GF for Python Programmers

Herbert Lange
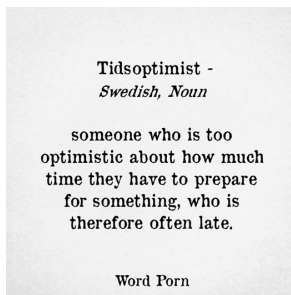
15. August 2017

Public Service Announcement
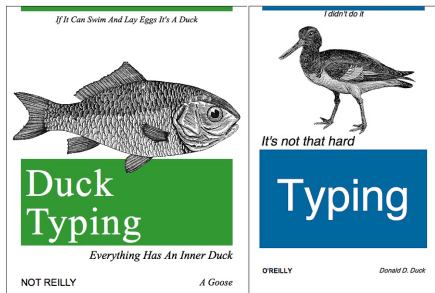
Join the GF IRC Channel #gf on Freenode
You can use the Freenode Webclient
`https://webchat.freenode.net/`

Disclaimer

- ▶ This tutorial is work in progress, you are the first ones to enjoy(?!?) this
- ▶ I am not really a Python programmer myself but I am trying my best to understand the Python way of doing things
- ▶ I hope we can find some connections Python ⇔ GF
- ▶ We use Python 3
- ▶ The slides and a extended tutorial can be found on `https://github.com/daherb/GF-for-Python-programmers`

Tidsoptimist -
Swedish, Noun

someone who is too
optimistic about how much
time they have to prepare
for something, who is
therefore often late.

Word Porn

Types

If It Can Swim And Lay Eggs It's A Duck

Duck Typing

Everything Has An Inner Duck

NOT REILLY   A Goose

I didn't do it

It's not that hard

Typing

O'REILLY   Donald D. Duck

- Python has types but people usually don't care too much about them
- But for GF types are important
- ⇒ You should start to be aware of your types (especially for functions)

Types in Python 1

You can use the `type()` function to figure out the type of expressions in Python

Exercise

Fire up your Python shell and have a look at the types of some expressions.
You can try 3, 3.0, "Foo", [1,2,3], (1,2,3), {'foo':1,'bar':2}, some defined variables, functions and lots of other things.

Types in Python 2

- We can have basic types (e.g. numbers, strings, ...)
- compound or complex types (e.g. lists, tuples, dictionaries, ...)
- types by enumerating possible values
- functions

Enumeration types

- We can define new types by enumerate all possible values
- These values are mapped to e.g. integers
- We can use them to define e.g. grammatical features like number, case, gender, etc.

```
>> class Number(Enum):
...    Sg = 1
...    Pl = 2
...
>> type(Number.Sg)
<enum 'Number'>
```

Dictionaries

- Mapping from values of one type to values of another type
- Access values with the [] operator

```
>>> man = {"s":{Number.Sg: "man" , Number.Pl: "men"},
           "g":Gender.Masc}
>>> man
{'g': <Gender.Masc: 1>,
 's': {<Number.Pl: 2>: 'men', <Number.Sg: 1>: 'man'}}
>>> man["s"][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'
```

Implement a function that takes a string of a noun and generates noun paradigms for English (or a language of your choice) as a dictionary of dictionaries. Also define all necessary grammatical features as enumeration types

Write a function that replaces all vowels with the corresponding umlaut

Functions in Python

- ▶ There are (at least) two ways to define functions in Python
- ▶ the most common one is to use `def` and give them a name directly
- ▶ but you can also define functions without names (anonymous functions)

```
>>> 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'>
```

Exercise

Write some functions both as "def"s and lambda expressions and try them on some parameters.

You can try e.g. functions on strings like the function that puts a suffix onto a string.

Types in GF

Different types in abstract, concrete and resource modules:

- in abstract you can just see it as a kind of context free grammar with grammatical categories and syntax rules or as function types for the concrete syntax
- in resource modules we can use lots of types we already know from other languages
- in concrete syntax we can mostly focus on string tuples, records, tables and parametric types

```
abstract Simple = {
  cat S ; NP ; VP ;
  fun
    sent : NP -> VP -> S ;
}
```

We can read it as a grammar with the three non-terminal symbols
S, NP and VP and the one grammar rule equivalent to the CFG
rule S --> NP VP or a function that takes two arguments, one of
type NP and one of type VP and returns something of type S.
What exactly this function does and what the parameters look like
does not matter at the moment.

```
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 ;
}

> cc s
"foobar"
0 msec
> cc st
"foo" ++ "bar"
0 msec
> cc succ i
43
0 msec
```

Parametric types

Parametric types can be seen as the GF equivalent of enumeration types. They have finite values that have to be enumerable (explicitly or implicitly)

```
resource Res = {
  param
  Number = Sg | Pl ;
  Gender = Fem | Masc | Neutr ;
  Agr = Ag Gender Number ;
}
```

The last one, the agreement on gender and number, implicitly lists all combinations of Number and Gender, e.g. Ag Sg Fem, Ag Sg Masc, Ag Sg Neutr, Ag Pl Fem, ...

Tables and Records

- GF has two different concepts, Tables and Records. In Python both could be replaced with dictionaries (even there are also named tuples in Python)
- Tables are like the dictionaries where we used Enums as keys. The operator ! is used to select a value from a table
- Records are like the dictionaries where we used strings as keys. The operator . is used to select a value from a table

Problem: Does not enforce totality

```
>>> 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>
```

GF does not allow that to happen. Tables have to be "total", i.e.
there must be a mapping for all possible values (but we can use
wildcards)

```
resource Res = {
  param
    Gender = Fem | Masc | Neutr ;
    Number = Sg | Pl ;
    Case = Nom | Gen | Dat | Acc ;
  mann : { s : Number => Case => Str ; g : Gender }
  mann = {
    s = table { Sg => table { Nom => "Mann" ;
                              Gen => "Mannes" ;
                              Dat => "Mann" ;
                              Acc => "Mann" } ;
                Pl => table { Nom => "Männer" ;
                              Gen => "Männer" ;
                              Dat => "Männern" ;
                              Acc => "Männer" }
              } ;
    g = Masc
  }
```

Pattern matching

It is cool magic available in several programming languages but in python you need 3rd party modules

Idea: e.g. if a noun ends in "y", then we replace it with "ies" to form the plural

You can e.g. use pattern matching in `case` statements to implement nice smart paradigms

```
resource Res = {
  param Number = Sg | Pl ;
  oper
    Noun : Type = Number => Str;
    noun : Str -> Noun =
      \s -> table {
        Sg => s ;
        Pl => case s of
             {
               fl + "y" => fl + "ies" ;
               _ => s + "s"
             }
        } ;
}
```

Exercise

Write your own small smart paradigm. As an inspiration you can take the following German noun phrases

Nom	Sg	der junge Mann	die junge Frau
Gen	Sg	des jungen Mannes	der jungen Frau
Dat	Sg	dem jungen Mann	der jungen Frau
Acc	Sg	den jungen Mann	die junge Frau
Nom	Pl	die jungen Männer	die jungen Frauen
Gen	Pl	der jungen Männer	der jungen Frauen
Dat	Pl	den jungen Männern	den jungen Frauen
Acc	Pl	die jungen Männer	die jungen Frauen

GF in Python

- Load pgf module

  ```
  import pgf
  ```

- Load grammar

  ```
  gr = pgf.readPGF("Foods.pgf")
  ```

- Parse sentence: Parsing is a function in the concrete syntax

  ```
  eng = gr.languages["FoodsEng"]
  i = eng.parse("this Italian pizza is very Italian")
  p,e = i.__next__()
  print(e)
  ```

- Generate trees: Generation is a function in the PGF grammar and linearization in the concrete syntax

  ```
  i = gr.generateAll(gr.startCat)
  p,e = i.__next__()
  print(eng.linearize(e))
  ```

Longer tutorial http:
//www.grammaticalframework.org/doc/python-api.html

Exercise
If you have the Python module installed, try to load a grammar, parse a few sentences, generate a few trees, linearize trees, etc.

Check out Universal Dependencies!!!
If you want to know more wait for the next week or talk to
Prasanth