Grammatical Framework for Python programmers

August 14, 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 aims for 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.

1 Types

1.1 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 even have to help Python to figure out 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 as the form 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 (!) 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 in Python are truth values. Python does not just have a boolean datatype but interprets certain values of other types as truth values.

The values considered **false** are:

- `None` i.e. the empty object
- `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`. Every value is mapped to a natural number. 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.

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

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 aproach is that Python does not enforce that we define mappings for all possible values which can be prone 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>
```

In addition we will have a look at 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 succesor function is defined in two differnt ways

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

1.2 Static typing in GF

Types in GF might seem a bit more complicated. 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 can think linguistically and treat it more or less like context-free grammars, i.e. categories and syntax rules.

So we will start by looking at resource modules. Here we can use all the different types available in GF. These incluse Token lists (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 and in resource modules the only way to define functions.

```
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 = \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 for most of these simple cases it is not really necessary because GF can infer the type automatically. The only case where the type has to be given is the function.

Also there is a difference between the operators `+` and `++`. The reason for that is that GF has a distinction between strings and string tuples as well as between compiletime

strings and runtime strings. For the moment we can say that string tuples are strings that are concatenated with a whitespace and strings are concatenated directly, i.e. `++` puts a space between two words and `+` does not. But `+` can only be used on compile-time e.g. in resource modules or with the exception of `oper`s and `++` can also be used at runtime e.g. when parsing input. If you don't understand that at the moment, that is perfectly fine. We will come back to this later.

Finally the function is esentially a lambda expression like the one used in Python. The difference is that the keyword `lambda` is replace by a backslash (\) and the colon that separates the parameter from the function body is replaced by an arrow (`->`).

```
> 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
```

In GF we also have ways to compose types to create new types. These compound types are called tables, records and function types.

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.

The main difference is that values in tables all have to have the same type while records can contain values of different types.

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

```
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