contains two `NP`s, *she* and *the woman*, which get connected with the **copula** *is* (or *are* or *am*, depending on the first `NP`). Applying this to our example, we build a GF expression of the form

```
mkCl <the capital of Italy> <Rome>
```

Both of the two arguments are `NP`s, as guaranteed by the `lincat`s of `Object` and `Value`, respectively. The `Object` part needs to be analysed further. It uses the following API functions, which we now annotate with the current example rather than the API documentation:

```
mkNP    : Det -> CN -> NP    -- the capital of Italy
the_Det : Det                -- the
mkCN    : CN -> Adv -> CN    -- capital of Italy
mkAdv   : Prep -> NP -> Adv  -- of Italy
possess_Prep : Prep          -- of
```

We recommend you to convince yourself that the functions are combined in a proper way, so that the value types of each subexpression match the expected argument types!

Once we manage to build a **type-correct** expression in this way from RGL function, we can be sure that the result is **grammatically correct**. Of course, it may not be the expression you want: to guarantee this, you need to test your grammar (e.g. with random generation and linearization) and gradually gain experience that reduces the need of testing.

The remaining judgements in **FactsEng** are simple. `IntValue` uses the RGL operation

```
  symb : Int -> NP  -- 23
```

Notice that we no more need to care about how a string is projected from `Int`, as we had to do in `facts1/`. As a rule of thumb,

- The only form of GF expression that is needed in concrete syntax when the RGL is used is function application.

2.3.3 Defining the concrete syntax lexicon

The concrete syntax of `CountryNames` opens `SyntaxEng` and now also `ParadigmsEng`:

```
concrete CountryNamesEng of CountryNames =
  open SyntaxEng, ParadigmsEng in {

lincat CName = NP ;

oper mkCName : Str -> NP = \s -> mkNP (mkPN s) ;

-- generated
lin Africa_CName = mkCName "Africa" ;
lin Asia_CName = mkCName "Asia" ;
lin Central_America_CName = mkCName "Central America" ;
```

A new kind of judgement, with the keyword `oper`, is used here:

```
  oper mkCName : Str -> NP = \s -> mkNP (mkPN s)
```

An `oper` judgement defines neither a function nor its linearization but an **auxiliary operation**, which is used in `lin` definitions. The operation `mkCName` is simple, using

- `mkNP : PN -> NP` from `SyntaxEng`, using a `PN` (**proper name**) as an `NP`
- `mkPN : Str -> PN` from `ParadigmsEng`, building a `PN` from a string

One could of course use the combination of those directly in each `lin` rule. But it is generally a good practice to define a specific `mk`C operation for every category C. Then it can be easily varied if for instance the linearization type of C changes.

The category `CName`and the operation `mkCName` can also be enriched with more information than just a plain proper name string. Examples include

- *United States*, which in some contexts appears with the definite article and also has alternative names such as *USA*
- *British Virgin Islands*, which is treated as a plural noun phrase

We will return to this question when discussing the German and Finnish implementations, which are impossible to get working with just plain strings.

2.3.4 The top module

The top module `CountriesEng` offers almost nothing new in terms of GF:

```
concrete CountriesEng of Countries = FactsEng, CountryNamesEng **
  open SyntaxEng, ParadigmsEng, SymbolicEng in {

lin
  cName name = name ;

  capital_Attribute = mkAttribute "capital" ;
  area_Attribute = mkAttribute "area" ;
  population_Attribute = mkAttribute "population" ;
  continent_Attribute = mkAttribute "continent" ;
  currency_Attribute = mkAttribute "currency" ;

  populationFact obj int =
    mkCl obj have_V2 (mkNP <symb int : Card> (mkN "inhabitant")) ;
  continentFact obj name = mkCl obj (SyntaxEng.mkAdv in_Prep name) ;

oper
  mkAttribute : Str -> CN = \s -> mkCN (mkN s) ;
}
```

The only new constructs are

- the **type annotation** `<symb int : Card>`,
- the **qualified name** `SyntaxEng.mkAdv`

The type annotation is needed for **overload resolution**: the RGL operation `symb` is **overloaded**, i.e. the same function name is used for several different functions, which only differ as for their type:

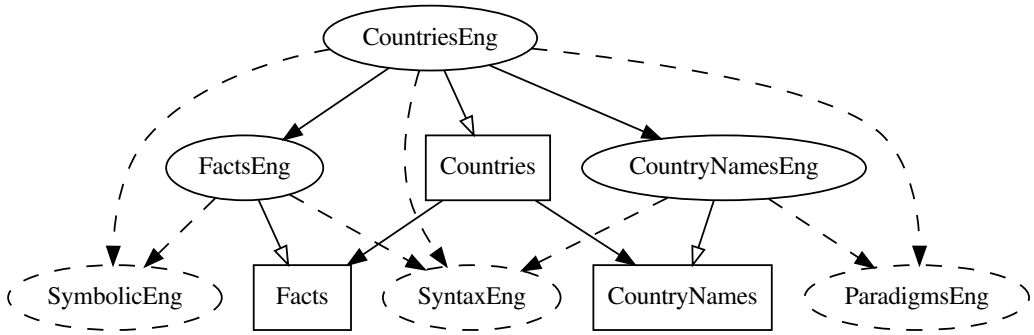
```
  symb : Int -> NP    -- 23 (is prime)
```

```
symb : Int -> Card  -- 23 (houses)
symb : Str -> NP    -- x
```

Overloading is used throughout the RGL to keep the number of function names manageable. It is usually resolved correctly by the type checker of GF — but in this very case, this did not happen, so we added a type annotation to guide the type checker.

The qualified name is needed because both `SyntaxEng` and `ParadigmsEng` define an operation called `mkAdv`. Conceptually, one would expect the choice to be made by overload resolution, because these operations have different types. For complicated technical reasons, this has not (yet) been implemented in GF.

Now that we have walked through the refactored modules using the RGL, we can draw the full picture of the module structure that shows both abstract, concrete, and resource modules:



Resource modules are shown as dashed ellipses and opening them as dashed arrows. This diagram hides dozens of modules that underlie the explicitly opened RGL modules. The user of the RGL seldom needs to be aware of them.

2.3.5 Python code: the generic method

As in `facts1`, we use a Python file to

- read data from a TSV file,
- map each fact into an abstract syntax tree,
- call GF's linearization to render the trees into each language in scope.

But this time we have modularized the Python code in a way analogous to the grammar:

- `data_facts.py` works for any grammar built on top of `Facts.gf`,
- `country_facts.py` adds things specific to `Countries.gf`.

The specific part is optional, in the sense that `data_facts.py` alone is sufficient to generate basic natural language texts based on the `Countries` grammar. These texts follow a simple pattern: for each data attribute, they use the tree

```
AttributeFact <attr_Attribute> obj val
```

where <attr_Attribute> is the GF function corresponding to the data attribute `attr`. The simple procedure is already better than `facts1` as it uses a grammar. The advantages are not seen in the English output, but make a clear difference in German and Finnish.

In the specialized procedure of `country_facts.py`, we moreover use special functions such as those for population and continent.

The module `facts2/data_facts.py` also scales up to the next stages and needs therefore not to be changed in `facts3`.

The central part of the code is the class `FactSystem`, which contains a grammar and methods for converting data into text by using the grammar.

```
class FactSystem:
    def __init__(self,fnames,gr,lang1):
        self.fieldnames = fnames # names of datafields as a string
        self.grammar = gr        # a PGF object
        self.language1 = lang1   # the language used for parsing input

    # read tuple data from tsv file
    def get_data(self,filename):
        ...
        return data

    # run a fact_generator function that converts data to trees
    def run(self,datafile,fact_generator):
            ...
            for tree in fact_generator(self,data):
                text.append(lang.linearize(tree))
            ...
```

A possibly surprising method in the class is one that uses the **parser**:

```
    # parse strings to trees in category cat
    def str2exp(self,cat,s):
        eng = self.grammar.languages[self.language1]
        try:
            pp = eng.parse(s,cat=pgf.readType(cat))
            _,e = pp.__next__()
            return e
        except:
            print("WARNING:","no", cat, "from", s)
            return pgf.Expr(s,[])
```

Even though we are mainly interested in generation and not parsing, this is a useful way to map, for instance, data attributes to GF attribute functions. Then we need not assume hard-coded conventions such as from data attribute `area` to GF function `area_Attribute`, as we did in `facts1`.

The `parse` method of a concrete syntax works as in the following Python session:

```
# extract a concrete syntax
>>> eng = gr.languages['CountriesEng']

# parse a string in a wanted category
>>> pp = eng.parse('the capital of Italy is Rome',
           cat=pgf.readType('Fact'))

# the result is a interator on probabilities and trees
# extract the first (most probable) tree
>>> _,tree = pp.__next__()

# show tre tree
>>> print(tree)
AttributeFact capital_Attribute (NameObject (cName Italy_CName))
  (NameValue (cName Rome_CName))
```

A baseline `fact_generator` is `simple_facts()`, which expresses each datapoint with `AttributeFact`, in the same way as at Stege 1 but now using the parser to map data attribute names to GF function names. The function if moreover generic for any dataset, as it takes the field names and the data

key from a class variable of a `FactSystem` (assuming that the key attribute is in field 0):

```
def simple_facts(factsys,data):
    "for every field in every tuple, generate an attribute fact sentence"
    fields = factsys.fieldnames.split()
    facts = []
    for tuple in data:
        ltuple = list(tuple)
        for (attr,val) in [(fields[i],ltuple[i]) for i in range(1,len(fields))
            fact = pgf.Expr("AttributeFact", [
                    factsys.str2exp("Attribute",attr),
                    factsys.str2exp("Object",ltuple[0]),
                    factsys.str2exp("Value",val)])
            facts.append(fact)
    return facts
```

To run NLG on a datafile, we then

- read a PGF grammar from a file,
- create a `FactSystem` with relevant arguments,
- run the `FactSystem` with a chosen `fact_generator`:

```
def example_run():
    gr = pgf.readPGF('Countries.pgf')
    factsys = FactSystem(
        'country capital area population continent currency',
        gr,
        'CountriesEng'
        )
    factsys.run('../data/countries.tsv',simple_facts)
```

2.3.6 Python code: domain-specific methods

Similarly to `Facts.gf`, `data_facts.py` applies to any data that is arranged in tables where the columns are attributes and the rows are their combinations corresponding to data objects. The generated language consists of straightforward sentences of the form *the Attribute of Object is Value*. The specific modules `CountryNames.gf` are `Countries.gf` are only used to obtain the natural language names of attributes and values.

The file `country_facts.py` makes use of the domain-specific GF constructs, illustrated by `populationFact` and `continentFact`. The fact generator function `country_facts_embedded` uses these functions for the population and the continent, whereas all the others are still treated in the generic way:

```
def country_facts_embedded(factsys,tuple):
    countr = factsys.str2exp("CName",tuple[0])
    cap    = factsys.str2exp('Name',tuple.capital)
    cont   = factsys.str2exp('CName',tuple.continent)
    curr   = factsys.str2exp('Name',tuple.currency)
    pop    = mkInt(tuple.population)
    are    = mkInt(tuple.area)

    factsys.grammar.embed("G")
    import G
    object = G.NameObject(G.cName(countr))

    return [
      G.AttributeFact(G.capital_Attribute, object, G.NameValue(cap)),
      G.AttributeFact(G.area_Attribute, object, G.IntValue(are)),
      G.populationFact(countr, pop),
      G.continentFact(countr, cont),
      G.AttributeFact(G.currency_Attribute, object, G.NameValue(curr))
      ]
```

This function illustrates another new functionality from Python's PGF library:

- `grammar.embedded("G")` creates a **Python module** called `G`.
  - This module can be **imported** just like any Python module.
  - It contains functions `G.`f for all functions f in the abstract syntax.

The use of the `G` functions is much more concise than the constructor `pgf.Expr`. Just compare the first returned tree with

```
pgf.Expr('AttributeFact',
  [pgf.Expr('capital_Attribute',[]), object, pgf.Expr('NameValue',[cap])])
```

Another advantage — even more important in larger projects — is that existence of `G` functions is checked by Python's dynamic type checker, so that

proper feedback is obtained for errors like misspellings. (Their arity, however, is only checked by an assertion in the C code underlying the Python's pgf module.) The generated module `G` is called an **embedded grammar**, since it is a grammar embedded in Python.

A possibly even easier way to construct abstract syntax trees in Python is, surprise surprise, by using the parser. The same behaviour as `country_facts_embedded` is obtained by `country_facts_parsed`:

```
def country_facts_parsed(factsys,tuple):
    countr = factsys.data2lin("CName",tuple[0])
    cap    = factsys.data2lin('Name',tuple.capital)
    cont   = factsys.data2lin('CName',tuple.continent)
    curr   = factsys.data2lin('Name',tuple.currency)
    pop    = mkInt(tuple.population)
    are    = mkInt(tuple.area)

    return [ factsys.str2exp('Fact',s) for s in
        [
        "the capital of {} is {}".format(countr,cap),
        "the area of {} is {}".format(countr,are),
        "{} has {} inhabitants".format(countr,pop),
        "{} is in {}".format(countr, cont),
        "the currency of {} is {}".format(countr,are)
        ]
      ]
```

The fact trees are now obtained by parsing strings with the concrete syntax of the input language. The strings are built separately for each tuple by inserting values in braces and using Python's `format()` method.

Notice the way in which the values are given:

```
cap = factsys.data2lin('Name',tuple.capital)
```

We could actually write simply

```
cap = tuple.capital
```

if we knew that the linearization of the capital is the same as its name in the data. But since this would be an unnecessarily restrictive assumption, we perform the roundtrip via parsing and linearization.

The roundtrip method is defined as follows in the `FactSystem` class:

```
    def exp2str(self,exp):
        eng = self.grammar.languages[self.language1]
        return eng.linearize(exp)

    def data2lin(self,cat,s):
        return self.exp2str(self.str2exp(cat,s))
```

As parsing is usually the most expensive PGF function, the parser method of defining trees may result in slower code than the embedding method. In the current case the difference is negligible: 0.367 seconds instead of 0.238 seconds for the whole table.

2.3.7 Other concrete syntaxes: German and Finnish

- TODO write about German first, then less about Finnish

The German concrete syntax has naturally the same structure as English. The `FactsFin` module looks as follows:

```
concrete FactsFin of Facts =
  open SyntaxFin, SymbolicFin, (E=ExtendFin) in {

lincat
  Fact = Cl ;
  Object = NP ;
  Attribute = CN ;
  Value = NP ;
  Name = NP ;

lin
  AttributeFact attr obj val = mkCl (mkNP (E.GenNP obj) prop) val ;
  NameObject name = name ;
  NameValue name = name ;
  IntValue int = symb int ;
}
```

Two new GF features are used:

- in the `open` list, the module `ExtendFin` is imported in a **qualified mode**, where all its uses need to be marked with `E.`;

- in `AttributeFact`, a qualified function `E.GenNP` is used; this is similar to `SyntaxEng.mkAdv` above, except that `E` is not the name of a module, but an alias name defined in the `open` list.

Using `ExtendFin` in a qualified mode would not be necessary, but it is a good practice to qualify modules outside the standard RGL API (which includes `Syntax`, `Paradigms`, and `Symbolic`). This makes it easier to see what parts of the code are likely to be different for different languages.

The function `GenNP` forms the genitive of a noun phrase to be used as a determiner. It does not belong to the standard API, because it is not available for all languages. For English, `ExtendEng.GenNP` does exist, but it is not used in this grammar. It would produce *France's capital* instead of *the capital of France*.

Apart from the genitive construction, the code in `FactsFin` is similar to `FactsEng`. When the RGL API is used, one can actually bootstrap new languages by just copying the code of an old language and changing the quoted strings. This would result in somewhat clumsy Finnish but still work mostly better than the simple templates of `facts1`.

The `CountryNamesFin` module is obviously the one that differs the most from English. The first step to create it when Wikidata is available is to fetch the country and other names with a query:

```
select ?country ?countryLabelEn ?countryLabelFi {
  ?country wdt:P31/wdt:P279* wd:Q3624078 .
  ?country rdfs:label ?countryLabelEn .
  ?country rdfs:label ?countryLabelFi .
  filter(lang(?countryLabelEn)='en')
  filter(lang(?countryLabelFi)='fi')
}
```

With some simple programming, the Finnish names can be inserted as linearizations of the constants:

```
lin Bucharest_CName = mkCName "Bukarest" ;
lin Finland_CName = mkCName "Suomi" ;
lin Marshall_Islands_CName = mkCName "Marshallinsaaret" ;
lin Russia_CName = mkCName "Venäjä" ;
lin United_States_of_America_CName = mkCName "Yhdysvallat" ;
```

However, this is not enough to obtain flawless Finnish. There are three kinds of problems — and these are in fact problems for many other languages as well, for instance French and German. So let us try to explain them on the general level:

- The morphological features (inflection, gender) are not always predictable from the string alone, but require more information. Example: *United States*, which is inflected in plural but agrees as singular (*The United States has...*).
- Some "names" are actually complex noun phrases with many parts that undergo agreement in syntactic combinations. Example: *new shekel*, where both the adjective and the noun need to be changed in the plural.
- When using a name as location, some countries may use a preposition or case corresponding to *on*, some to *in*.

The outcome of this is that we need to

- enrich the linearization type of `CName`,
- introduce constructors that can provide information not visible in the string.

Here is the code from the beginning of `CountryNamesFin`. Very similar code will be needed for instance for French and German.

```
concrete CountryNamesFin of CountryNames =
  open SyntaxFin, ParadigmsFin, Prelude in {

lincat CName = LocName ;

oper LocName = {np : NP ; isIn : Bool} ;

oper mkCName = overload {
  mkCName : Str -> LocName = \s -> {np = mkNP (foreignPN s) ; isIn = True} ;
  mkCName : N -> LocName = \n -> {np = mkNP n ; isIn = True} ;
  mkCName : NP -> LocName = \np -> {np = np ; isIn = True} ;
  } ;

 exCName : LocName -> LocName = \name -> name ** {isIn = False} ;
 sgCName : LocName -> LocName = \name ->
   name ** {np = forceNumberNP singular name.np} ;
```

The code still gets a lot for free from the RGL, but does need some GF features of a "lower level" than mere applications of functions:

- The RGL module `Prelude` is opened to provide `Bool` of **booleans** and its elements `True` and `False`.
- The linearization type of `CName` is a **record type** with two **fields**: one for a noun phrase with **label** `np`, another for a boolean with label `isIn` (defining whether to use "internal" or "external" locative cases).
- The operation `mkCName` is now **overloaded** and covers three functions with the same name but different types. The simplest function needs just a string (with the "most probable" inflection inferred), the next one a noun (`N`), which can be given by any of the `mkN` instance of the RGL, and the most complex one an arbitrary `NP`.
- The definitions of `mkCName` functions are **records** that match the record type `LocName`.
- Two additional operations can be used to change the fields of an already constructed `CName`. The changes are performed by the **record extension** operator `**`, which overwrites (or adds) the values of desired fields in a record.

It is a good practice to use records and record types explicitly in as few places as possible. For instance, in `CountryNamesFin` they are used only in the few `oper` definitions in the beginning of the file. In this way, if we need to change the type of `CName` — which is common under the development phase of a grammar — we only need to change these `oper`s and not all over the place in the lexical items.

Here are some examples of how the enriched set of constructors is used:

```
lin Bucharest_CName = mkCName "Bukarest" ;
  -- no changes needed
lin Finland_CName = mkCName (mkN "Suomi" "Suomia") ;
  -- select a less common paradigm
lin Marshall_Islands_CName =
  exCName (mkCName (mkNP thePl_Det
      (mkN "Marshallin" (mkN "saari" "saaria")))) ;
  -- genuinely plural noun phrase, preposition "on"
lin Russia_CName = exCName (mkCName "Venäjä") ;
  -- one says "on Russia" in Finnish!
lin United_States_of_America_CName =
  sgCName (mkCName (mkNP thePl_Det (mkN "Yhdysvalta"))) ;
  -- plural inflection, singular agreement
```

Most of the strings give the right properties "out of the box" and hence need

no changes in the geneted file. But a manual inspection round is needed to make sure that all names get their right grammatical properties.

The top module `CountriesFin` has to treat `CName`, with its enriched type, in a correct way. In the following code, we have omitted the attribute definitions, which pose no particular problems.

```
concrete CountriesFin of Countries = FactsFin, CountryNamesFin **
  open SyntaxFin, ParadigmsFin, SymbolicFin, Prelude in {

lin
  cName name = name.np ;

  populationFact cname int =
    mkCl cname.np (mkV2 (caseV (locCase cname) have_V2))
      (mkNP <symb int : Card> (mkN "asukas")) ;
  continentFact cname name =
    mkCl cname.np (SyntaxFin.mkAdv (casePrep (locCase name)) name.np) ;

oper
  locCase : LocName -> Case = \name -> case name.isIn of {
    True => inessive ;
    False => adessive
    } ;
}
```

The following explanations refer to Finnish, but are relevant for other languages as well:

- `cName` must now pick the `np` field of its argument to be type-correct.
- The locative case is extracted from the `CName` by the `locCase` operation. It would be possible to have the case directly as a field of `LocCase`. However, a boolean parametre is preferrable, because in Finnish it affects many other things besides the inessive/adessive choice.
- The definition of `locCase` uses a **case expression**, which selects different values depending on an argument — here depending on a boolean.

2.3.8 Exercises

Add a language

Add some fact rules

Vary the data

2.4 Stage 3: Building a fluent text

We have by now generated sentences for each atomic fact separately. Each country is thereby described by five sentences. The repetitive text this gives has been slightly improved by specialized expressions for continent and population facts. But a lively text requires more radical measures:

- replacement of names by **referring expressions**, such as pronouns (*it*) and definite descriptions (*the country*, *this European country*);
- **aggregation** of several facts to single sentences: *France is a European country with 66628000 inhabitants*;
- **text layout** with proper punctuation and capitalization, possibly with **markup** such as HTML tags;
- **variation** so that country descriptions can look different for different countries, or for the same country at different occasions.

In this section, we will go through these techniques one by one. The resulting code is found in `facts3/`. This directory also contains the grammar rules used in the next section. We will no longer show entire modules in the text, but just the individual rules under discussion.

2.4.1 Referring expressions

2.4.2 Syntactic aggregation

2.4.3 Unlexing and markup

hover for referent

2.4.4 Creating alternative renderings

2.5 Stage 4: Data aggregation and content planning

Until now, we have built sentences and texts that accurately represent all information in a dataset. This is something that NLG in any case has to be able to do, at least on demand. The result is texts from which the GF parser could actually reconstruct the original data.

But in many NLG tasks, the relation to the dataset is less direct. We want things such as

- **data selection**: show only data considered interesting, not all data;
- **data trimming**: **rounding** of population *30327877*, to *30 million*, and **conversion** of *147181 square kilometres* to *56827 square miles*;
- **data aggregation**: build sentences that are *valid consequences* from the data but not explicity shown there, for instance, *there are 45 countries in Asia*.

Most of the work in these categories is done on the Python side: what is needed in GF is just some new categories and functions which enable expressing these derived facts. Following the overall structure of GF-NLG, Python functions analyse data to find interesting derived facts and convert them to abstract syntax trees, whereupon GF takes over and linearizes the trees to different languages. A slight difference from the earlier is that the choice of trees can be language dependent — for instance that the English text uses miles but the Finnish text kilometres.

A common term for the task of choosing what data to show is **content planning**. Much of NLG research has focused on how to automate this task. For instance, in creating stock market reports from numeric data, one has to find interesting trends fast to be able to report them before it is too late.

However, the Abstract Wikipedia project cannot fully rely on automation, since human judgement is needed to decide what is interesting and relevant to show. And not only this: human judgement is needed in **text planning** as well, to select truly readable ways to express the selected data. Hence attention has to be paid to **interaction**, where a human author can easily create multilingual texts with the help of algorithms but in full control over them.

2.5.1 Automatic planning

2.5.2 Manual planning and document authoring

2.6 Sharing code between languages via a functor

The concrete syntax of `Facts` in `facts3` is built by using a **functor**, a.k.a. a **parameterized module** or an **incomplete module**. What makes it incomplete is that it imports one or more **interfaces**, which just declare the types of some functions but do not give their definitions. The definitions are given in **instances** of interfaces. The RGL is built in such a way that the modules `Syntax` and `Symbolic` are interfaces, which have different instances in different languages. We have already used this fact implicitly, when we have copied much of the code from English to Finnish, which is possible because the RGL modules for both have the same interface. But at this stage, we make a more systematic use of this, which makes it easier to extend the grammar with new functions and new languages.

A functor module looks as follows:

```
incomplete concrete FactsFunctor of Facts =
  open Syntax, Symbolic in {

lincat
  Doc = Text ;
  Sentence = S ;
  Fact = Cl ;
  ...
lin
  OneSentenceDoc sent = mkText sent ;
  AddSentenceDoc doc sent = mkText doc (mkText sent) ;
  ConjSentence a b = mkS and_Conj a b ;
  FactSentence fact = mkS presentTense positivePol fact ;
  ...
```

Except for the header, it looks *exactly* like a normal concrete syntax. The differences in the header are

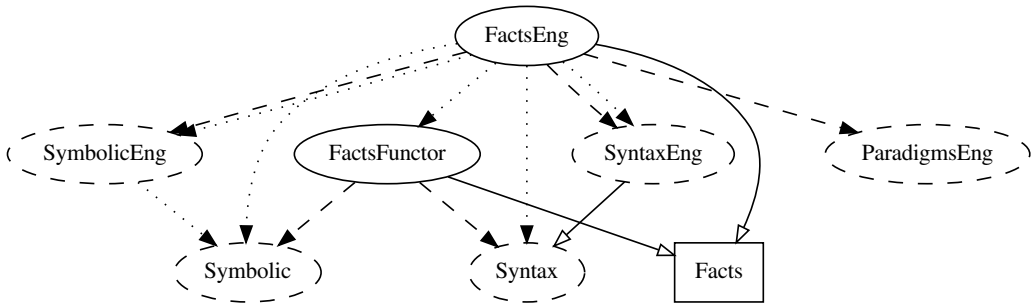
- the keyword `incomplete`,

- opening interfaces rather than complete resource modules.

The **instantiation** of a functor looks as follows:

```
concrete FactsEng of Facts = FactsFunctor with
  (Syntax = SyntaxEng),
  (Symbolic = SymbolicEng)
  ;
```

That is, it only needs to indicate what instances is interface has.



Functors help extending a grammar in both the abstract and concrete dimensions:

- to add an abstract function, its concrete syntax only needs to be defined in the functor,
- to add a language, its concrete syntax needs only a few lines of trivial GF code.

For example, a German concrete syntax can be written

```
concrete FactsGer of Facts = FactsFunctor with
  (Syntax = SyntaxGer),
  (Symbolic = SymbolicGer)
  ;
```

But what to do if the functor definitions are not good enough for a language. They are guaranteed to be grammatically correct (modulo the correctness of the RGL), but can be clumsy. For these situations, one can use **restricted inheritance** from the functor, simply excluding some of its definitions and replacing them by the desired ones:

```
concrete FactsFin of Facts =
  FactsFunctor - [AttributeFact]
  with
    (Syntax = SyntaxFin),
    (Symbolic = SymbolicFin) **
  open ParadigmsFin,(E=ExtendFin) in {

lin
  AttributeFact attr obj val = mkCl (mkNP (E.GenNP obj.np) attr) val ;
}
```

This forces the deviant rule for attribute facts in Finnish, which was discussed above.

In our experience, restricted inheritance is used only for a small minority of functions in different languages. Hence the use of functors is a productive way to build multilingual grammars. If moreover the translations of names can be extracted from data, adding a new language is almost an automatic task — provided that it is included in the RGL.

2.7 Where is the standard NLG pipeline?

- our presentation has followed an opposite order: planning last

Chapter 3

Generation from formal languages

- a more systematic view of second-order data is logical formulas that can be treated as data
- in earlier project, content-planning has been taken for granted, as the formal structures have been given

3.1 Predicate calculus

- the CADE paper 2011 presents an architecture that has shown useful
- the Haskell part is here rewritten in Python

3.2 Query languages

- SQL and relational databases
- SPARQL and Wikidata
- the CADE architecture in action
- two-way conversions (also from NL to formal)

3.3 Specification languages

- OCL and Z
- examples of really large documents

- engineer, manager, and customer views
- the CADE architecture in action
- two-way conversions (also from NL to formal)

3.4 CNL as a formal language

- e.g. Attempto Controlled English (ACE)
- as formal as any, if we have a parser — just easier for a layman to read or write
- essentially what is used in `facts3/country_facts.country_texts_parsed`

Chapter 4

An API for NLG functionalities

- documentation of the most mature state reached in the project
- morphological lexica
- semantic multilingual lexica
- semantic RGL level, e.g. tenses, referring expressions
- CNL for fact documentation