Grammatical Framework for Python programmers

September 18, 2017

DISCLAIMER: I AM NOT A PYTHON PROGRAMMER BUT I'LL TRY MY BEST TO EXPLAIN CONCEPTS IN A WAY THAT SHOULD BE FAMILIAR TO PEOPLE WHO KNOW PYTHON

This short tutorial is aimed at Python programmers who would like to learn how to use the Grammatical Framework. It tries to meet you in familiar terrain and take you gently over to the way GF works. By the end of this tutorial you won't know all of GF, but you will be able to consume the regular GF documentation with much less friction: think of this tutorial as an English speaker learning a little bit of Italian before learning Latin.

1 Types

1.1 Types in Python

As a Python programmer you usually don't think too much about types. Unfortunately in GF types much more important. Python is, by default, dynamically typed. GF is statically typed. So we'll start by going over Python's type system, and relate that to GF's type system.

1.2 Dynamic typing in Python

Basic types in Python consist of Strings and Numbers (Integer, Float, Complex). A special case are boolean (logical expression) that will be explained later. You can ask Python to give you the type for expressions. These types are automatically inferred – we don't have to tell Python what type a variable has, as can be seen in the last example below.

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
```

```
>>> type("Foo")
<class 'str'>
>>> type(complex('1+2j')) # Is that still considered basic?
<class 'complex'>
>>> a=3
>>> type(a)
<class 'int'>
```

These basic types can be used in compound types. Compound types are among others Lists, Tuples and Dictionaries. These again can be part of other compound types as well, e.g. list containing lists as elements. Python does not really enforce that all elements of a list have the same type (Disclaimer: In Python 3.5 they introduces strict typing and type annotations to solve that problem)[1] as you can see in the examples. To access elements in compound objects we can use the `[]` operator.

```
>>> type([])
<class 'list'>
>>> type([1,2,3])
<class 'list'>
>>> type([1,2,"foo","bar"])
<class 'list'>
>>> type(())
<class 'tuple'>
>>> type((1,2))
<class 'tuple'>
>>> type((1,2,"foo"))
<class 'tuple'>
>>> type({})
<class 'dict'>
>>> type({'foo':1,'bar':2})
<class 'dict'>
>>> type({'foo':1,'bar':'baz'})
<class 'dict'>
```

A special case: Boolean values. Python's Boolean "datatype" simply uses 1 for True and 0 for False:

```
>>> True + True
2
```

Python interprets values of other types as truthy or falsy values:
The values considered **false** are:

- `None` i.e. the empty object

[1] `https://docs.python.org/3/library/typing.html`

- `False` i.e. the logical constant
- `zero` of any numeric type, e.g. `0, 0.0, 0j`.
- any empty sequence, e.g.`'', (), []`, i.e. empty string, tuple, or list.
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a ___bool___() or ___len___() method, when that method returns the integer zero or bool value False.

Everything else is considered **true**.

Another interesting group of datatypes are enumeration types where you define a type by listing all possible values. In Python enumerable types are objects of class `enum`. They can also be used as keys in dictionary. That gives us a way to express a mapping from grammatical number and case to a word form for german nouns.

Enumeration types are useful in linguistics to represent inflection: languages inflect words by gender, case, number, tense, and so on. We can set up a variety of enum classes for each inflection.

```
>> class Number(Enum):
...   Sg = 1 # singular
...   Pl = 2 # plural
...
>>> man={Number.Sg:"man",Number.Pl:"men"}
>>> man[Number.Sg]
'man'
>>> class Case(Enum):
...   Nom = 1
...   Gen = 2
...   Dat = 3
...   Acc = 4
...
```

Dictionary keys can be numbers, and enums are numbers. So enums can be used as dictionary keys. Now we can express the declension of German nouns in a dictionary:

```
>>> mann={Number.Sg:{Case.Nom:"Mann",
...                  Case.Gen:"Mannes",
...                  Case.Dat:"Mann",
...                  Case.Acc:"Mann"},
...       Number.Pl:{Case.Nom:"Männer",
...                  Case.Gen:"Männer",
...                  Case.Dat:"Männern",
...                  Case.Acc:"Männern"}
... }
```

```
>>> mann[Number.Sg][Case.Gen]
'Mannes'
```

Other languages might call this data structure a hash, or a map, or an associative array, or a record. Python calls it a dictionary. Linguists also call it a dictionary. Isn't that nice?

Exercise 1. *If you haven't done so before, try the type() function in Python on different values and compare the output.*

One problem with this approach is that Python does not enforce that we define mappings for all possible values. This can easily lead to errors. In the next example we only define values for some of the keys and then try to access undefined values which leads to an error.

```
>>> mann={
...   Number.Sg:{
...     Case.Nom:"Mann"
...   },
...   Number.Pl:{
...     Case.Dat:"Männern"
...   }
... }
>>> mann[Number.Sg][Case.Gen]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: <Case.Gen: 2>
```

An aside: a mathematician would say that this abbreviated dictionary represents a partial function, as opposed to a total function, and the error here is due to the lack of totality. The mathematician thinks of the dictionary as a function in the sense that it maps input (grammatical number and case) to output (a German word).

So that brings us to actual Python functions. Functions in Python are usually defined and given a name with the `def` keyword. But Python also supports so-called anonymous functions or lambda expressions. Below the successor function is expressed in three different ways.

```
# 1. Plain old function definition of succ1.
>>> def succ1(x) :
...   return x+1
...
>>> type(succ1)
<class 'function'>

# 2. The variable succ2 contains a lambda function.
```

```
>>> succ2 = lambda x : x+1
>>> type(succ2)
<class 'function'>

# 3. We don't even bother to give it a name.
>>> type(lambda x: x+1)
<class 'function'>
```

1.3 Static typing in GF

Static types are more complicated. There are more rules. You will find the same kinds of rules in any statically typed language. GF's type system will be familiar to Haskell or Rust programmers; but this tutorial is intended for Python programmers, so we will assume no familiarity with that tradition. As static types come to Python, Python programmers will have to climb much the same learning curve as what follows here.

GF's primitive types will be familiar from Python.

Python has strings (called `str`). GF has strings (called `Str`).

Python has integers (called `int`). GF has integers (called `Int`).

Python has floating-point numbers (called `float`). GF's are called `Float`.

Python has dicts (of the form `{ 'foo': 123 }`). GF has records (of the form `{ foo: 123}`). You get the 123 out of a Python dict by saying `{'foo':123}['foo']`. You get the 123 out of a GF record by saying `{ foo:123 }.foo`.

How do you explicitly tell Python, or GF, for that matter, about types?

In Python, you use type hints. Instead of saying

```
def add(amount1, amount2):
    return amount1 + amount2
```

you would say

```
def add(amount1: int, amount2: int) -> int:
    return amount1 + amount2
```

Mypy has good documentation, and you can refer to PEP 484 for full details.

In GF, the equivalent function definition might read:

```
def add : Int -> Int -> Int
add amount1 amount2 = amount1 plus amount2
```

The left two `Int`s above describe the arguments to `add`, and the rightmost `Int` describes the return value.

1.4 Linguistically speaking

GF code is organized into modules.

Abstract modules are language-independent, and describe the semantic categories and linearization relationships at a very high level. If you have the idea that "Italian wine is delicious", you might break that down as a predicate sentence (x y is z) which has an adjectival noun phrase (x = Italian, y = wine), a copula ("is"), and an adjective (z = delicious).

The abstract grammar would concern itself primarily with outlining the parts of speech (x, y, z) and how they fit together (e.g. nouns can be modified by adjectives), and only mention in passing that wine is a particular instance of a noun, and Italianness and deliciousness are adjectival modifiers of interest.

Concrete modules are where the language-specific details appear: if your application translates between English and Italian, you would have two concrete modules. One module knows that wine is spelled "wine" and the other knows that wine is spelled "vino".

Resource modules are a good place to factor out auxiliary functions that are common to English and Italian, so you don't have to repeat yourself. For example, the idea of negation could appear in a resource module: corked wine is *not* delicious, in any language.

Because we have abstract modules, concrete modules and resource modules we have different types that are important in different situations.

In the abstract syntax we do not really have to care about types. Here we think linguistically and treat it more or less like context-free grammars, i.e. categories and syntax rules.

So we will start our examination of GF types by looking at resource modules. Here we can use all the different types available in GF. These include Token lists (a.k.a. arrays of Str), Integers, Floats, and Bool. Most of them are defined in a module called Predef but Bool is defined in the module Prelude. Anonymous function types (lambdas) are standard types in GF. In resource modules, functions are defined as lambdas. (In practice, two equivalent syntaxes are available, just as, in the Python examples above, succ1 and succ2 are equivalent.)

```
resource SimpleTypes = open Predef,Prelude in {
  oper
    s : Str = "foo" + "bar" ;
    st : Str = "foo" ++ "bar";
    i : Predef.Int = 42;
    f : Predef.Float = 23.5;
    b : Bool = False ;
    succ : Int -> Int;
    succ = \i -> plus i 1 ;
}
```

As you can see in the examples above, for each variable we first give the type. That

is best practice but in simple cases not necessary because GF can infer the type automatically. The only case where the type must be given is the function. But it is good discipline to annotate your types: it is like brushing your teeth.

1.5 Compile-Time Tokens vs Run-Time Strings

This is as good a time as any to point out the difference between the operators `+` and `++`. This is a common source of problems. GF scrupulously observes the difference between compile-time string operations and run-time string operations. But this distinction is more obvious to GF than it is to you, and can be the source of mysterious errors.

GF's idea of a sentence is a list of strings. In Python, a list of strings is written `['Italian','wine','is','delicious']`.

In GF, it is written `"Italian" ++ "wine" ++ "is" ++ "delicious"`. Why no square brackets? Because GF isn't a general purpose programming language: you hired it to do a very specific job in computational linguistics, so GF's whole world is sentences and words. But if you really miss the square brackets, then as syntactic sugar, you could also say `["Italian wine is delicious"]`.

The `++` operator concatenates lists of tokens. The `+` operator glues two strings into a single token. What's a token? A token is a discrete word in a sentence.

```
"potato" = "po" + "ta" + "to"
```

Now for the dreaded compile-time string token rule: GF requires that every token – every separate word – be known at compile-time. Rearranging known tokens in new ways, no problem: GF can generate an infinite variety of different combinations of words. That's its job. Coleridge reminds us that "prose is words in their best order" – but GF is quite willing to play with words in any order your grammar can devise.

But they have to be words known to GF at compile-time. GF is not improv: as Shakespeare might have said, if anybody's going to make up new words around here, it'll be the playwright, not the actor. You can `+` tokens together but only at compile-time. If you try to do it at run-time, you will get weird errors, like `unsupported token gluing` or, worse, `Internal error in GeneratePMCFG`.

This is very different to what Python does: Python quite happily manipulates strings at any time, because to Python, strings are just arrays of characters. Space is just another character. But to GF, words carry meaning; and run-time is too late to make up new words and new meanings.

So how do you know whether a line of code is executing at compile-time or at run-time?

We'll get to that later.

1.6 Lambda Abstractions

Back to the above code example, the last line – the function – is esentially a lambda expression like the one used in Python. The difference is that Python's keyword `lambda` is shortened to a backslash (\), and the colon that separates the parameter from the

function body is replaced by an arrow (`->`). The same arrows appear in Python's type hints for functions.

```
> cc s
"foobar"
0 msec
> cc st
"foo" ++ "bar"
0 msec
> cc b
Prelude.False
0 msec
> cc i
42
0 msec
> cc f
23.5
0 msec
> cc bar "foo"
"foobar"
0 msec
```

1.7 Compound Types

In GF can compose types to create new types. There are a few ways to construct compound types: records, tables and function types. GF expects you to *declare* the type, which amounts to saying "this is the shape of the thing". You can then *define* a variable that matches the type.

Both tables and records can be seen as special forms of dictionaries. Tables are mappings from grammatical features to other values. A simple example are inflection tables that map e.g. from gender and number to the correct word form. Records usually are used to keep subparts of phrases as well as inherent grammatical features like the gender in nouns.

```
cat    -- from the abstract grammar
  Food;
fun
  Potato, Kartoffel : Food;

 -- from the concrete grammar for English, FoodEng
lincat
  Food  = { s : Number => Str };
  -- in English, nouns are inflected by number
```

```
  -- from the concrete grammar for German, FoodGer
  lincat
    Food = { g : Gender ; s : Number => Case => Str }
    -- in German, nouns have inherent gender and are inflected by case and number
```

In Python, you're probably used to sounding out types in your head, like "an array of dicts from string to int".

Can you sound out GF's types?

In English, the "Food" type is a record with a field `s` whose value is a table keyed by Number returning a String.

In German, the "Food" type is a record with a field `s` whose value is a Gender, and a field `s` whose value is a table keyed by Number returning a table keyed by Case returning a String.

It is important to be able to sound out GF's types in your head, because the vast majority of compile-time frustrations in statically typed languages have to do with getting the types wrong; but once you do get the types right, everything else seems to fall into place.

In Python, once you have a class defined, you can start instantiating objects into that class.

In GF, once you have a type *declared* you can *define* a variable that instantiates the type.

First you set it all up, then you pull out what you want.

In Python, once you have an object defined, you can start accessing values from it: `mything.get('attrName')`

In GF, once you have a complex data structured defined, you can pull values out of it using `.` (for records) and `!` (for tables).

From earlier in this tutorial, we know that records are basically dictionaries. Given a key, you reach into the record and pull out the value corresponding to the key. Record keys must be known at compile-time.

Unlike Python, you can't put a record key into a variable. Python lets you say

```
mykey = 'foo'
{ 'foo' : 2, 'bar : 4 }[mykey]
```

But you can't do that in GF.

Functions map inputs to output via a bunch of arbitrary computation. Given some input, a function thinks hard and returns some result.

What's a table? Something in between.

A table is a bit like a function: functions use `\` while tables use `\\`. Functions use `->` and tables use `=>`. These similarities are not accidental.

A table is a bit like a record: records use `.` while tables use `!`. Records are constructed by `{ foo = bar }` while tables are constructed by `table { foo => bar }`. These similarities are also not accidental.

You can think of a table as a fancy case statement. What's a case statement? In Python, switch, or case, statements are ... um, not gonna happen. Oops! Is Python the only language that doesn't have a case statement?!

Never mind, we will explain case statements from first principles. It's a common pattern: given a thing, you step through an ordered list of alternatives, testing each alternative to see if the thing is equal to it. When a match is found, control branches accordingly. Here's what a case statement looks like in C++. It takes an int and prints something to output.

```
switch(1) {
    case 1: { int x = 0;
              std::cout << x << '\n';
              break;
            }
    case 2 : cout << '2';
             break;
    default: std::cout << "default\n"; // no error
             break;
}
```

In this C++ example the input to the switch was hardcoded but in practice you would typically use a variable in place of the `1`.

In GF, a table lets you step through an ordered list of patterns, testing your string against each pattern. When a match is found, the relevant computation is returned. But the test is not always an equality test: it can be a regexp-like test.

In fact, case statements in GF are syntactic sugar for tables. A case statement sets up a table and immediately looks up your given key in it. The GF tutorial offers a great example of a table/case statement under smart paradigms.

Computer scientists call tables "finite functions": as the GF reference manual says, *it is possible to finitely enumerate all argument-value pairs; this, in turn, is possible because the argument types are finite.*

Hence the rule: values in tables all have to have the same type while records can contain values of different types.

The keys in records must be known in advance, and are matched exactly, while tables perform pattern-matching against the "lookup key".

To complete the picture, let's talk about functions. If records are accessed using '`.`' and tables are looked-up using '`!`', then functions are applied using ' ' – a space.

In Python, functions are called with parameters: `myfun(arg1, arg2)`.

In GF, that looks like: `myfun arg1 arg2`. The syntax descends from the mathematical tradition via the lambda calculus and Haskell.

To consolidate, let's look at some examples.

```
resource Comparison = open Prelude in {
  oper
    -- first we declare types
```

```
    myrecord : { one   : Predef.Int ;
                 two   : Predef.Int ;
                 three : Predef.Int ;
                 four  : Predef.Int };
    mytable  : Str => Predef.Int;
    myfunc   : Str -> Predef.Int;

    -- then we define values
    myrecord =        { one = 1 ; two = 2 ; three = 3 ; four = 4 };
    mytable  = table { "one"   => 1
                     ; "two"   => 2
                     ; "three" => 3
                     ; "four"  => 4
                     ; _       => 0 };
    myfunc s = ifTok Predef.Int s "one"   1   (
               ifTok Predef.Int s "two"   2   (
               ifTok Predef.Int s "three" 3   (
               ifTok Predef.Int s "four"  4 0)));

}
```

A syntax note: unlike Python, every statement in GF has to be terminated with a ; (semicolon). Semicolons are also used as record and table separators. In Python the separator is a comma, not a semicolon.

output:

```
> cc myrecord.one
1

> cc mytable ! "two"
2

> cc myfunc "three"
3
```

It is much more natural to do what is effectively a table lookup using, well, a table. In fact GF's `if_then_else` control structure is ultimately implemented using a case statement, which in turn is really a table.

So, what's the difference here? Both tables and functions can handle arbitrary input of the correct type, but you can't dereference a record label that doesn't already exist.

```
> cc mytable ! "asfd"
0
```

```
> cc myfunc "bogus"
0

-- under construction: let us talk about the three functiony bits of GF:
  lin
  def
  oper
```

In the last kind of modules (we want to look at here) are concrete modules. Here we can use most of the things we have seen for the resource modules. But in the end it boils down to using strings in different ways. We store strings in record fields, select the right strings from tables and put them together at the right point. Sometimew we need to additionally store grammatical features. So we need strings, tables, records, and `param` types.

2 Context-free Grammars

A simple form of grammars are the so-called context-free grammars. They can be used in lots of applications both in computer science and linguistics but have limitations that encourage us to use more expressive formalisms like GF. We start with simple grammars in Python with NLTK, then show to translate them to a format that also works in GF and finally show how to implement them in the full GF formalism.

2.1 Context-free Grammars in Python/NLTK

```
from nltk import CFG,ChartParser
from nltk.tokenize import SpaceTokenizer
grammar = CFG.fromstring("""
  S -> NP VP
  NP -> Det N
  VP -> IV
  Det -> 'the'
  N -> 'man'
  IV -> 'walks'
  """)
#>>> grammar
#<Grammar with 14 productions>
#>>> grammar.start()
#S
#>>> grammar.productions()
#[S -> NP VP, NP -> Det N, VP -> IV, Det -> 'the', N -> 'man', IV -> 'walks']
parser = ChartParser(grammar)
parses = parser.parse_all(SpaceTokenizer().tokenize("the␣man␣walks"))
#>>> parses
```

```
#[Tree('S', [Tree('NP', [Tree('Det', ['the']), Tree('N', ['man'])]), Tree('VP
```

```
from nltk import CFG,ChartParser
from nltk.tokenize import SpaceTokenizer
grammar = CFG.fromstring("""
  S -> NP VP
  NP -> Det N
  VP -> IV
  Det -> 'the'
  N -> 'man'
  N -> 'men'
  IV -> 'walks'
  IV -> 'walk'
  """)
parser = ChartParser(grammar)
parses = parser.parse_all(SpaceTokenizer().tokenize("the␣man␣walk"))
#>>> parses
#[Tree('S', [Tree('NP', [Tree('Det', ['the']), Tree('N', ['man'])]), Tree('VP
```

```
from nltk import CFG,ChartParser
from nltk.tokenize import SpaceTokenizer
grammar = CFG.fromstring("""
  S -> NP_Sg VP_Sg
  S -> NP_Pl VP_Pl
  NP_Sg -> Det N_Sg
  NP_Pl -> Det N_Pl
  VP_Sg -> IV_Sg
  VP_Pl -> IV_Pl
  Det -> 'the'
  N_Sg -> 'man'
  N_Pl -> 'men'
  IV_Sg -> 'walks'
  IV_Pl -> 'walk'
  """)
parser = ChartParser(grammar)
parses = parser.parse_all(SpaceTokenizer().tokenize("the␣man␣walk"))
#>>> parses
#[]
parses = parser.parse_all(SpaceTokenizer().tokenize("the␣man␣walks"))
#>>> parses
#[Tree('S', [Tree('NP_Sg', [Tree('Det', ['the']), Tree('N_Sg', ['man'])]), Tr
```

Exercise 2. *Write a context-free grammar accounting for the following sentences in Italian:*

Il mio hovercraft è pieno di anguille

Io non acquisterò questo disco, perché è graffiato
TODO
Try to parse the following strings:
The first one should be accepted and the second one rejected.

2.2 Context-free Grammars in GF

```
Sentence    . S    ::= NP VP
NounPhrase  . NP   ::= Det N
VerbPhrase  . VP   ::= IV
Determiner  . Det  ::= "the"
Man         . N    ::= "man"
Walk        . IV   ::= "walks"
```

```
Sentence    . S    ::= NP VP
NounPhrase  . NP   ::= Det N
VerbPhrase  . VP   ::= IV
Determiner  . Det  ::= "the"
Man         . N    ::= "man"
Men         . N    ::= "men"
Walks       . IV   ::= "walks"
Walk        . IV   ::= "walk"
```

```
SentenceSg    . S      ::= NPSg VPSg
NounPhraseSg  . NPSg   ::= Det NSg
VerbPhraseSg  . VPSg   ::= IVSg
SentencePl    . S      ::= NPPl VPPl
NounPhrasePl  . NPPl   ::= Det NPl
VerbPhrasePl  . VPPl   ::= IVPl
Determiner    . Det    ::= "the"
Man           . NSg    ::= "man"
Men           . NPl    ::= "men"
Walks         . IVSg   ::= "walks"
Walk          . IVPl   ::= "walk"
```

Exercise 3. *Write the same grammar from the previous task in GF. Generate all trees accounted for by the grammar. Can you guess how many trees will be generated from a grammar?*

3 Step beyond Context-freeness: Tables and Records

4 Smart paradigms

A smart paradigm in the GF jargon is a function that takes one or a few word forms and uses this information provided to generate the whole paradigm, i.e. the list of all

word forms depending on the grammatical features the word is inflected on.

We can implement this kind of function both in Python and GF.

4.1 Smart paradigms in Python

Exercise 4. *Implement a function that takes a string of a noun and generates the regular noun paradigm for English as a dictionary of dictionaries. Also define all necessary grammatical features as enumeration types*

4.2 Smart paradigms in GF

PATTERN MATCHING

```
resource SmartParadigm = {
  param Case = Nom | Gen | Dat | Acc ;
  param Number = Sg | Pl ;
  oper
    Noun : Type = { s : Number => Case => Str } ;
    smartNoun : Str -> Noun = \n ->
      let um = umlaut n in
      { s = table Number { Sg =>
                             table Case {
                               Nom | Dat | Acc => n ;
                               Gen => n + "es"
                             } ;
                           Pl =>
                             table Case {
                               Nom | Gen | Acc=> um + "er" ;
                               Dat => um + "ern"
                             }
          }
      } ;
    umlaut : Str -> Str = \s ->
      case s of {
        p + "a" + s => p + "ä" + s ;
        _ => s
      } ;
}
```

5 Other problems

- compile-time vs. run-time strings
- ...

6 The GF-Python API

GF ships with a Python runtime: that means your Python program can use the GF API, which is called `pgf`. There is a longer tutorial at `http://www.grammaticalframework.org/doc/python-api.html`. The Python runtime depends on the C runtime. If these runtimes aren't available in the package you downloaded, it is possible to compile them from source. First compile the C runtime, then install the Python runtime; the INSTALL files describe the process.