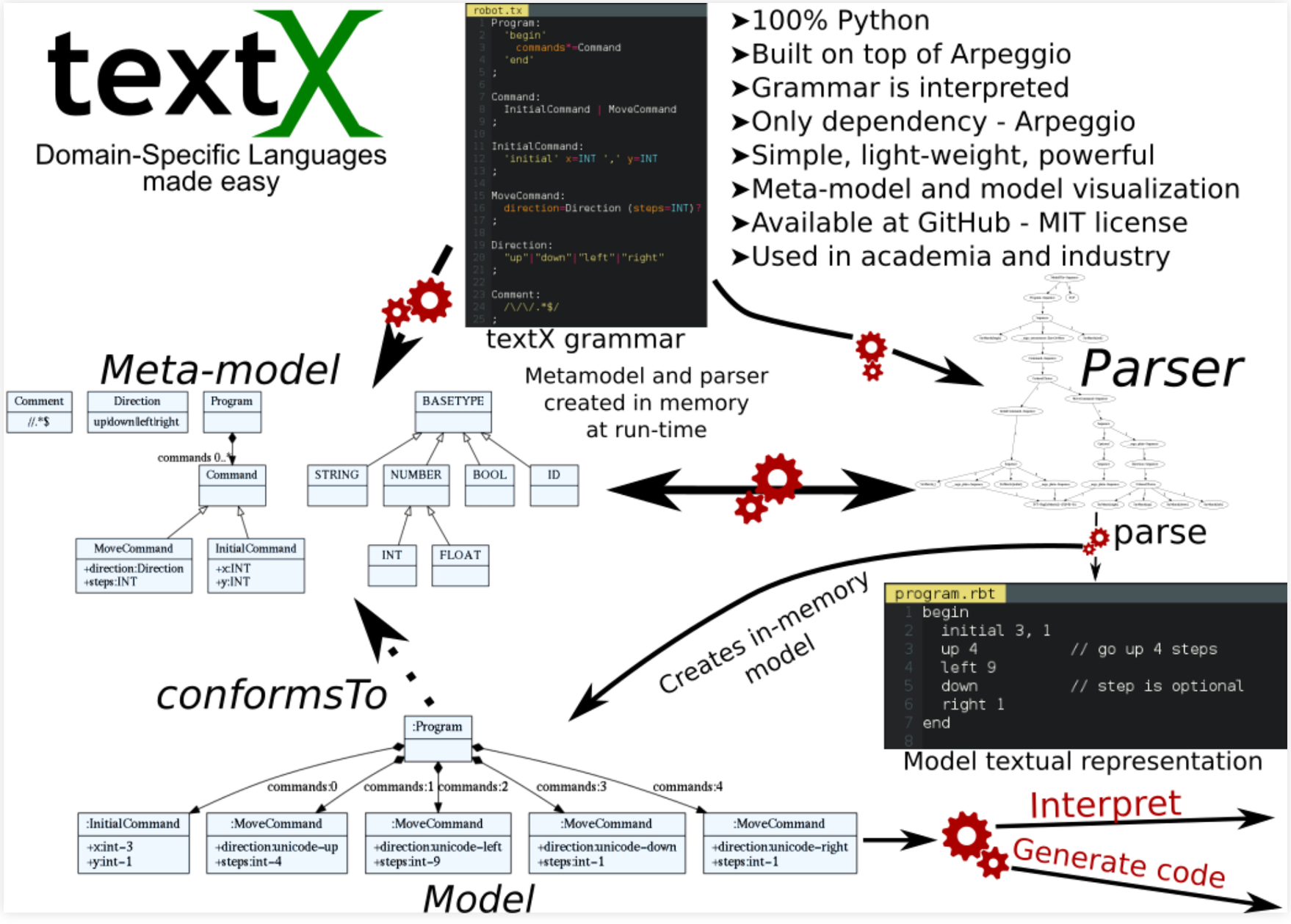
Developing Domain-Specific Languages in Python with textX

Igor Dejanović (igord at uns ac rs)

Strumenta Meetups, 24 February 2022

Table of Content

Overview


textX
Domain-Specific Languages
made easy
robot.tx
Program:
'begin'
commands*=Command
'end'
;
Command:
InitialCommand | MoveCommand
;
InitialCommand:
'initial' x=INT ',' y=INT
;
MoveCommand:
direction=Direction (steps=INT)?
;
Direction:
"up"|"down"|"left"|"right"
;
Comment:
/\/\/.*$/
;
textX grammar
➤100% Python
➤Built on top of Arpeggio
➤Grammar is interpreted
➤Only dependency - Arpeggio
➤Simple, light-weight, powerful
➤Meta-model and model visualization
➤Available at GitHub - MIT license
➤Used in academia and industry
Meta-model
Metamodel and parser
created in memory
at run-time
Parser
parse
Comment
//.*$
Direction
up|down|left|right
Program
commands 0..*
Command
BASETYPE
STRING
NUMBER
BOOL
ID
INT
FLOAT
MoveCommand
+direction:Direction
+steps:INT
InitialCommand
+x:INT
+y:INT
Creates in-memory
model
conformsTo
program.rbt
begin
initial 3, 1
up 4 // go up 4 steps
left 9
down // step is optional
right 1
end
Model textual representation
:Program
commands:0
commands:1
commands:2
commands:3
commands:4
:InitialCommand
+x:int=3
+y:int=1
:MoveCommand
+direction:unicode=up
+steps:int=4
:MoveCommand
+direction:unicode=left
+steps:int=9
:MoveCommand
+direction:unicode=down
+steps:int=1
:MoveCommand
+direction:unicode=right
+steps:int=1
Model
Interpret
Generate code

A brief history

A brief history

- An idea to implement xText in Python

A brief history

- An idea to implement xText in Python
- Initial release in 2014

A brief history

- An idea to implement xText in Python
- Initial release in 2014
- 1.0 - 2016, 2.0 - 2019, 3.0 is going to happen soon

A brief history

- An idea to implement xText in Python
- Initial release in 2014
- 1.0 - 2016, 2.0 - 2019, 3.0 is going to happen soon
- 2017 - Vincent Aranega - pyEcore integration

A brief history

- An idea to implement xText in Python
- Initial release in 2014
- 1.0 - 2016, 2.0 - 2019, 3.0 is going to happen soon
- 2017 - Vincent Aranega - pyEcore integration
- 2018 - Pierre Bayerl joins the project - Bus factor improved :)

A brief history

- An idea to implement xText in Python
- Initial release in 2014
- 1.0 - 2016, 2.0 - 2019, 3.0 is going to happen soon
- 2017 - Vincent Aranega - pyEcore integration
- 2018 - Pierre Bayerl joins the project - Bus factor improved :)
- 570 stars, 64 forks, 18 contributors on GitHub

Setup

To create Python environment

```
python -m venv venv
```

then, activate the environment and install textX.

```
source venv/bin/activate
pip install textx[cli]
```

Hello World

```
model = """
hello You, Me; Everybody
"""
mm_str = r"""
Hello: 'hello' to_greet+=Who[/,|;/];
Who: name=ID;
"""
from textx import metamodel_from_str
mm = metamodel_from_str(mm_str)
m = mm.model_from_str(model)
print([who.name for who in m.to_greet])
```

```
['You', 'Me', 'Everybody']
```

See: matches, assignments, and rule types.

Hello from file

In file `hello.tx`:

```
Hello: 'hello' to_greet+=Who[/,|;/];
Who: name=/[^,;]+/;
```

In file `model.hello`:

```
hello World, Solar System; Universe
```

In file `hello.py`:

```
from textx import metamodel_from_file
def who_processor(who):
    who.name = who.name.strip()
mm = metamodel_from_file('hello.tx')
mm.register_obj_processors({'Who': who_processor})
print(mm)
m = mm.model_from_file('model.hello')
print([who.name for who in m.to_greet])
```

```
<textx.metamodel.TextXMetaModel object at 0x7f5add1b7dc0>
['World', 'Solar System', 'Universe']
```

`textx` command

Used for checking the grammar, investigating languages and generators and running generators.

```
$ textx --help
Usage: textx [OPTIONS] COMMAND [ARGS]...

Options:
  --debug  Debug/trace output.
  --help   Show this message and exit.

Commands:
  check             Check/validate model given its file path.
  generate          Run code generator on a provided model(s).
  list-generators  List all registered generators
  list-languages   List all registered languages
  version           Print version info.
```

Grammar check

```
textx check --grammar hello.tx model.hello
```

```
Error: None:1:1: error: Expected 'hello' at position /home/igor/repos/igordejanovic.github.io/p
```

Running example - Workflow DSL

A tiny workflow DSL will be used in the rest of the slides.

Model/Program

An example in this language might look like:

```
package BuildHouse {
    task buyLand {
    searchAds, chooseLand, buyLand
    next makePlan
    }
    task makePlan {
    searchforArchitect, giveInstructions, choosePlan
    next buildHouse
    }
    task buildHouse {
      buildHouse
    }
}
package BuildFence {
  task buildFence {
    chooseMaterial, buildFence
  }
}
```

Metamodel/Grammar

```
Model: elements*=Element;
Element: Package | Task;
Package: 'package' name=ID '{'
          elements*=Element
    '}';
Task: 'task' name=ID '{'
          steps+=ID[',']
          ('next' next=[Task])?
    '}';
```

Check/use the language

```
from textx import metamodel_from_file
mm = metamodel_from_file('workflow.tx')
model = mm.model_from_file('example.workflow')
print(model)
```

```
<textx:workflow.Model instance at 0x7f655fbd4340>
```

```
textx check --grammar workflow.tx example.workflow
```

```
/home/igor/repos/igordejanovic.github.io/presentations/2022-Strumenta/example.workflow: OK.
```

Visualize (meta-)model

Visualization is done using textX's generator framework.

```
$ textx list-generators
any -> dot          textX[2.3.0]  Generating dot visual...
textX -> dot        textX[2.3.0]  Generating dot visual...
textX -> PlantUML   textX[2.3.0]  Generating PlantUML v...
```

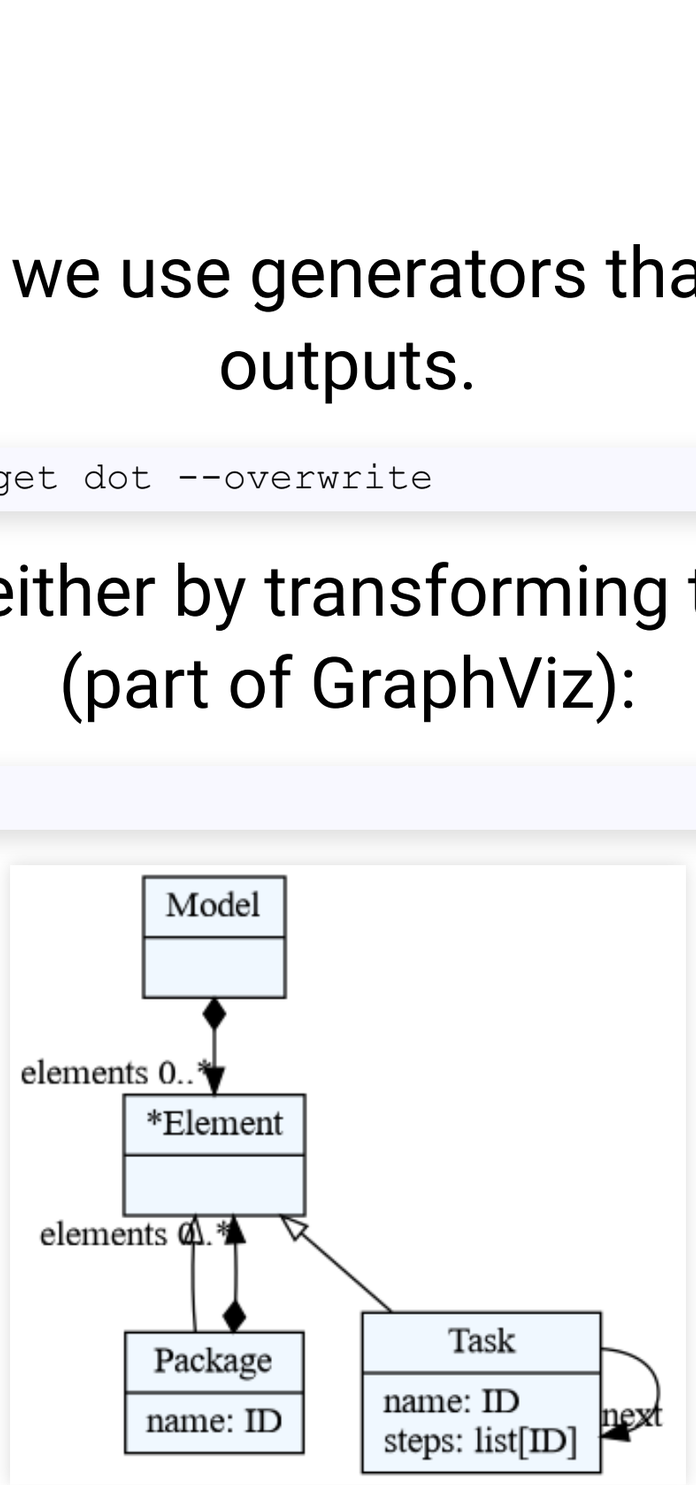
1. PlantUML is a DSL for creating UML diagrams — https://plantuml.com/

To visualize (meta-)model we use generators that produce `dot` or `plantuml` outputs.

```
textx generate workflow.tx --target dot --overwrite
```

`dot` file can be visualized either by transforming to an image using `dot` tool (part of GraphViz):

```
dot -Tpng -O workflow.dot
```



Or opening it in some `dot` visualizer, e.g. xdot.

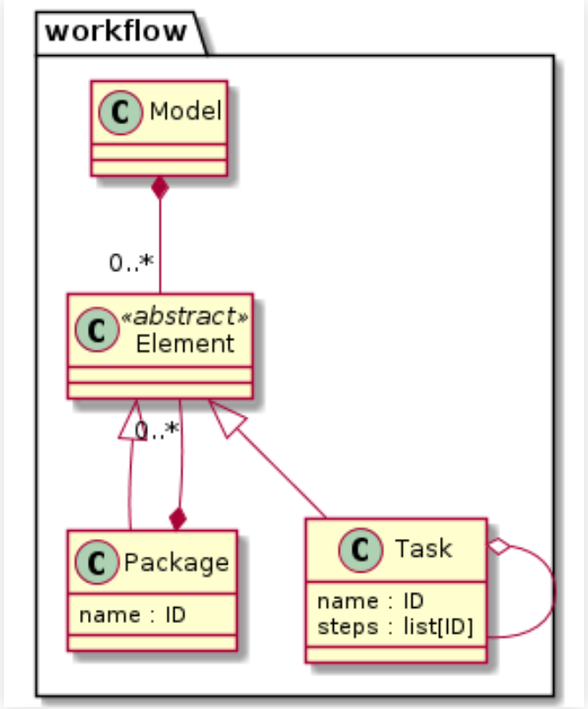
Similarly, we can produce PlantUML diagram by specifying `plantuml` target.

```
textx generate workflow.tx --target plantuml --overwrite
```

```
Generating plantuml target from models:
/home/igor/repos/igordejanovic.github.io/presentations/2022-Strumenta/workflow.tx
-> /home/igor/repos/igordejanovic.github.io/presentations/2022-Strumenta/workflow.pu
    To convert to png run "plantuml workflow.pu"
```

And then convert it to `png` image using `plantuml`:

```
plantuml workflow.pu
```



You get a nice UML diagrams directly from your grammars.

Some differences to xText

A few notes for those familiar with xText.

Lexical grammar

textX doesn’t have a separate lexical grammar. There are Match rules that resembles something close to lexical grammar but not quite.

Assignments

textX integrates repetition and assignments:

In xText you would write:

```
Domainmodel :
    (elements+=Type)*;
```

While in textX it would be:

```
Domainmodel :
    elements*=Type;
```

Optional assignment in xText:

```
static?='static'?
```

In textX:

```
static?='static'
```

Regex matches

textX has simple string matches (like `'something to match'`) and regex matches where you can use a full power of Python regex engine inside `/.../`.

For example:

```
Person:
    name=/[a-zA-Z]+/ age=INT;
```

Repetition modifiers

textX provides a syntactic construct called repetition modifier which enables parser to be altered during parsing of a specific repetition expression.

xText:

```
list_of_ints+=INT (',' list_of_ints+=INT)*
```

textX:

```
list_of_inst+=INT[',']
```

Modifier can also be a regex match:

```
list_of_ints+=INT[/,|;/]
```

Repetition modifier can be applied to any repetition (zero or more, one or more, optional, unordered group).

```
(First /\d+/ Second)*[',']
```

Besides matches there are other modifiers. For example EOL terminator:

```
STRING*[',', eolterm]
```

would match the first line of:

```
"first", "second", "third"
, "fourth"
```

Unordered groups

Xtext support unordered groups using the `&` operator.

```
Modifier:
    static?='static'? & final?='final'? & visibility=Visibility;

enum Visibility:
    PUBLIC='public' | PRIVATE='private' | PROTECTED='protected';
```

In textX unordered groups are specified as a special kind of repetitions. Thus, repetition modifiers can be applied also:

```
Modifier:
    (static?='static' final?='final' visibility=Visibility)#[',']

Visibility:
    'public' | 'private' | 'protected';
```

match:

```
private, static, final
static, private, final
public, static
private
...
```

Scoping

Scoping in textX is done either by using Python through registration of scope providers, or declaratively using Reference Resolving Expression Language.

Xtext provides a Scoping API which can be used by the Xtend code to specify scoping rules.

More differences

For more differences please see this page.

Language/generator registration

Create language description

```
from textx import LanguageDesc

def entity_metamodel():
    # Construct and configure the meta-model
    # e.g. by calling metamodel from file
    ...

entity_lang = LanguageDesc(
    'entity',
    pattern='*.ent',
    description='Entity-relationship language',
    metamodel=entity_metamodel)
```

Programmatic registration

`LanguageDesc` instance can be registered programmatically by the `register_language` function:

```
from textx import register_language
register_language(entity_lang)
```

The meta-model can be accessed from any Python program like this:

```
from textx import metamodel_for_language
lang_mm = metamodel_for_language('entity')
```

Declarative registration

- Registration can be done declaratively using `setup.py` or `setup.cfg`.

```
setup(
    ...
    entry_points={
        'textx_languages': [
            'entity = entity.metamodel:entity_lang',
        ],
    },
```

Using a decorator

There is a convenient `language` decorator to make registration easier.

```
from textx import language

@language('entity', '*.ent')
def entity_lang():
    """
    Entity-relationship language
    """
    # Create, configure and return an instance of the meta-model
    ...
```

Scoping and RREL

- In a link rule reference, the name matched at the location must be resolved to the referenced object.

  For example

  ```
  Attribute: 'attr' ref=[Class] name=ID ';';
  ```

- Global search by default.
- Programmatic scoping providers may be registered to resolve references.

RREL

Declarative specification of reference resolving strategy.

Example:

```
Attribute: 'attr' ref=[Class|FQN|^packages*.classes] name=ID ';';
```

RREL operators and markers

- `.` - *dot navigation*. Searches for the attribute in the current AST context.
  - e.g. `.` is this object, `..` is parent, `...` is a parent of a parent
  - relative lookup. Example: `.a.b`
- `~` - do not consume name.
  - `~extends*.methods` - search for method name in the inheritance hierarchy.
- `*` - repeat/expand - `.~extends*.methods` expands to: `.methods`, `.~extends.methods`, `.~extends.~extends.methods` ...
- `^` - bottom-up search. Example `^packages*.classes` expands to `.classes`, `..packages.classes`, `...packages.packages.classes` ...

Extending example to use FQN

Model

```
package BuildHouse {
        task feasibility DONE {
             next buyLand
        }
        task buyLand DONE {
             searchAds, findLand, buyLand
             next makePlan
        }
        task makePlan DOING {
             chooseArchitect, giveInstructions, choosePlan
             next buildHouse
        }
        task buildHouse TODO {
             buildHouse
             next BuildFence.feasibility
        }
        task moveIn {}
}
package BuildFence {
        task feasibility TODO {}
        task buildFence {
             chooseCompany, giveInstructions, buildFence
             next BuildHouse.moveIn
        }
}
```

Meta-model

```
Model: elements+=Element;
Package: 'package' name=ID '{'
            elements+=Element
         '}';
Element: Package | Task;
Task: 'task' name=ID (state=State)? '{'
            steps*=Step[',']
            ('next' next+=[Task|FQN|^elements*.elements][','])?
      '}';
State: 'TODO' | 'DOING' | 'DONE';
Step: !'next' ID;
FQN: ID+['.'];
Comment: /\/\/.*/;
```

Project scaffolding

```
$ textx startroject <folder>
```

Command asks a few questions and generates the project files. To install the generated project in developers mode (editable):

```
$ pip install -e <folder>
```

1. Answers are cached and will be reused as the default answers in further runs.

After installation the language (or generator) is visible to the `textx list-languages` or `textx list-generators` commands.

`startproject` command is not defined in the base `textX` library but in `textX-dev` package. Thus to have it registered we must install this project:

```
pip install textX-dev
```

Alternatively, we can install all `dev` dependencies:

```
pip install textX[dev]
```

- Questionnaire DSL
- textx-dev startproject questions model

textX-LS

- textX-LS is a language server that provides smartness for all domain specific languages based on textX.
- Consists of three parts: core, server and VS Code client.
- Uses two textX generators:
  - textx-gen-vscode - generates VS Code extension for registered textX language.
  - textx-gen-coloring - generates TextMate compatible syntax highlighting (e.g. VSCode).

A more complex example - PyFlies

- PyFlies is Domain-Specific Language (DSL) for designing experiments in psychology

Where to go next?

Where to go next?

- textX docs and tutorials

Where to go next?

- textX docs and tutorials
- Alessio Stalla, Quick Domain-Specific Languages in Python with textX

Where to go next?

- textX docs and tutorials
- Alessio Stalla, Quick Domain-Specific Languages in Python with textX
- textX discussions and the issue tracker