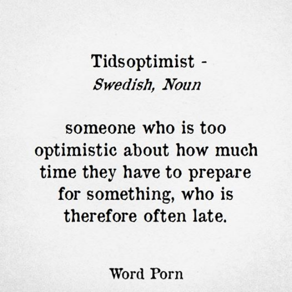
GF for Python Programmers

Herbert Lange

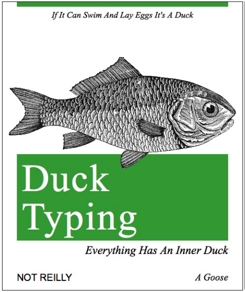
August 15, 2017

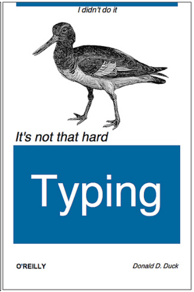
Disclaimer

- 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 $\Leftrightarrow$ GF
- We use Python 3
- The slides and a extended tutorial can be found on `https://github.com/daherb/GF-for-Python-programmers`



Types





- Python has types but people usually don't care too much about them
- But for GF types are important
- $\Rightarrow$ 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'
```

Exercise

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

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.

Types in GF

Different types in abstract, concrete and resource modules:

- in abstract no types in the programming language sense, you can just see it as a kind of context free grammar with grammatical categories and syntax rules
- 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 ;
}
```

Here 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`

```
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 ;
}
```

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

Tables and Records

- GF knows both 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
- Records are like the dictionaries where we used strings as keys

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)

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 phrase

GF in Python

- Load pgf module

```
import pgf
```

- Load grammar

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

- Parse sentence: Parsing is a funtion 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.