Discovery of Pyrser

Lionel "iopi" Auroux

Jeudi, 17/07/2014

Pyrser in few words

- A PEG parser tools in python 3.x
- Available on PYPI (pip install :P)
- Documented (sphinx) : http ://pythonhosted.org//pyrser/
- Tested (699 Unit Test)
- Will be use/Used by students each years (since 2013)
- Inspirated by codeworker (www.codeworker.org)
- Grammars as Classes, so inheritable
- Rules as methods, so overloadable
- Not so context-free (PEG)
- A Type system module (parsing is not enough)
- Still in development... 0.0.3 but completly functionnal (CNORM 4.0)

Plan

- About parsing...
- Motivations
- Use case 1 : parsing XML
- Parsing is not enough, Type System !
- Use case 2 : ToyLanguage4Typing
- Contribution/Conclusion

About parsing...

Since 1970, goal evolves

- From 1 language compiler, 1 translation
- To N entangled languages with N ad-hoc tools
- i.e : Web Stack (HTML+PHP+JS)
- i.e : C++/Doxygen
- i.e : FramaC (C + ACSL)

Emergence of Model Driven Engineering !

Using Parsing/Generating technics mostly in case of MDE for :

- Defining DSLs for ad-hoc tools
- Handle all the stuff
- Generating many things (1-N files)
- Extensibility and Customisation
- Not really a compiler in classical terms...

"Yacc is dead", M. Might, D. Darais, cs.PL 24/10/10 <arXiv :1010.5023>

We need

- WYSIWYG Parser
- Not WYSIWYGIYULR(k)—'what you see is what you get if you understand LR(k)
- Manage Ambiguous grammar
- For engineer not for computer scientist

About parsing...

"Parsing Expression Grammars", B. Ford, 2004, 31st ACM SIGPLAN-SIGACT

PEG provides

- Prioritized choices
- Greedy rules
- Syntactic predicates
- Unlimited lookahead
- Backtracking

More easy and versatile than LR

Motivations

Old EPITA C++2C project

2003 KOOC : Kind Of Objective C

OO features on a superset of C -> need to give a grammar (C) to students !

Very short period (3 month) -> need quick prototyping

Motivations

Evolving since 2003

First version in PERL

From 2005 to 2012 in CODEWORKER (PEG)

Since 2013 in PYTHON

Motivations

A root of many projects

CNORM version 4 in Python with Pyrser

KOOC since 2013

Rathaxes...(MDE apply to driver development)

So Pyrser...

Provides Codeworker features to Python (PEG)

Choose of Python for readability and community

Describes grammar thru a 'BNF-like' description

No embedded language in the description (language agnostic or MVC)

Few abstractions to connect parser engine and scripting language (other language than Python)

Grammar composition

Dependency Injection

Let's apply it to an example : XML

We've got few abstractions

- Rules
- Nodes
- Hooks
- Directives

Use case 1 : Parsing XML

Rules allow us to describe our grammar

```
=           define a non-terminal rule
|           alternatives (lower priority than space)
A           call a non-terminal
'a'         terminal charater
"abc"       terminal text
'a'..'z'    terminal range
[ ]         define groups
?,*,+       classic repeater
some predicates:
~           complement
!           negative lookahead
!!          positive lookahead
-> A        read all until A
```

Some examples :

```

// classic identifier
id = [['a'..'z'|'A'..'Z'|'_']['0'..'9'|'a'..'z'|'A'..'Z'|'_']*]

just_equal = [ '=' !'='] // in C

all_but_eol = [ ~'\n'+ ]
all_but_eol2 = [ ->'\n' ] // read the first \n equivalent to ~'\n'+ '\n'
```

Use case 1 : Parsing XML

Nodes and Hooks allow to interact with python

```python
from pyrser import meta, grammar
from pyrser.parsing import Node

txtbnf = grammar.from_string("""
    plop =[ id:i #test_hook(_, i)]
""")

# txtbnf is a CLASS
@meta.hook(txtbnf)
def test_hook(self, ast: Node, i: Node) -> bool:
    # ast
    # capture value and build a node
    ast.node = self.value(i) # capture value
    return True
itxt = txtbnf()
# rule 'plop' as entry point
res = itxt.parse("cool", "plop")
```

API is richer than this example !

Use case 1 : Parsing XML

Finally, Directives have a global effect

We could understand it with a little example :

```
tag = [
    @ignore("blanks")
    "<" id:nt attr*
    [ "/>"
    |">" [tag|data]* "</" #text(nt) ">"
    ]
]
data = ~'<'+
attr = [ id "=" qvalue]
id = [['a'..'z'|'A'..'Z'|'_']['0'..'9'|'a'..'z'|'A'..'Z'|'_']*]
qvalue = [@ignore("null") string]

string = ['"' ['\\' #char | ~'"']* '"']
// very near the RE \"(\\.|[^"])*\"
```

Use case 1 : Parsing XML

New Directives could be user defined

Grammar are Classes, so inheritable and composable

Rules are overloadable

```
from pyrser.grammar import *
class GramA(Grammar):
    grammar = """ ... """
    entry = "entry_point"
class GramB(Grammar, GramA):
    grammar = """ ... """
    entry = "entry_point"
class GramC(Grammar):
    grammar = """ ... """
    entry = "entry_point"
class GramD(Grammar, GramA, GramC):
    grammar = """ R = [GramA.R | GramC.Z] """
    entry = "GramA.entry_point"
```

Hooks are overloadable

Complete example with JSON in Tutorial I on
http ://pythonhosted.org//pyrser/

Parsing is not enough, Type System !

Parsing is just the way to get an AST but is not enough

We need :

- A conveniance way to do AST visiting (tree transformation, generation)
- To check conformity of AST from a semantic

Parsing is not enough, Type System !

Let's focus on conformity checks (AST visiting is trivial in python)

We need a Type System Module to do the job

Generals goals :

- Classic static typing : declarative or inference
- Versatility
- Extensibility

Remember goals for KOOC :

- Handle OO features : function overloads, kind of subtyping (inheritance is not really subtyping), covariant return types
- Handle C narrow (ugly) typing, implicit conversion, ...

Parsing is not enough, Type System !

Difficult to mimic/use functionnal TS as is (with camlp4)

- How handle ugly things with a functionnal TS ?

Need an expedient, an hybrid TS

- Provide a subset (or reinterpretation) of features found in functionnal language
- A toolbox to compose our semantic
- Usable outside pyrser (legacy parsing) Dependency Injection Again !

Parsing is not enough, Type System !

Types ? a set of syntactic constraints on language expressions.

- We could list them : declarative system
- We could deduce them thru uses : inference system

Our expedient :

- Easy type algebra : operators, statement are all functions
- Only have to find type of functions that use them...
- ...Deduce/Reduce from literals
- ...Deduce/Reduce from operators (polymorphic or not)
- ...Deduce/Reduce from statements (all is expression or not)
- ...Deduce/Reduce from declarations if present
- Polymorphic type (local or type reconstruction)
- Variadic functions are allowed (we want to type C)
- Choose between Implicit or Explicit conversion (no op if subtype)

Parsing is not enough, Type System !

We provide few basic abstractions :

- Scope : universal container (block, namespace...)
- Signature : all things in a scope a identifiable
- Symbol : at the end all is symbol

And more complex abstractions :

- Var
- Val : true, false, 12, "foobar"
- Fun
- Type : ADT meaning

Parsing is not enough, Type System !

We just need to tag our AST with instances of these classes...

And to use generic strategies provide by the Inference submodule to :

- Type check and/or infer types
- Alter tree if TS need it
- And more...

Use case 2 : ToyLanguage4Typing

Let see some examples with a Toy Language...

TL4T is a language define just for Unit/Regression Test and for tutorials

Very easy to understand

Follow some examples illustrate features of the TS

Complete example in Tutorial II & III on http ://pythonhosted.org//pyrser/

Implicit/Explict conversion and subtyping ?

- We provide Type Operation to find if t2 is subtype of t1
- You could also say, implicitly convert t2 to t1 thru function f
- You could also say, prohibe convertion t2 to t1

Use case 2 : ToyLanguage4Typing

```
from tests.grammar.tl4t import *
from pyrser.type_system import *
test = TL4T()
res = test.parse("""
    s = "toto" + 42;
""")
res.type_node = Scope(is_namespace=False)
res.type_node.add(Type("string"))
res.type_node.add(Type("int"))
res.type_node.add(Var("s", "string"))
res.type_node.add(Fun("=", "string", ["string", "string"]))
res.type_node.add(Fun("+", "string", ["string", "string"]))
f = Fun("to_str", "string", ["int"])
res.type_node.add(f)
n = Notification(
    Severity.INFO,
    "implicit conversion of int to string"
)
res.type_node.addTranslator(Translator(f, n))
res.type_node.addTranslatorInjector(createFunWithTranslator)
res.infer_type(res.diagnostic)
print(res.to_tl4t())
```

Will show us :

```
    s = "toto" + to_str(42);
```

Selecting function overloads like in KOOC ?

```
fun f(int, char) -> string;
fun f(int, int) -> double;

var a = f(12, 12); // here var a : double
var a = f(12, 'c'); // here var a : string
```

Use case 2 : ToyLanguage4Typing

How type literals ?

Val could have many types... C

```c
int median(int, int);
...
int b = median('a', 'c');
```

Or Val could have unique fix types... F#

```
12s -> 12 as short
12  -> 12 as int
...
```

We allow both...

Variadic functions ?

```
fun printf(string, ...) -> void;
...
var a = 12;
// here fun printf: (string, int, int) -> void
printf("%d + %d\n", 42, a);
```

We could always access in annoted AST to the define type and the instanciate type ! Useful for polymorphic functions.

- All presented features will be available in the next 0.0.3 Pyrser versions
- Next type checked version of CNORM
- Deadlines (commit on PYPI) next student sessions (september 2014)
- A lot of features in the PIPE : cythonization, effective system, ...
- Migration to GITHUB
- 1 Lead dev + 3 contributors but need contributors...