УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Проф. др Игор Дејановић

textX

скрипта за предмет Језици специфични за домен

Нови Сад, 2025

Садржај

Глава 1

Основни подаци

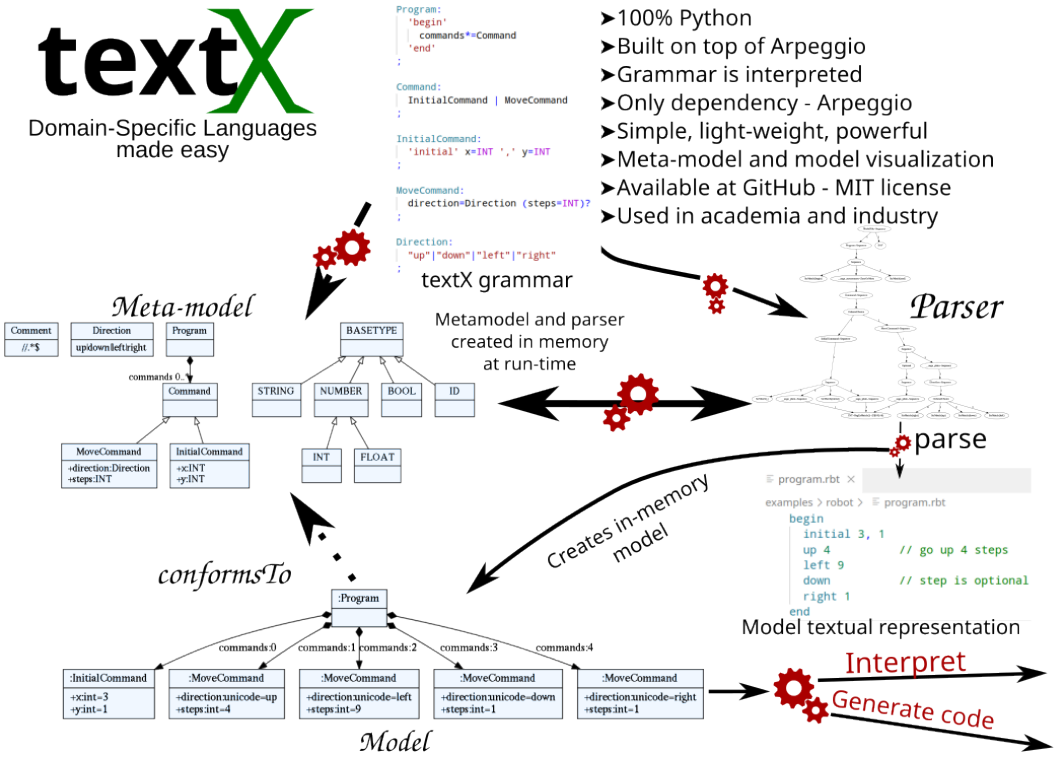
- 100% Пајтон код
- МИТ лиценца - користи се и у комерцијалним решењима и на истраживачким пројектима
- Најпопуларнија библиотека за израду ЈСД-ова у Пајтону
  - GitHub stars - 800+
  - GitHub forks - 80
  - GitHub contributors - 24
  - GitHub used by - 584
- Екосистем под GitHub организацијом https://github.com/textX/
- Документација, примери и туторијали доступни на https://textx.github.io/textX/
- Интеграција са VS Code за све језике базиране на textX-у кроз textX-LS пројекат[1]
- Онлајн игралиште за испробавање https://textx.github.io/textx-playground/

Напомена: ова скрипта прати слајдове које можете наћи на линку https://igordejanovic.net/courses/tech/textX/

[1] https://github.com/textX/textX-LS

Глава 2

Архитектура



Глава 3

Инсталација

3.1 Инсталација

- Са PyPI

```
~> mkdir jsd
~> cd jsd
~/jsd> uv init
Initialized project `jsd`
?main ~/jsd> ls
main.py  pyproject.toml  README.md
?main ~/jsd> uv add textx[cli]
Using CPython 3.14.0
Creating virtual environment at: .venv
Resolved 5 packages in 8ms
Installed 3 packages in 15ms
 + arpeggio==2.0.3
 + click==8.3.1
 + textx==4.2.3
```

```
?main ~/jsd> uv run textx
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.
```

3.2 Инсталација развојне верзије са GitHub-а

```
~> mkdir jsd
~> cd jsd
~/jsd> uv init
Initialized project `jsd`
?main ~/jsd> uv add git+https://github.com/textx/textx@master
Using CPython 3.14.0
Creating virtual environment at: .venv
Resolved 3 packages in 210ms
    Updated https://github.com/textx/textx
(6ef5164d77988a8b905a98748e90586c44efba9e)
      Built textx @ git+https://github.com/textx/textx@6ef5164d77988a8b905a
98748e90586c44efba9e
Prepared 1 package in 2.20s
Installed 2 packages in 2ms
 + arpeggio==2.0.3
 + textx==4.3.0.dev0 (from git+https://github.com/textx/textx@6ef5164d77988
a8b905a98748e90586c44efba9e)
```

3.3 Инсталација за развој

```
  ~> mkdir jsd
  ~> cd jsd
  ~/jsd> git clone git@github.com:textX/textX
  Cloning into 'textX'...
  remote: Enumerating objects: 14302, done.
  remote: Counting objects: 100% (2641/2641), done.
  remote: Compressing objects: 100% (975/975), done.
  remote: Total 14302 (delta 1463), reused 2486 (delta 1365), pack-reused
  11661 (from 1)
  Receiving objects: 100% (14302/14302), 17.28 MiB | 12.45 MiB/s, done.
  Resolving deltas: 100% (8808/8808), done.
  ~/jsd> cd textX
  master ~/jsd/textX> make dev
  rm -fr build/
  rm -fr dist/
  rm -fr .eggs/
  ...
   + urwid-readline==0.15.1
   + wcwidth==0.2.14
   + webencodings==0.5.1
```

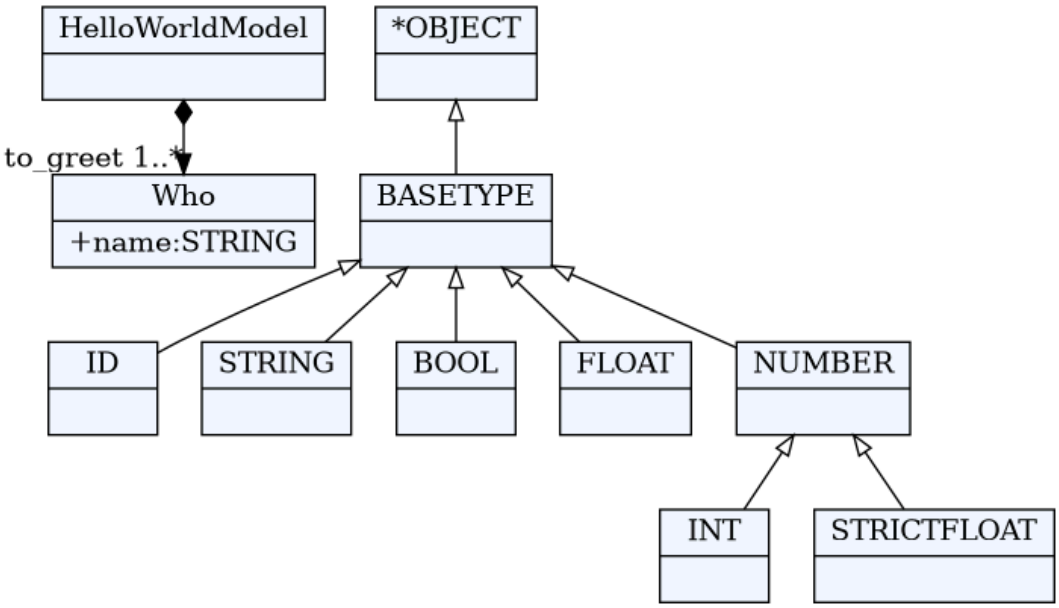
Глава 4

Основна употреба

4.1 Граматика = мета-модел + конкретна синтакса

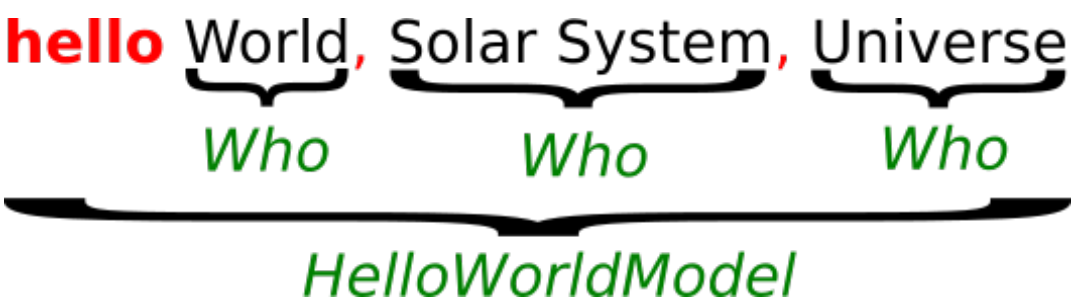
```
HelloWorldModel:
  'hello' to_greet+=Who[',']
;

Who:
  name = /[^,]*/
;
```

```
from textx import metamodel_from_file
hello_meta = metamodel_from_file('hello.tx')
```



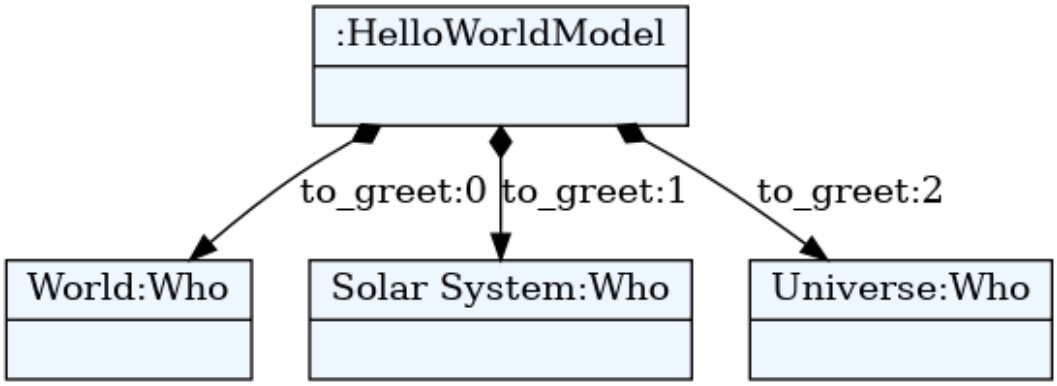
4.2 Модел = програм

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



4.3 Парсирање - инстанцирање модела

```
example_hello_model = hello_meta.model_from_file('example.hello')
```



- Модел је граф Пајтон објеката чија структура је у складу са граматиком (нпр. `HelloWorldModel` објекат садржи Пајтон листу `to_greet`).
- Модел можемо даље интерпретирати, анализирати, генерисати код…

4.4 Провера и визуализација мета-модела

- textX ће при парсирају граматике пријавити синтаксне грешке.
- Ако желимо можемо проверити граматику у току развоја:

```
textx check hello.tx
```

```
/home/igor/repos/igordejanovic.github.io/courses/tech/textX/hello.tx: OK.
```

- У случају грешке биће пријављена тачна локација.
- или визуализовати

```
textx list-generators
```

```
any -> dot                textX      Generating dot visualizations from
arbitrary models
textX -> dot       textX   Generating dot visualizations from textX grammars
```

```
textX -> PlantUML     textX     Generating PlantUML visualizations from
textX grammars
```

```
textx generate hello.tx --target dot
```

```
Generating dot target from models:
/home/igor/repos/igordejanovic.github.io/courses/tech/textX/hello.tx
-> /home/igor/repos/igordejanovic.github.io/courses/tech/textX/hello.dot
   To convert to png run "dot -Tpng -O hello.dot"
```

4.5 Робот пример

fajl `robot.tx`

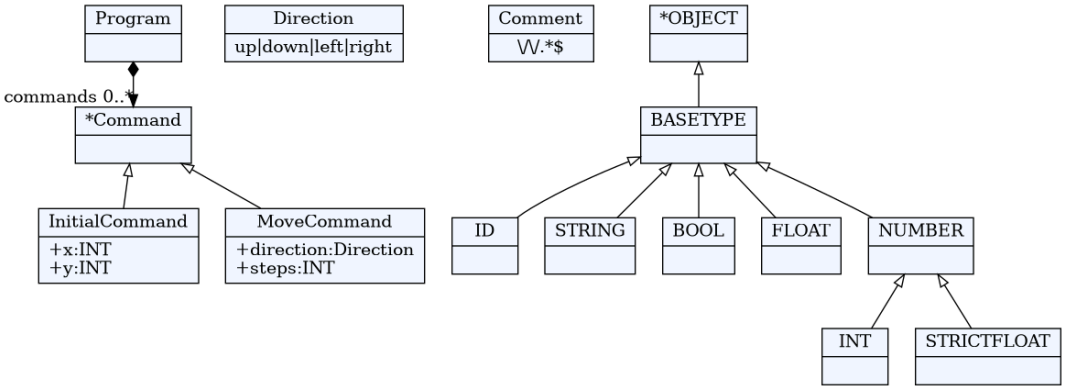
```
Program:
  'begin'
    commands*=Command
  'end'
;

Command:
  InitialCommand | MoveCommand
;

InitialCommand:
  'initial' x=INT ',' y=INT
;

MoveCommand:
  direction=Direction (steps=INT)?
;

Direction:
  "up"|"down"|"left"|"right"
;

Comment:
  /\/\/.*$/
;
```

fajl `program.rbt`

```
begin
    initial 3, 1
    up 4
    left 9
    down
    right 1
end
```

4.6 Инстанцирање мета-модела

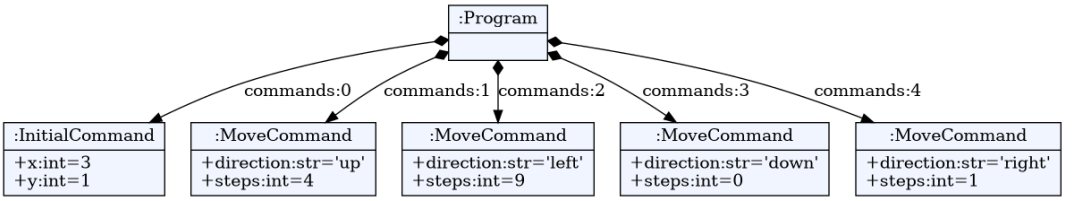
```
from textx import metamodel_from_file
robot_mm = metamodel_from_file('robot.tx')
```




```
textx generate robot.tx --target dot
dot -Tpng -O robot.dot
```

4.7 Парсирање и инстанцирање модела

```
robot_model = robot_mm.model_from_file('program.rbt')
```




```
textx generate program.rbt --grammar robot.tx --target dot
dot -Tpng -O program.dot
```

4.8 Шта радити са моделом?

- Интерпретирање
- Генерисање кода
- Разне врсте анализе и трансформације

4.9 Интерпретирање Робот модела

```
class Robot(object):

    def __init__(self):
        # Initial position is (0,0)
        self.x = 0
        self.y = 0

    def __str__(self):
        return "Robot position is {}, {}.".format(self.x, self.y)
```

4.10 Интерпретирање Робот модела

```
def interpret(self, model):

    # model is an instance of Program
    for c in model.commands:

        if c.__class__.__name__ == "InitialCommand":
            print("Setting position to: {}, {}".format(c.x, c.y))
            self.x = c.x
            self.y = c.y
        else:
            dir = c.direction
            print("Going {} for {} step(s).".format(dir, c.steps))

            move = {
                "up": (0, 1),
                "down": (0, -1),
                "left": (-1, 0),
                "right": (1, 0)
            }[dir]

            # Calculate new robot position
            self.x += c.steps * move[0]
            self.y += c.steps * move[1]
```

4.11 Интерпретација Робот модела

```
robot = Robot()
    robot.interpret(robot_model)
```

Проблем: Ако не задамо корак подразумевано је 0 (textX дефинише *default* вредности за базичне типове).

4.12 Object processor

```
def move_command_processor(move_cmd):
    """
    This is object processor for MoveCommand instances.
    It implements a default step of 1 in case not given
    in the program.
    """

    if move_cmd.steps == 0:
        move_cmd.steps = 1
```

```
MoveCommand:
  direction=Direction (steps=INT)?
;
```

Регистрација процесора на мета-моделу:

```
# Register object processor for MoveCommand
        robot_mm.register_obj_processors({'MoveCommand':
move_command_processor})
```

Сада се робот понаша исправно.

Глава 5

textX језик

5.1 textX граматичка правила

textX метајезик, односно граматика, се састоји од скупа правила.

Свако правило има јединствено име дефинисано на почетку пре двотачке, и тело правила које описује образац који мора бити препознат од стране парсера. Правило се завршава карактером `;`. Поред обрасца за парсирање, правила у исто време дефинишу концепте циљног језика тј. његове апстрактне синтаксе, односно метамодела. Ови концепти ће у време извршавања бити доступни као Пајтон класе и биће коришћени за инстанцирање објеката које парсер препозна у улазном текстуалном фајлу модела/програма.

На пример, ако развијамо језик за опис цртежа, концепти овог језика би могли бити `Shape`, `Line`, `Circle` итд. Следеће правило се користи да опише концепт `Circle`:

```
Circle:
  'Circle' '(' color=ID ',' size=INT ')'
;
```

5.2 Врсте правила

Постоје три врсте правила у textX-у:

- Обична правила (*Common rules*),
- Апстрактна правила (*Abstract rules*), i
- Правила препознавања (*Match rules*).

5.3 Обична правила

Обична правила су правила која садрже бар један израз доделе (видети 5), односно имају дефинисане атрибуте. Ова врста правила ће за последицу имати динамичко креирање Пајтон класа које ће бити инстанциране за време парсирања улазног стринга.

На пример:

```
InitialCommand:
  'initial' x=INT ',' y=INT
;
```

Правило `InitialCommand` ће довести до креирања Пајтон класе истог имена чије instance ће imati dva atributa: `x` и `y`.

5.4 Апстрактна правила

Апстрактна правила су правила која немају изразе доделе и референцирају бар једно апстрактно или обично правило. Најчешће су дефинисана као уређени избор других правила јер се користе да генерализују друга правила. На пример:

```
Program:
  'begin'
    commands*=Command
  'end'
;

Command:
  MoveCommand | InitialCommand
;
```

У овом примеру, Пајтон објекат у листи `commands` ће бити или `MoveCommand` или `InitialCommand`. `Command` правило је апстрактно. Ово правило никада неће резултовати Пајтон објектом.

Апстрактно правило може да се користи и у референцама повезивања (видети 5). На пример:

```
ListOfCommands:
  commands*=[Command][',']
;
```

Такође, апстрактна правила могу референцирати правила препознавања и базичне типове. На пример:

```
Value:
    STRING | FLOAT | BOOL | Object
    | Array | "null"
;
```

У овом примеру, базични типови као и препознавање стринг `"null"` су правила препознавања, али `Object` и `Array` су обична правила и стога је `Value` правило апстрактно.

Апстрактна правила могу бити сложена секвенца или уређени избор референци и правила препознавања док год имамо бар једну референцу на апстрактно или обично правило. На пример:

```
Value:
  'id' /\d+-\d+/ | FLOAT | Object
;
```

Правило чије тело се састоји само од једне референце препознавања на друго апстрактно или обично правило је такође апстрактно правило:

```
Value:
    OtherRule
;
```

Уколико је `OtherRule` апстрактно или обично правило тада је и `Value` апстрактно правило.

5.5 Правила препознавања

Правила препознавања су правила које немају директних или индиректних израза доделе, односно сва референцирана правила су такође правила препознавања. Обично се користе да дефинишу набројиве вредности или сложена препознавања стринга која се не могу исказати обични регуларним изразом.

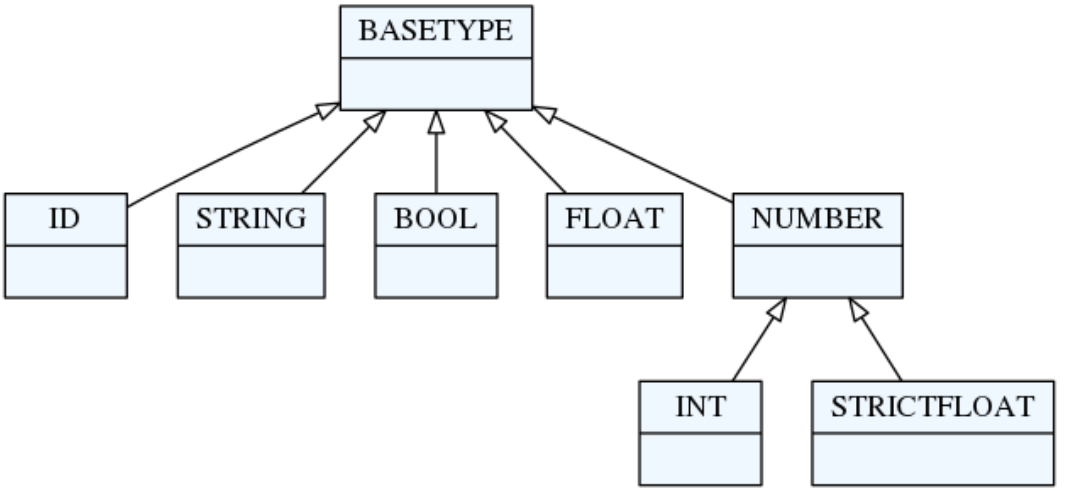
На пример:

```
Widget:
  "edit"|"combo"|"checkbox"|"togglebutton"
;

Name:
  STRING|/(\w|\+|-)+/
;
```

Ова правила могу да се користе искључиво у референцирању преко препознавања, односно не могу се користити у референцама везе јер не дефинишу “праве” објекте. Њихов тип у време извршавања је увек основни Пајтон тип (`str`, `int`, `bool`, `float`).

Сва базична, имплицитна, textX правила (нпр. `INT`, `STRING`, `BASETYPE`) су правила препознавања.

5.6 Базична правила/типови



- `ID` --- препознаје `[^\d\W]\w*\b`. Конвертује препознати низ карактера у Пајтон `str` тип.
- `INT` --- препознаје целе бројеве `[-+]?[0-9]+`. Конвертује препознати низ карактера у Пајтон `int` тип.
- `FLOAT` --- препознаје реалне бројеве. Конвертује препознати низ карактера у Пајтон `float` тип.
- `BOOL` --- препознаје булову вредност (`0/1`, `true/false`). Препознати низ карактера се конвертује у Пајтон `bool` тип.
- `STRING` --- препознаје стринг под једноструким или двоструким знацима навода. Знаци навода се могу наћи унутар стринга, али уколико су истог типа као и наводи који се користе да ограниче стринг мора се користити префикс `'\'` (енг. *backslash escaping*).

Препознати уграђени типови се аутоматски конвертују у одговарајуће Пајтон типове и постављају на подразумевану вредност у оквиру опционих препознавања.

5.7 Препознавања (*Matches*)

Поред уграђених базичних правила, правила препознавања су правила најнижег нивоа. Представљају основне градивне јединице сложенијих правила. Ова правила ће конзумирати део улаза уколико је препознавање успешно.

Постоје две врсте препознавања: *препознавање стринга* и *препознавање регуларног израза*.

5.8 Препознавање стринга

Препознавање стринга се пише као стринг у једноструким или двоструким знацима навода. Овако написано правило препознаће задати стринг са улаза у облику у ком је задат.

Примери:

```
'blue'
'zero'
'person'
```

5.9 Препознавање регуларног израза

Препознавање регуларног израза користи Путхон регуларне изразе[1] који се наводе унутар `/ /`. Дакле, дефинишу класу стрингова који се могу наћи на улазу.

Примери:

- `/\d+/` --- препознаје стринг од једне или више цифри.
- `/\d{3,4}-\d{3}/` --- 3 или 4 цифре, затим ‘-’ па затим још 3 цифре.
- `/[^}]*/` --- нула или више карактера различитих од ‘}’.

1. За увод у регуларне изразе на програмском језику Пајтон видети https://docs.python.org/3/howto/regex.html

5.10 Секвенца (*Sequence*)

Секвенца је најједноставнији сложени израз који се добија навођењем подизраза један иза другог. На пример, следеће правило:

```
Colors:
  "red" "green" "blue"
;
```

је дефинисано као секвенца која се састоји од три стринг препознавања (`red`, `greeen` и `blue`). Секвенца ће успешно бити препозната ако су препознати сви њени подизрази у редоследу у ком су наведени. Претходно `Colors` правило ће препознати следећи стринг:

```
red green   blue
```

Уколико је укључено аутоматско прескакање празних карактера (*whitespace skip*), што је подразумевано, тада се може између два подизраза у секвенци наћи произвољан број празних карактера као што је приказано у претходном примеру.

5.11 Уређени избор (*Ordered choice*)

Уређени избор се наводи као скуп израза раздвојених знаком `|`. Овај израз ће покушати да препозна подизразе с лева на десно. Први израз који се успешно препозна биће коришћен као резултат препознавања.

На пример, следеће правило

```
Color:
  "red" | "green" | "blue"
;
```

ће препознати или реч `red` или `green` или `blue` при чему ће парсер да покуша сваки од подизраза с лева на десно.

Ово је у супротности са класичним парсерима где је оператор избора неуређен. textX као технологију за парсирање користи Arpeggio парсер који је базиран на PEG формализму где је оператор избора уређен. Због тога парсирање не може бити вишезначно, односно ако се улаз исправно парсира може постојати само једно стабло парсирања.

5.12 Опционо препознавање (*Optional*)

Опционо препознавање је израз који ће да покуша да препозна свој подизраз ако може, али ће успети у препознавању и уколико подизраз није препознат.

На пример, уколико имамо правило

```
MoveUp:
  'up' INT?
;
```

`INT` ће бити опционо (јер је наведен знак `?`) па ће бити могуће навести број иза речи `up`, али неће бити обавезно.

Следеће линије ће бити успешно препознате претходним правилом:

```
up 45
up 1
up
```

Опциони изрази могу бити и сложенији. На пример, код правила

```
MoveUp:
  'up' ( INT | FLOAT )?
;
```

имамо да је уређени избор опциони тако да ћемо у овом случају моћи навести цео или реалан број (`INT` или `FLOAT`), али нисмо обавезни да то учинимо.

5.13 Понављања (*Repetitions*)

Понављање нула или више пута (*zero or more*) се наводи употребом оператора * иза подизраза. Подизраз ће у том случају бити препознат нула или више пута.

На пример, уколико имамо правило

```
Colors:
  ("red" | "green" | "blue")*
;
```

понављање је примењено на уређени избор унутар заграде. Стога, парсер ће да покуша да препозна елементе уређеног избора с лева на десно и када обави успешно препознавање понављаће га док год успева да препозна једну од задатих речи.

Следећи улаз ће бити успешно парсиран:

```
red blue green
```

али такође и

```
red blue blue red red blue green
```

или празан стринг (препознавање нула пута).

5.14 Понављања (*Repetitions*)

Понављање једном или више пута се наводи употребом оператора + иза подизраза. Подизраз ће у том случају бити препознат један или више пута.

На пример, уколико имамо правило које је слично претходном, али користи овај облик понављања

```
Colors:
  ("red" | "green" | "blue")+
;
```

обавезно је навођење бар једне боје, али може се навести и више у произвољном редоследу и са понављањем као и код претходног правила. Дакле, ово правило не препознаје празан стринг.

5.15 Доделе (*Assignments*)

Доделе се користе као део поступка за дедукцију метамодела. Свака додела резултује креирањем атрибута мета-класе креиране textX правилом.

Свако правило има своју леву страну (LHS) и десну страну (RHS). LHS је увек име атрибута који ће бити креиран док је десна страна правило које одређује шта се препознаје на датом месту као и тип објекта који ће бити инстанциран и додељен атрибуту

при парсирању и инстанцирању модела. PXC је увек референца на друго правило или једноставно препознавање.

На пример:

```
Person:
  name=Name ',' surname=Surname ','
    age=INT ',' height=INT ';'
;
```

Напомена: Правило `Name` и `Surname` су дефинисани у граматици, али нису дати у овом примеру.

5.16 Доделе - креирање атрибута

Претходни пример описује правило и мета-класу `Person` које ће парсирати и инстанцирати објекат са четири атрибута.

- `name` --- где ће се правилом `Name` препознати објекат мета-класе `Name` са улаза, инстанцирати и доделити атрибуту,
- `surname` --- исто као и за `name`, али се користи правило `Surname` и додељује атрибуту `surname`,
- `age` --- користиће се уграђено базично правило `INT` и број који се препозна конвертоваће се у Пајтон `int` тип и доделити атрибуту,
- `height` --- исто као и за атрибут `age`, али ће се препознати број доделити `height` атрибуту `Person` инстанце.

Пример:

```
Petar, Petrović, 25, 185;
```

Зарези, дати у претходном примеру, који ће бити препознати између препознавања правила доделе, као и тачка-зарез на крају, морају се наћи у улазном стрингу, али ће бити одбачени приликом креирања модела јер немају никакво семантичко значење. Кажемо да представљају *синтаксни шум*.

5.17 Доделе - конверзије и вредности

Увек када је на RHS неки од базичних типова (нпр. `INT`, `BOOL`, `FLOAT`, `ID`) доћи ће до конверзије препознатог стринга у одговарајући Пајтон тип (нпр. `int`, `bool`, `float`, `str`).

Ако је на RHS препознавање стринга или регуларног израза као у следећем примеру:

```
Color:
  color=/\w+/
;
```

тада ће атрибут на LHS (`color`) бити постављен на вредност коју препозна RHS правило.

Уколико се на RHS налази референца на друго правило тада ће бити препознат и инстанциран објекат класе датог правила и атрибут на LHS ће бити референца на дату инстанцу. На пример:

```
Project:
  'project' name=ID 'lead' lead=Person
;
```

`lead` атрибут биће референца на објекат класе `Person` а правило `Person` мора успешно препознати овај објекат иза кључне речи `lead`.

Постоје четири врсте доделе: *obična*, *bulova*, *nula ili više* и *jedan ili više*.

5.18 Обична додела

Обична додела (=) ће обавити препознавање RHS једном и објекат који се креира доделити атрибуту на LHS.

```
a=INT
b=FLOAT
c=/[a-Z0-9]+/
dir=Direction
```

5.19 Булова додела

Булова додела (?=) ће поставити LHS атрибут на `True` уколико је RHS препознат на улазу или на `False` уколико није.

```
cold ?= 'cold'
number_given ?= INT
```

5.20 Додела нула/један или више

Додела нула или више (*=) ће препознавати RHS све док успева и све објекте редом сместити у Пајтон листу која је на LHS. Ако препознавање не успе ни једном LHS ће бити празна листа.

```
commands*=Command  // опциони низ команди
numbers*=INT       // опциони низ целих бројева
```

Додела један или више (+=) ради исто као претходна с тим што RHS мора да препозна бар један објекат тј. LHS никада неће бити празна листа.

```
numbers+=INT       // низ целих бројева, мора постојати бар један
```

5.21 Референце (*References*)

Правила могу да се међусобно референцирају. Референце се наводе на RHS. Постоје два начина референцирања правила: *референцирање преко препознавања* и *референцирање преко везе*.

5.22 Референцирање преко препознавања

Референцирање преко препознавања се дефинише простим навођењем имена неког другог правила. Можемо такође рећи и да правило које врши референцирање *позива* правило које се референцира. На месту референцирања textX ће покушати да препозна циљно правило у целости и, уколико га препозна, инстанцираће га и доделити атрибуту на левој страни. Референцирање преко препознавања има семантику садржавања. Објекат који референцира *садржи* објекат који се референцира. Као додатну помоћ textX ће аутоматски креирати Пајтон референцу `парент` на објекту који се референцира. Овај атрибут ће референцирати на објекат који референцира.

```
Structure:
  'structure' '{'
    elements*=StructureElement
  '}'
;
```

У претходном примеру правило `Structure` референцира преко препознавања правило `StructureElement`. Унутар тела `Structure` концепта, биће препознато нула или више инстанци `StructureElement` класе. Инстанце ће бити додељене `elements` атрибуту који ће, у овом случају, бити типа Пајтон листе.

5.23 Референцирање преко везе

Референцирање преко везе наводи име циљног правила унутар угластих заграда. На овом месту, парсер ће покушати да препозна име циљног објекта а не објекат у целости. Циљни објекат мора да буде дефинисан негде друго унутар модела. Уколико је објекат са датим именом пронађен, textX ће аутоматски да атрибуту на левој страни додели вредност референце на циљни објекат.

```
ScreenType:
  'screen' name=ID "{"
  '}'
;

ScreenInstance:
  'screen' type=[ScreenType]
;
```

`type` атрибут који припада правилу `ScreenInstance` референцира преко везе правило `ScreenType`.

Ово би био пример правилне употребе:

```
// Ово је дефиниција ScreenType објекта
// који се зове Introduction
screen Introduction {

}

// А ово је инстанца ScreenInstance која
// референцира претходни Introduction ScreenType.
screen Introduction
```

Иза кључне речи `screen` на последњој линији неће бити препознат цео `ScreenType`, као што би то био случај са референцирањем преко препознавања, већ ће бити препознато име (у овом случају `Introduction`) `ScreenType` објекта и аутоматски ће веза бити разрешена у референцу на дати објекат која ће бити додељена `type` атрибуту `ScreenInstance` инстанце.

Подразумевано се користи `ID` правило за препознавање назива циљног објекта. Уколико желимо то да променимо можемо урадити следеће:

```
ScreenInstance:
  'screen' type=[ScreenType|WORD]
;
```

У претходном примеру ће за препознавање имена циљног објекта бити коришћено правило `WORD`.

5.24 Синтаксни предикати (*Syntactic predicates*)

Синтаксни предикати су оператори који се користе за имплементацију погледа унапред (енг. *lookaheads*). Поглед унапред је техника која омогућава да се донесе одлука о примени правила за парсирање на основу дела улазног стринга који следи без да се тај део стринга обради тј. конзумира. Ови оператори се наводе као префикс за неко textX правило. Синтаксни предикат заједно са правилом које следи чини ново правило које успева или не успева без конзумирања улаза. Најчешће ће овакво правило бити део секвенце чиме ће се омогућити одустајање од даље анализе секвенце уколико предикат није успео.

Постоје две врсте синтаксних предиката у textX-у: `Not` и `And`.

5.25 Негативни поглед унапред

`Not` --- негативни поглед унапред (`!`) --- Успева уколико правило које следи иза `!` предиката не препознаје наставак улазног стринга и обрнуто.

Пример проблема:

```
Expression: Let | ID | NUMBER;
Let:
    'let'
        expr+=Expression
    'end'
;
```

У претходном примеру имамо рекурзивно правило `Let` које се индиректно референцира преко правила `Expression`. Проблем је у томе што ће `ID` правило које се позива из `Expression` правила препознати кључну реч `end` што ће довести до тога да ниједно `Let` правило неће моћи успешно да се заврши.

Да бисмо решили овај проблем модификујемо граматику на следећи начин:

```
Expression: Let | MyID | NUMBER;
Let:
    'let'
        expr+=Expression
    'end'
;
Keyword: 'let' | 'end';
MyID: !Keyword ID;
```

Уместо директне употребе уграђеног `ID` правила уводимо правило `MyID` које користи `Not` синтаксни предикат да спречи препознавање кључних речи `let` и `end` као изразе `Expression` правила. На овај начин ће `end` бити конзумирано као завршетак `Let` правила и граматика ће исправно функционисати.

5.26 Позитивни поглед унапред

`And` --- позитивни поглед унапред (&) --- Успева уколико правило које следи иза `&` предиката препознаје наставак улазног стринга и обрнуто.

Пример:

```
Model:
    elements+=Element
;
Element:
    AbeforeB | A | B
;
AbeforeB:
    a='a' &'b'      // правило успева само
                    // ако 'b' следи после 'a'
;
A: a='a';
B: a='b';
```

Уколико имамо улазни стринг `"a  a  a  b"`, прва два `a` токена ће бити препознати правилом `A` док ће трећи токен `a` бити препознат правилом `AbeforeB`. Иако се увек проверава прво `AbeforeB`, правило неће успети за прва два `a` токена јер иза не следи токен `b`. Последњи токен ће бити препознат правилом `B` јер га претходно успешно правило `AbeforeB` није конзумирало са улаза.

5.27 Уклањање препознатог улаза (*Match Suppression*)

Некада је потребно дефинисати правило препознавања које ће вратити само део препознатог улаза. У овом случају можемо користити оператор за уклањање препознатог улаза (`-`) који се наводи после израза препознавања.

На пример:

```
FullyQualifiedID[noskipws]:
    /\s*/-
    QuotedID+['.']
    /\s*/-
;
QuotedID:
    '"'?- ID '"'?-
;
```

У претходном примеру желимо да препознамо потпуно квалификоване идентификаторе (*Fully Qualified IDs*) где делови имена могу бити под знацима навода. На пример --- `first."second".third`. Такође, желимо да уклонимо знаке навода из имена. Један начин би био да радимо постпроцесирање после завршеног парсирања, али је за ту намену елегантније решење употреба оператора `-`.

Правило `FullyQualifiedID` користи `noskipws` модификатор да би онемогућио појаву празних карактера унутар потпуно квалификованих имена. Због тога се морају на почетку и на крају препознати празни карактери и одбацити уколико постоје што се обавља правилом `/\s*/-`.

Правило `FullyQualifiedID` даље препознаје један или више `QuotedID` одвојених тачкама. Правило `QuotedID` препознаје `ID` које опционо може бити унутар знакова навода а затим препознате знакове навода одбацује употребом `"-"` оператора.

5.28 Модификатори понављања (*Repetition modifiers*)

Користе се за модификацију понашања свих оператора понављања (*, +, *=, +=). Наводе се унутар угластих заграда иза оператора понављања. Може се навести више модификатора и у том случају се раздвајају зарезима.

У текућој имплементацији је дефинисано два модификатора понављања: модификатор сепарације и модификатор краја линије.

5.29 Модификатор сепарације

Модификатор сепарације (*Separator modifier*) се користи да дефинише сепаратор код вишеструког препознавања. Наводи се као једноставно препознавање (препознавање стринга или регуларног израза).

```
numbers*=INT[',']
```

У овом примеру имамо препознавање низа целобројних вредности. Као модификатор сепарације дефинисан је зарез па се очекује да између свака два броја која се препознају буде наведен зарез. На пример:

```
45, 47, 3, 78
```

Такође, можемо дефинисати као модификатор препознавање регуларног израза. На пример:

```
fields += ID[/;|,|:/]
```

У овом случају као модификатор сепарације наводи се регуларни израз који дефинише да ће низ поља бити раздвојено карактером који може бити тачка-зарез, зарез или двотачка. Тако ће успешно да се препозна следећи стринг:

```
first, second; third, fourth: fifth
```

5.30 Модификатор краја линије

Модификатор краја линије (*End-of-line terminate modifier*) се наводи као кључна реч `eolterm`. Уколико је укључен овај модификатор оператори понављања ће завршити понављање на крају текућег реда тј. радиће само за текући ред.

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

Код овог примера вршимо препознавање нула или више стрингова раздвојених зарезима, али само унутар текућег реда. Ако задамо следећи улаз:

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

правило ће препознати и конзумирати само прави ред. Стринг `"fourth"` неће бити обухваћен.

Треба обратити пажњу да употреба `eolterm` модификатора ступа на снагу одмах по завршетку претходног препознавања.

```
Conditions:
  'conditions' '{'
    varNames+=WORD[eolterm]
  '}'
```

У овом примеру један или више `WORD` препознавања мора бити обављено одмах иза `conditions {`, на истој линији. То није била наша намера јер не желимо да ограничимо корисника већ да му допустимо да пређе у следећи ред и, уколико жели, направи произвољан број празних редова. Да би ово омогућили морамо да препознамо и одбацимо све празне карактере пре почетка првог успешног `WORD` препознавања. То радимо на следећи начин:

```
Conditions:
  'conditions' '{'
    /\s*/
    varNames+=WORD[eolterm]
  '}'
```

Искористили смо препознавање регуларног израза `/\s*/` да прескочимо све празне карактере, укључујући и крајеве линија, све до првог успешног препознавања `WORD` правила.

Глава 6

Метамодели

У textX-u метамодел је објекат који садржи све релевантне податке о језику, све класе језика, као и парсер који ће бити у стању да чита програме/моделе на датом језику и креира објектни модел текстуалне репрезентације. Метамодели се креирају Пајтон функцијама `metamodel_from_file` и `metamodel_from_str` из пакета `textx.metamodel`. Ове функције примају textX граматику језика а враћају метамодел објекат уколико граматика нема грешака или одговарајући изузетак уколико грешка постоји.

Пример креирања метамодела из граматике дефинисане у фајлу:

```
from textx.metamodel import metamodel_from_file
my_metamodel = metamodel_from_file('my_grammar.tx')
```

Парсирање текстуалне репрезентације модела и креирање меморијске објектне репрезентације се обавља позивом метода `model_from_file` і `model_from_str` метамодел објекта.

```
my_model = my_metamodel.model_from_file('some_input.md')
```

6.1 Корисничке класе

За свако обично textX правило динамички се креира Пајтон класа истог назива у метамоделу. Ове класе ће бити инстанциране за време парсирања када се успешно препозна запис објекта у улазном стрингу. Инстанце ће чинити део објектног графа односно модела.

У већини случајева динамички креиране класе ће бити сасвим довољне, али постоје ситуације код којих ћемо желети да сами дефинишемо класу која ће бити инстанцирана при препознавању одређеног правила. Да би ово постигли користимо параметар `classes` при инстанцирању метамодела. Овај параметар представља листу корисничких класа чија имена морају бити иста као имена правила за које ће бити инстанциране.

```
from textx.metamodel import metamodel_from_str

grammar = '''
EntityModel:
  entities+=Entity
;

Entity:
  'entity' name=ID '{'
    attributes+=Attribute
  '}'
;

Attribute:
  name=ID ':' type=[Entity]
;
'''

class Entity:
  def __init__(self, parent, name, attributes):
    self.parent = parent
    self.name = name
    self.attributes = attributes


# Користимо "нашу" Entity класу.
# "Attribute" класа ће бити креирана динамички.
entity_mm = metamodel_from_str(grammar, classes=[Entity])
```

Метамодел `entity_mm` се може користити да инстанцира моделе где ће при инстанцирању `Entity` класе бити коришћена Путхон класа из претходног примера. Класа `Attribute` која је последица истоименог правила ће бити креирана динамички.

Корисничка класа треба да има конструктор који прима све атрибуте који су дефинисани граматичким правилом (у овом случају `name` и `attributes`). Уколико класа представља дете у вези родитељ-дете (видети наредну секцију), тада је као први параметар обавезан `parent` који представља везу према објекту родитељу.

6.2 Везе родитељ-дете

Често у моделима постоји инхерентна веза типа родитељ-дете. У претходном примеру свака инстанца `Attribute` класе припада некој инстанци `Entity` класе.

textX има аутоматску подршку за овај тип везе и динамички креира `parent` атрибут на свим објектима типа дете.

Обратити пажњу да је, код дефинисања корисничких класа, неопходно обезбедити овај параметар као први параметар конструктора код свих класа које представљају дете у вези родитељ-дете јер ће textX позивати конструктор уз прослеђивање референце на родитељски објекат приликом парсирања.

6.3 Процесори

За дефинисање додатне статичке семантике модела textX омогућава дефинисање процесора. Процесори су Пајтон објекти који се могу позвати (*callables*) и који могу да додатно провере и модификују препознати објекат у току парсирања. Користе се код правила која није могуће дефинисати граматиком.

Постоје две врсте процесора: процесори објеката и процесори модела.

Процесори објеката (*object processors*) --- су процесори који се позивају после сваког успешног препознавања објекта. Као једини параметар добијају објекат који требају да провере/модификују.

Процесори модела (*Model processors*) --- су процесори који се позивају када се цео модел успешно парсира. Као параметар добијају метамодел и модел. Могу да обаве произвољну проверу и/или модификацију модела. Региструју се позивом методе `register_model_processor` над метамодел објектом.

```
from textx.metamodel import metamodel_from_file

# Процесор модела је функција који ће прихватити метамодел
# и модел као своје параметре.
def check_some_semantics(metamodel, model):
  ...
  ... Врши проверу модела и баца TextXSemanticError
  ... ако су семантичка правила нарушена.

my_metamodel = metamodel_from_file('mygrammar.tx')

# Региструјемо модел процесор на инстанци метамодела
my_metamodel.register_model_processor(check_some_semantics)

# Парсирамо модел. Функција check_some_semantics ће бити
# аутоматски позвана након успешног парсирања да обави
# додатну семантичку проверу модела.
my_metamodel.model_from_file('some_model.ext')
```

6.4 Уграђени објекти

Често је потребно да у сваком моделу имамо објекте који су увек присутни, без потребе да их корисник дефинише, и које можемо референцирати из остатка модела. Да би олакшали посао кориснику можемо на нивоу метамодела регистровати уграђене

објекте. Тако регистровани објекти биће имплицитно део сваког модела. Класичан пример су примитивни типови (нпр. `integer`, `string`, `float`). Ако желимо да ово буду прави објекти а не само кључне речи, креираћемо нове објекте и регистровати над метамоделом. Наравно, да би могли инстанцирати класе у Пајтон-у потребно је да их региструјемо као корисничке класе (видети 6).

```
class Entity:
  def __init__(self, parent, name, attributes):
      self.parent = parent
      self.name = name
      self.attributes = attributes

entity_builtins = {
        'integer': Entity(None, 'integer', []),
        'string': Entity(None, 'string', [])
}
entity_mm = metamodel_from_file(
  'entity.tx',
  classes=[Entity],        # Регистровање Entity
                           # корисничке класе,
  builtins=entity_builtins # Регистровање integer и string
)                          # уграђених Entity објеката
```

У претходном примеру региструјемо корисничку класу `Entity` и затим две њене инстанце (`integer` и `string`) које ће представљати уграђене типове за атрибуте.

- За више детаља видети пример https://github.com/igordejanovic/textX/tree/master/examples/Entity

6.5 Аутоматска иницијализација атрибута

Вредности атрибута објеката препознатих у улазном стрингу биће подешени на вредности дате у тексту. Уколико је атрибут опциони и није наведен у улазном стрингу свакако ће бити креиран на објекту. Његова вредност биће подразумевана и зависиће од типа.

Подразумеване вредности за базичне textX типове су следеће:

- `ID` --- празан стринг --- `''`
- `INT` --- int --- `0`
- `FLOAT` --- float --- `0.0`
- `BOOL` --- bool --- `False`
- `STRING` --- празан стринг --- `''`

Сваки атрибут са мултиплицитетом нула или више (*=) а који не препознаје нити један елемент са улаза биће иницијализован на празну Пајтон листу. Уколико је мултиплицитет један или више (+=) то захтева препознавање бар једног објекта на улазу, па ће вредност атрибута бити Пајтон листа са свим препознатим објектима.

Иако овај механизам аутоматске иницијализације доста олакшава посао, постоје ситуације када може да засмета. На пример, проблем настаје уколико желимо да разликујемо ситуацију у којој корисник није дефинисао опциони елемент од ситуације у којој је дефинисао, али је ставио подразумевану вредност.

Механизам аутоматске иницијализације се може искључити постављањем параметра `auto_init_attributes` на `False` при позиву конструктора метамодела. У том случају, уколико вредност није наведена на улазу, биће постављена на `None`. То важи за све обичне доделе (=). Код опционих додела (?=) вредност ће бити `False` уколико није наведена. Код додела са мултиплицитетом вишим од 1 (*= и +=) атрибут ће увек бити иницијализован на Пајтон листу.

6.6 Конфигурација парсера

Arpeggio парсер креиран од стране textX-а се може конфигурисати са становишта осетљивости на величину слова (*case-sensitivity*), третирању празних карактера (*white-space handling*) и аутоматској детекцији кључних речи.

6.6.1 Конфигурација парсера - *case-sensitivity*

Подразумевано, парсер је осетљив на величину слова. Тако ће речи `Entity` и `entity` бити третиране као различите. Уколико је наш језик такав да величина слова није битна можемо при креирању метамодела проследити параметар `ignore_case` са вредношћу `True`.

```
from textx.metamodel import metamodel_from_file

my_metamodel = metamodel_from_file('mygrammar.tx',
                                   ignore_case=True)
```

6.6.2 Конфигурација парсера - *whitespaces*

Парсер подразумевано прескаче празне карактере (`spaces`, `tabs`). Због тога у граматикама није потребно експлицитно препознавати празне карактере. Постоје језици где су празни карактери сигнификантни. У таквим случајевима можемо искључити прескакање празних карактера параметром `skipws` који постављамо на `False`. Додатни механизам је редефинисање скупа празних карактера са параметром `ws` који представља стринг који се састоји од карактера из скупа празних карактера.

```
from textx.metamodel import metamodel_from_file
my_metamodel = metamodel_from_file('mygrammar.tx',
                                   skipws=False, ws='\s
')
```

Ова правила се могу дефинисати и на нивоу појединачних правила граматике.

6.6.3 Конфигурација парсера - *keywords*

При креирању ЈСД обично је пожељно да се кључне речи препознају у целини, односно да се не препознају уколико су делови других елемената језика (нпр. идентификатора). Због тога, textX може да буде конфигурисан да препознаје све што изгледа као идентификатор или кључна реч у целости. На пример, за `Entity` језик кључна реч `entity` не би смела да се препозна у стрингу `entity1`.

Могли бисмо да постигнемо одговарајући ефекат употребом регуларних израза и параметра за препознавање границе речи:

```
Enitity:
/\bentity\b/ name=ID ...
```

али би граматика у том случају била тешка за читање и одржавање. textX због тога уводи параметар метамодела `auto_kwd` који се може поставити на вредност `True` и производи исти ефекат.

```
from textx.metamodel import metamodel_from_file
my_metamodel = metamodel_from_file('mygrammar.tx',
                                   autokwd=True)
```

Глава 7

Модели

textX модели су објектни графови обичних Пајтон објеката (*Plain Old Python Object --- POPO*). Ови објекти су креирани из улазног стринга који по структури одговара дефинисаној textX граматици уз потенцијалну модификацију од стране процесора (видети 6).

На неки начин модел је сличан апстрактном синтаксном стаблу (AST) код класичног парсирања, али је на вишем семантичком нивоу јер су све референце разрешене и структура је облика графа.

Сваки објекат модела је инстанца класе креиране динамички на основу textX правила или дефинисане од стране корисника употребом механизма корисничких класа (видети 6).

Модел се креира позивом метода `model_from_file` и `model_from_str` метамодела.

```
from textx.metamodel import metamodel_from_file

my_mm = metamodel_from_file('mygrammar.tx')

# Kreiranje modela iz tekstualnog opisa
my_model = my_mm.model_from_file('some_model.ext')
```

У наставку текста, као пример, користимо `Entity` језик коришћен у секцији о корисничким класама (видети 6).

Садржај фајла `entity.tx` који дефинише језик (тј. метамодел) је следећи:

```
EntityModel:
  entities+=Entity
;

Entity:
  'entity' name=ID '{'
    attributes+=Attribute
  '}'
;

Attribute:
  name=ID ':' type=[Entity]
;
```

7.1 Специјални textX атрибути

Сваки Пајтон објекат textX модела који није базични Пајтон тип (нпр. `int`, `str`) поседује специјални атрибут `_tx_position` који представља апсолутну позицију објекта у текстуалном улазу. За конверзију позиције у ред и колону може се користити метода парсера `pos_to_linecol`.

На пример:

```
line, col = entity_mm.parser.pos_to_linecol(
      person_model.entities[0]._tx_position)
```

ће вратити линију и колону првог ентитета модела `person.ent`.

Поред позиције коренски објекат модела има и атрибуте:

- `_tx_filename` --- пуна путања и назив фајла из којег је модел учитан или `None` уколико је учитан из Пајтон стринга,
- `_tx_metamodel` --- референца на метамодел са којим је дати модел усклађен.

Глава 8

Визуализације

Метамодел, модел и стабло парсирања се могу трансформисати у *dot* запис у циљу визуализације. *dot* је текстуални ЈСД и алат за опис графова и њихову визуализацију и део је *GraphViz* пакета. `textx.export` модул садржи Пајтон функције `metamodel_export` и `model_export` за трансформацију метамодела, односно модела у *dot* запис.

Ако textX ради у моду за отклањање грешака (*debug*, видети 9), метамодел, модел и стабло парсирања ће бити аутоматски експортовани у *dot* запис.

dot фајлови се могу директно приказати у неком од доступних визуализатора (нпр. xdot[1], ZGRViewer[2]) или се, употребом *dot* алата, фајл може трансформисати у неки од битмапираних или векторских графичких формата (нпр. PNG, SVG, JPG, PDF).

1. xdot --- https://github.com/jrfonseca/xdot.py
2. ZGRViewer --- http://zvtm.sourceforge.net/zgrviewer.html

8.1 Визуализација метамодела

Метамодел се може визуализовати директно из програмског кода на следећи начин:

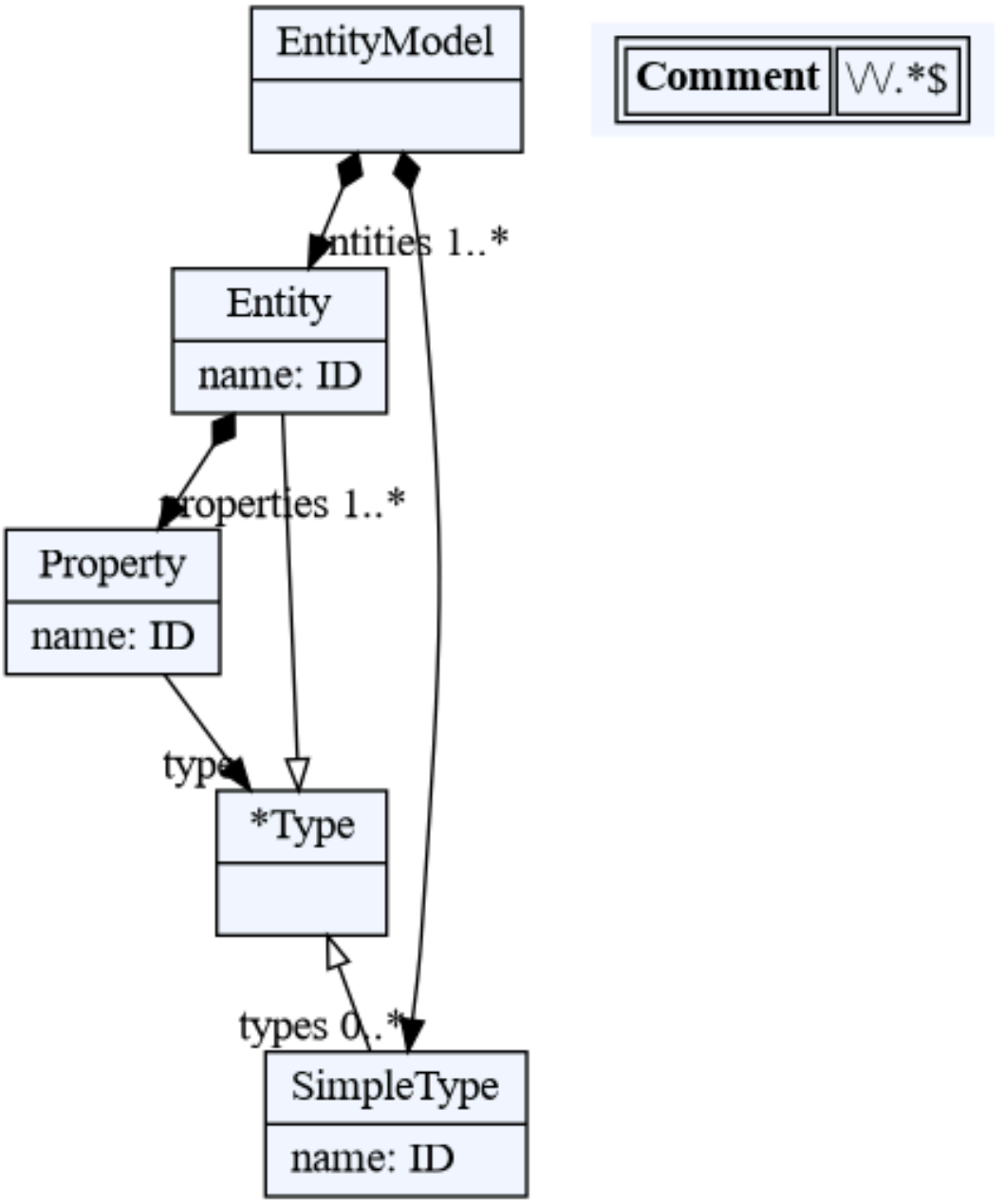
```
from textx.metamodel import metamodel_from_file
from textx.export import metamodel_export

entity_mm = metamodel_from_file('entity.tx')

metamodel_export(entity_mm, 'entity.dot')
```

Позив функције `metamodel_export` над метамоделом ће произвести *dot* фајл датог имена (у овом случају `entity.dot`).

Текстуални *dot* фајл можемо превести у неки од графичких формата на следећи начин:

```
$ dot -Tpng -O entity.dot
```

Ова команда ће произвести фајл `entity.dot.png` графичког битмапираног формата PNG.



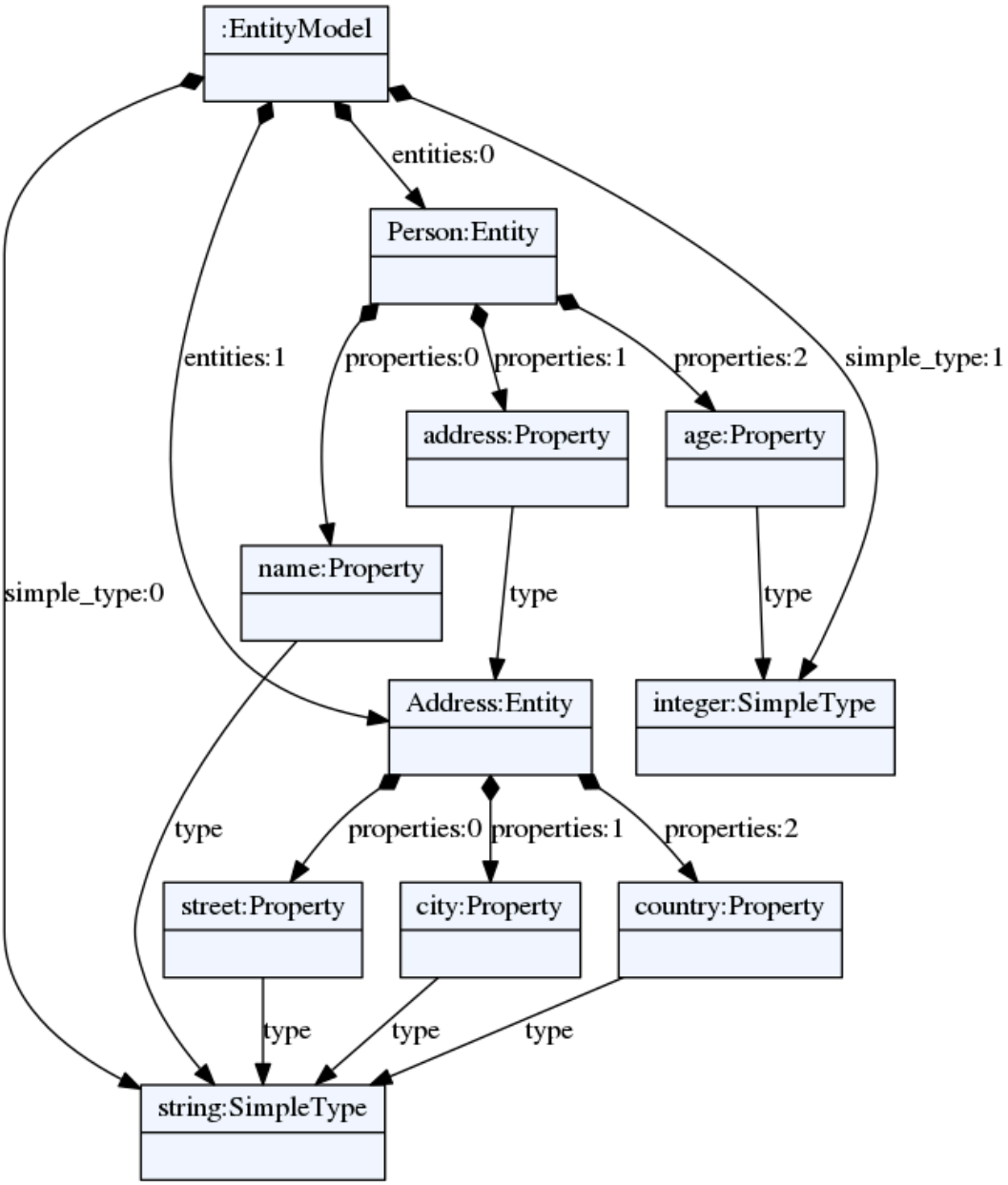
8.2 Визуализација модела

Такође, и модели се могу визуализовати из програмског кода. То се изводи на следећи начин:

```
from textx.export import model_export

person_model = entity_mm.model_from_file('person.ent')

model_export(person_model, 'person.dot')
```

Претходни кôд ће произвести фајл `person.dot` који се може превести у графички формат следећом командом:

```
$ dot -Tpng -O person.dot
```



Визуализација може да се обави и употребом `textx` komande описане у наставку.

Глава 9

Обрада грешака

Уколико textX детектује синтаксну грешку приликом парсирања описа метамодела (граматике језика) или описа модела, доћи ће до појаве изузетка `TextXSyntaxError` односно `TextXSemanticError`. Оба изузетка наслеђују `TextXError` и описани су у модулу `textx.exceptions`. Сви textX изузеци имају атрибут `message` који носи поруку о грешци и атрибуте `line` и `col` који представљају ред односно колону у којој се грешка десила.

textX подржава отклањање грешака (*debugging*) и на нивоу метамодела (граматике) и на нивоу модела. Подразумевано textX не ради у моду за отклањање грешака, али се може поставити у тај мод употребом `debug` параметра код инстанцирања метамодела односно модела.

```
from textx.metamodel import metamodel_from_file

robot_metamodel = metamodel_from_file('robot.tx', debug=True)
```

или

```
robot_program = robot_metamodel.model_from_file('program.rbt',
                                                debug=True)
```

Када textX ради у *debug* моду на конзоли ће бити приказане детаљне информације о свим акцијама које textX предузима приликом парсирања и анализе стабала парсирања. Такође, *dot* фајлови за стабла парсирања за метамоделе и моделе као и сами модели и метамодели, ће бити аутоматски креирани.

textX се у мод за отклањање грешака може поставити и употребом `textx` команде и параметра `-d` (видети 11).

```
$ textx -d visualize robot.tx program.rbt

 *** PARSING LANGUAGE DEFINITION ***
New rule: grammar_to_import -> RegExMatch
New rule: import_stm -> Sequence
New rule: rule_name -> RegExMatch
New rule: param_name -> RegExMatch
New rule: string_value -> OrderedChoice
New rule: rule_param -> Sequence
Rule rule_param founded in cache.
New rule: rule_params -> Sequence
...

>> Matching rule textx_model=Sequence at position 0 =
  >> Matching rule ZeroOrMore in textx_model at posit
      >> Matching rule import_stm=Sequence in textx_m
        ?? Try match rule StrMatch(import) in import_
        >> Matching rule comment=OrderedChoice in imp
            ?? Try match rule comment_line=RegExMatch
            -- NoMatch at 0
            ?? Try match rule comment_block=RegExMatc

...


Generating 'robot.tx.dot' file for meta-model.
To convert to png run 'dot -Tpng -O robot.tx.dot'
Generating 'program.rbt.dot' file for model.
To convert to png run 'dot -Tpng -O program.rbt.dot'
```

Глава 10

RREL

Претходна секција увела је појам провајдер опсега. У новијом верзијама textX-а постоји и додатни начин дефинисања правила разрешења референци која се наводе директно у граматици као део референце. Овај језик називамо језик израза за разрешење референци (енг. *Reference Resolving Expression Language* --- *RREL*). Ово је преферирани начин дефинисања правила разрешења и у већини случајева је довољан. Уколико је потребна специфична претрага која није подржана са RREL онда се може креирати наменски провајдер опсега који се региструје програмски над моделом (видети претходну секцију).

RREL израз се пише као трећи део референце преко везе (видети 5).

На пример:

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

10.1 Навођење у textX граматици

Ово правило граматике има референцу `ref`. Свака референца преко везе се пише у угластим заградама и има најмање један а највише три дела раздвојена карактером `|`. Први део дефинише тип циљног објекта, у овом случају то је `Class`. Други део дефинише образац који ће парсер да препозна на месту референце. Уколико није дефинисан подразумева се `ID`. Парсер ће у оцом случају препознати правило `FQN`. И на крају, трећи део дефинише RREL израз који дефинише правило проналаска циљног објекта у моделу. У овом случају правило гласи `^packages*.classes`.

Свака референца у моделу генерално је облика имена раздвојених тачком. На пример, референца може бити `package1.component4` или само `component4`. Даље можемо да генерализујемо и кажемо да је референца низ имена где је `ID` само специјалан случај и представља низ дужине један. Имена не морају, у општем случају, бити раздвојена тачком. Корисник може навести наменско правило препознавања и регистровати процесор који ће да разложи име на низ делова. Али због једноставности у наставку сматрамо да се делови имена раздвајају увек са тачком.

10.2 RREL оператори

RREL израз се гради употребом RREL оператора. Дефинисани су следећи оператори:

- `.` --- навигација тачком. Врши се претрага атрибута у текућем AST контексту. Може се користити и за навигацију уз везе садржавања (родитељ-дете). На пример "`.`" је текући објекат, "`..`" је родитељски објекат, "`...`" је родитељ родитеља итд. Ако RREL израз почиње тачком претрага се извршава релативно почевши од текућег AST контекста (текуће локације). У супротном имамо апсолутну путању где претрага почиње од корена модела, осим ако израз не почиње оператором "`^`". На пример, `.a.b` значи претрагу атрибута `a` на текућој локацији и затим претрагу атрибута `b`.
- `parent(TIP)` --- врши се претрага уз родитељске везе све док се не нађе објекат типа `TIP`.

10.3 RREL оператори

- `~` --- маркер који се може применити над елементом израза и који носи информацију да се текућа колекција не претражује по текућем делу имена већ да се цела процесира. На пример, уколико тражимо методу уз хијерархију наслеђивања (веза `extends`), могли би написати `~extends*.methods`, где ће (због `*`, видети у наставку) бити прво претражена `methods` колекција текућег контекста. Затим се итерира кроз све елементе `extends` колекције без конзумације дела имена које се тражи (због `~`) и затим се текући део имена тражи у `methods` колекцији објекта из `extends`.
- `*` --- представља понављање. Резултује развијањем израза нула или више пута. Прва експанзија је нула пута, затим једном, два пута итд. На пример, `~extends*.methods` ће прво да претражује у колекцији `methods`, затим уколико текући део имена није пронађен у `~extends.methods`, затим у `~extends.~extends.methods` итд. Уколико процес не доведе до проналаска циљног објекта, због употребе `*`, израз се даље развија и претрага се наставља док се објекат не пронађе или док не дођемо до краја ланца наслеђивања и тада пријављујемо грешку да објекат није пронађен.

10.4 RREL оператори

- `^` --- претрага од дна ка врху (енг. *bottom-up*). Овај оператор дефинише да се текућа путања развија од дна ка врху, уз родитељски ланац. Претрага почиње на текућем AST контексту и иде се уз родитељски ланац за број компоненти текућег израза. Затим се покушава претрага. На пример, `^a.b.c` почиње на текућем контексту и прво се пење уз родитељски ланац на родитеља. Затим се врши претрага за атрибутом `a` који има име текућег дела имена референце. Затим се тражи атрибут `b`, и на крају се тражи `c`. Уколико претрага не успе, пењемо се уз родитељски ланац један ниво више и поново покушавамо претрагу.
- `,` --- дефинише секвенцу израза које треба покушати у редоследу дефинисања.

10.5 RREL приоритети оператора

Приоритети од највишег до најнижег су: *, ., ,.

~ и ^ се сматрају маркерима а не операторима.

Евалуација RREL израза тече на следећи начин:

- Израз се развија тако што * креће од нуле.
- Навигација и препознавање уз конзумацију препознатих делова имена.
- Процес се понавља.

Процес се зауставља када:

- Све могућности су исцрпљене и нисмо нашли циљни објекат. Грешка.
- При експанзији * дошли смо у ситуацију да смо исцрпели све делове имена пре него што смо завршили са RREL изразом. Грешка.
- Исцрпели смо све делове имена, такође и све делове RREL израза и пронашли смо објекат. Ако се тип објекта не поклапа са оним што је дефинисано референцом у граматици пријављујемо грешку, у супротном успешно смо пронашли објекат.

10.6 RREL пример - граматика/метамодел

Видети textX тестове[2].

[2] https://github.com/textX/textX/tree/master/tests/functional/test_scoping

```
Model: packages*=Package ;

Package:
        'package' name=ID '{'
        (components+=Component
        | instances+=Instance
        | connections+=Connection
        | packages+=Package
        | interfaces+=Interface
        )*
        '}'
;

Interface: 'interface' name=ID;
Component:
    'component' name=ID ('extends' extends+=[Component:FQN|
^packages*.components][','])? '{'
        slots*=Slot
    '}'
;
Slot: SlotIn|SlotOut;
SlotIn:
    'in' name=ID
    ('(' 'format' formats+=[Interface:FQN|^packages*.interfaces][','] ')')?
;
SlotOut:
    'out' name=ID
    ('(' 'format' formats+=[Interface:FQN|^packages*.interfaces][','] ')')?
;

   Instance:
       'instance' name=ID ':' component=[Component:FQN|
   ^packages*.components] ;
   Connection:
       'connect'
         from_inst=[Instance:ID|^instances] '.'
         from_port=[SlotOut:ID|.~from_inst.~component.(~extends)*.slots]
       'to'
         to_inst=[Instance:ID|^instances] '.'
         to_port=[SlotIn:ID|.~to_inst.~component.(~extends)*.slots]
   ;

   FQN: ID+['.'];
   Comment: /\/\/.*$/;
```

10.7 RREL пример - модел

```
package interfaces {
  interface B
  interface C
}
package base {
  interface A
  interface D
  component Start {
    out output1 (format A)
  }
  component Middle {
    in input2 (format A,interfaces.B,interfaces.C)
    out output2 (format interfaces.B,interfaces.C,D)
  }
  component End {
    in input3 (format interfaces.B,interfaces.C,D)
  }
}
package usage {
  instance start : base.Start
  instance action1 : base.Middle
  instance action2 : base.Middle
  instance action3 : base.Middle
  instance end : base.End
  connect start.output1   to action1.input2
  connect action1.output2 to action2.input2
  connect action2.output2 to action3.input2
  connect action3.output2 to end.input3
}
```

Глава 11

`textx` команда

11.1 `textx` команда

Поред употребе textX библиотеке из програмског кода неке од основних операција, као што су, на пример, провера синтаксне исправности метамодела и модела, се могу обавити употребом `textx` ЦЛИ команде. Ова команда је проширива и састоји се од низа команди које су регистроване од стране Путхон пакета. Све команде су регистроване не исти начин и нема разлике између базичних textX команди и команди које су регистроване од стране других Пајтон пакета.

`textx` CLI команда није подразумевано инсталирана када инсталирате textX библиотеку. Да бисте имали ову команду доступну потребно је да инсталирате `cli` зависности на следећи начин:

```
pip install textx[cli]
```

11.2 Основне поткоманде

Основно упутство за употребу може се добити позивом команде без параметара или са параметром `--help` односно `-h`.

```
$ 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.
```

Излистане команде доступне су од стране основне textX библиотеке. Додатне команде ће бити доступне по инсталацији Пајтон пакета који региструју нове textX команде.

11.3 `check` поткоманда

Komanda `check` koristi se za proveru sintaksne ispravnosti modela i metamodela, odnosno gramatike. U slučaju postojanja greške u (meta)modelu biće prijavljena greška sa tačnom lokacijom i prikazom okolnog konteksta. Na primer, da proverimo ispravnost modela (`program.rbt`) ako imamo ispravnu gramatiku (`robot.tx`).

```
$ textx check --grammar robot.tx program.rbt
Error:
/home/igor/repos/textX/textX/examples/robot/program.rbt:3:3:
Expected 'initial' or 'up' or 'down' or 'left' or 'right'
        or 'end' => 'al 3, 1   *gore 4    '
```

Vidimo da u redu 3, koloni 3 imamo grešku. Parser nam prijavljuje šta je očekivano na toj lokaciji i iza znaka `=>` vidimo deo fajla, odnosno kontekst gde se greška nalazi. Karakter `*` obeležava lokaciju unutar konteksta.

11.4 Визуализација `generate` поткомандом

Следеће што можемо урадити јесте визуализација (мета)модела која се обавља командом `generate` која позива регистровани генератор. Генератори су компоненте које генеришу кôд на основу модела. Генератори и језици се могу регистровати (видети 12). Регистровани генератори и језици се листају командама `list-generators` и `list-languages`.

Пошто визуализација представља трансформацију (мета)модела у слику, урађена је као стандардни генератор. Да бисмо видели који генератори су нам доступни позваћемо команду `list-generators` на следечи начин:

```
$ 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...
```

Видимо да имамо три регистрована генератора од стране textX пакета. Први генератор је у стању да трансформише било који модел на *dot* језик. Други генератор трансформише textX моделе (тј. метамоделе) у *dot*. Трећи генератор трансформише метамоделе у PlantUML дијаграме[1].

1. PlantUML је текстуални ЈСД за креирање UML дијаграма --- https://plantuml.com/

11.5 Генерисање `dot` фајла

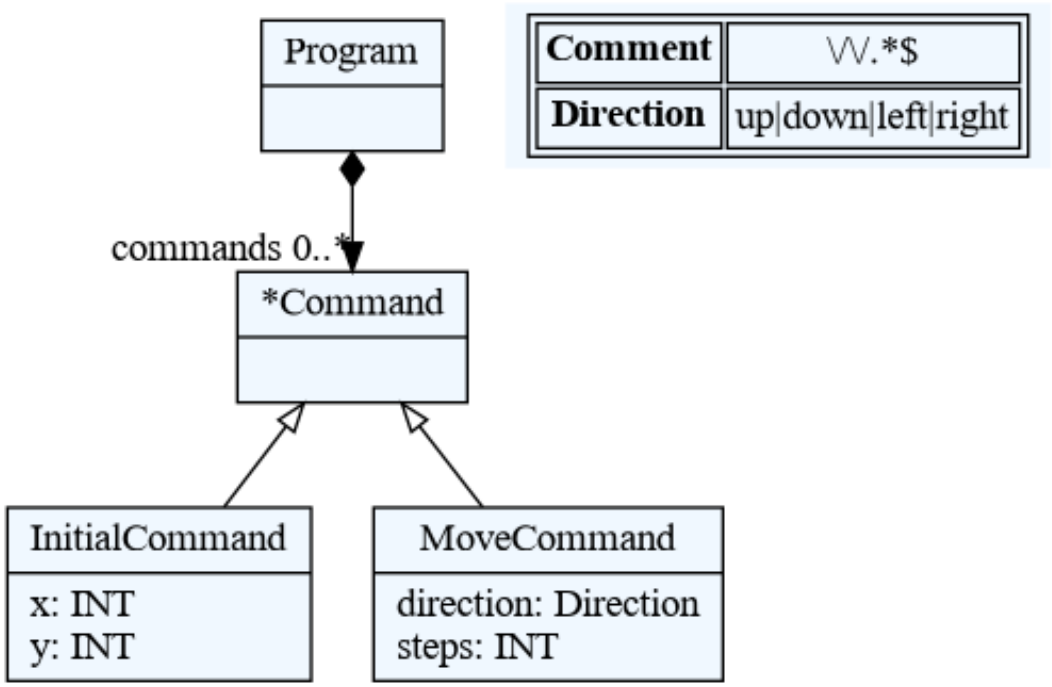
На пример, за визуализацију метамодела потребно је урадити следеће:

```
$ textx generate robot.tx --target dot --overwrite
Generating dot target from models:
/home/igor/repos/textX/textX/examples/robot/robot.tx
-> /home/igor/repos/textX/textX/examples/robot/robot.dot
    To convert to png run "dot -Tpng -O robot.dot"
```

Овом командом позивамо генератор који је регистрован за `.tx` екстензију и бирамо циљни формат/платформу, у овом случају `dot`. Генератор који ће бити позван је `textX -> dot` са списка добијеног употребом `list-generators`. `dot` фајл који смо добили можемо конвертовати у слику према упутству:

```
$ dot -Tpng -O robot.dot
```

Добићемо слику у облику PNG фајла.



Уместо креирања слике, `dot` фајл можемо прегледати неким од `dot` прегледача. На пример:

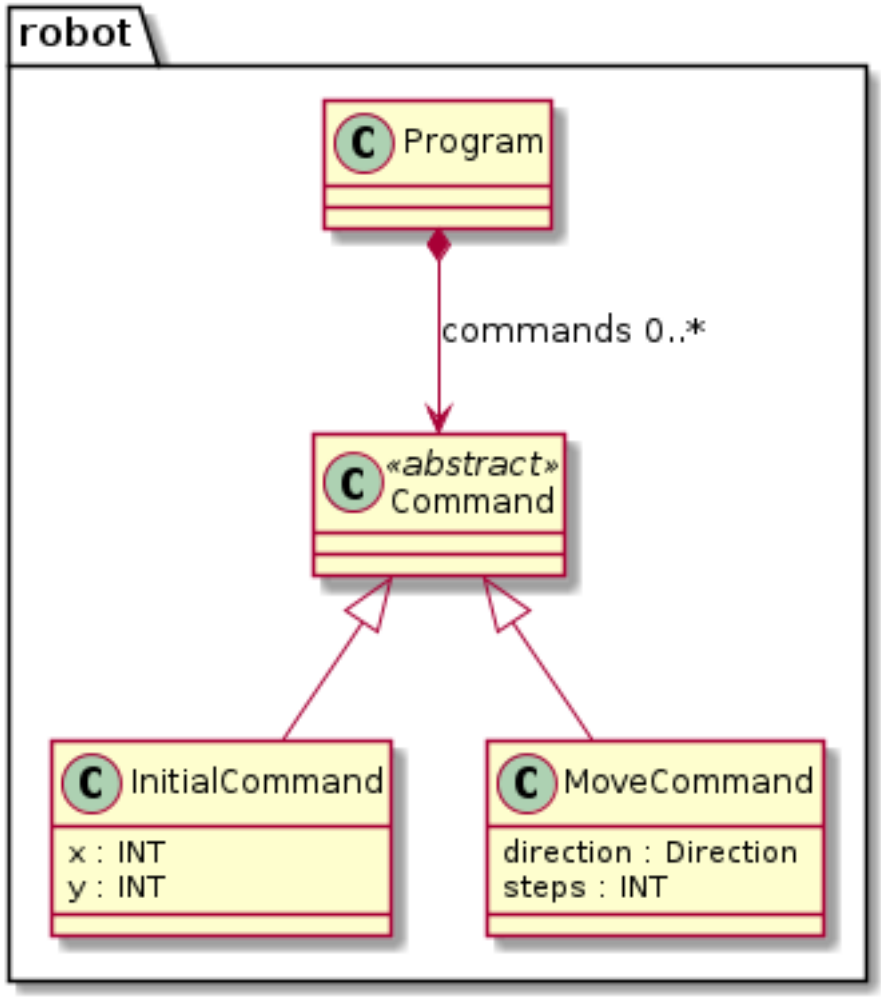
```
$ xdot robot.dot
```

11.6 Конверзија у PlantUML

Алтернативно, метамодел можемо конвертовати у UML дијаграм употребом PlantUML алата. Да бисмо креирали PlantUML фајл из метамодела позивамо команду `generate` на следећи начин:

```
$ textx generate robot.tx --target plantuml --overwrite
Generating plantuml target from models:
/home/igor/repos/textX/textX/examples/robot/robot.tx
-> /home/igor/repos/textX/textX/examples/robot/robot.pu
    To convert to png run "plantuml robot.pu"
```

Затим можемо креирати PNG слику са `plantuml` командом према упутству.



Match rules:

Name	Rule details
Direction	up|down|left|right
Comment	\/\/.*$

11.7 Генерисање `dot` фајла за модел

Уколико желимо да визуализујемо модел потребно је навести и граматику на следећи начин:

```
$ textx generate --grammar robot.tx program.rbt --target dot --overwrite
Generating dot target from models:
/home/igor/repos/textX/textX/examples/robot/program.rbt
-> /home/igor/repos/textX/textX/examples/robot/program.dot
    To convert to png run "dot -Tpng -O program.dot"
```

Глава 12

Регистрација језика и генератора

textX обезбеђује механизам за динамичку регистрацију и откривање језика и генератора кôда. Овим се постиже могућност употребе кориснички регистрованих језика и генератора кроз `textx` команду (видети 11).

12.1 Регистрација језика

Инстанцирати `LanguageDesc` класу где као параметре наводимо: јединствени назив језика, фајл образац/екстензију за моделе на датом језику, опис и функцију (тачније Пајтон *callable*) који врши креирање и конфигурацију метамодела.

```
from textx import LanguageDesc

def entity_metamodel():
    # Funkcija konstruiše i vraća metamodel
    # Npr. poziva metamodel_from_file
    ...

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

Инстанцу `LanguageDesc` затим можемо регистровати употребом `register_language` позива:

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

По обављеној регистрацији метамодел се може добити на следећи начин:

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

Декларативан начин регистрације путем `setup.py`, односно `setup.cfg`. Користимо улазне тачке (енг. *entry points*), стандардан механизам Пајтон `setuptools` пакета:

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

Улазна тачка се зове `textx_languages` - листа стрингова облика "<име језика> = <путања_до_LanguageDesc_инстанце>". У овом примеру инстанца `LanguageDesc` се налази у пакету `entity.metamodel`. Варијабле тј. референца се зове `entity_lang`.

Алтернативно, можемо користити и новији начин употребом `setup.cfg` фајла:

```
[options.entry_points]
textx_languages =
    entity = entity.metamodel:entity_lang
```

Декоратор `language` - параметри су назив језика и екстензија фајлова модел. *Docstring* се користи као опис.

```
from textx import language

@language('entity', '*.ent')
def entity_lang():
    """
    Entity-relationship language
    """
    # Funkcija konstruiše i vraća metamodel
    # Npr. poziva metamodel_from_file
    ...
```

Уколико смо успешно регистровали језик команда `textx list-languages` ће га приказати у листи.

12.2 Регистрација генератора

Користимо инстанцу `GeneratorDesc`. При инстанцирању наводимо: назив језика, назив циљне технологије, опис и на крају функцију која врши генерисање. Функција мора бити следећег облика:

```
def generator(metamodel, model, output_path, overwrite, debug,
              **custom_args)
```

Параметри генератор функције су следећи:

- `metamodel` --- инстанца метамодела изворног језика,
- `model` --- инстанца модела за који вршимо генерисање,

- `output_path` --- циљна путања у фајл систему где треба сместити генерисани кôд,
- `overwrite` --- да ли се врши преписивање циљних фајлова,
- `debug` --- да ли се генератор позива у моду за отклањање грешака,
- `**custom_args` --- додатни параметри специфични за генератор.

Декоратор `generator` - параметри су назив језика и назив циљне платформе. Опис генератора се наводи у *docstring*-у генератор функције.

```
from textx import generator

@generator('entity', 'java')
def entity_java_generator(metamodel, model, output_path,
                          overwrite, debug, **custom_args)
    "Entity-relationship to Java language generator"
    # Код који врши генерисање на основу модела.
```

Регистрација се врши у `setup.cfg` (или `setup.py`) на исти начин као и језик с тим што се улазна тачка назива `textx_generators`:

```
[options.entry_points]
textx_generators =
    entity_java = entity.generators:entity_java_generator
```

По успешној регистрацији команда `textx list-generators` ће листати информације о генератору.

Регистровани генератор се може позвати командом `text generate`. На пример:

```
$ textx generate mymodel.ent --target java --overwrite
        --meaning_of_life 42
```

У претходној команди позивамо генератор регистрован за фајлове са екстензијом `.ent` над моделом `mymodel.ent`. У питању је *Entity* језик. Циљна платформа је `java`. Додатни параметар `meaning_of_life` је специфичан за генератор и биће прослеђен кроз `custom_args` речник.

Глава 13

Креирање иницијалног пројекта - *scaffolding*

Иницијални пројекат креирамо са следећом командом:

```
$ textx startproject <folder>
```

Ова команда покреће генератор који поставља пар питања[1] и затим креира пројекат у задатом фолдеру. Добра пракса је да се затим пројекат инсталира у текуће радно окружење у развојном моду са:

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

1. Одговори на питања се кеширају и биће подразумевана при наредном покретању генератора.

По успешној инсталацији ваш језик, односно генератор је исправно регистрован и видљив за команде `textx list-languages`, односно `textx list-generators`.

Команда `startproject` није дефинисана основном textX библиотеком већ пројектом `textX-dev`. Због тога је потребно инсталирати овај Пајтон пакет или директно са:

```
$ pip install textX-dev
```

или путем инсталације свих развојних зависности за textX на следећи начин:

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

да бисте имали `startproject` доступну као поткоманду `textx` команде.

Глава 14

Примери

14.1 Генерисање кода - Entity пример

- Референцирање других објеката.
- Употреба обрађивача шаблона (**Template Engines**) за генерисање кода.
- http://textx.github.io/textX/stable/tutorials/entity/

14.2 State Machine

- Видео туториал
- http://textx.github.io/textX/stable/tutorials/state_machine/

14.3 Израда мини компајлера - ppci

- Windel Bouwman[3]
- https://ppci.readthedocs.io/en/latest/howto/toy.html

14.4 Quick Domain-Specific Languages in Python with textX

- https://tomassetti.me/domain-specific-languages-in-python-with-textx/
- Written by Alessio Stalla

14.5 PyFlies - језик за психолошке тестове

- https://pyflies.github.io/pyflies/latest/
- https://github.com/pyflies
- Video tutorials[4]

[3] http://www.windel.nl/

[4] https://www.youtube.com/playlist?list=PLOgHC04drILVjnXQTFEL7sJDyfKMR21Vg

Глава 15

Протокол језичких сервера (*Language Server Protocol - LSP*)

Развој парсера и генератора кода за ЈСД је само основни део посла. Да би ЈСД био прихваћен од стране корисника потребно је да постоје за њега развијени едитори, дебагери, профајлери и сл. Односно, потребно је развити и одржавати заједно са ЈСД и комплетан ланац алата (енг. *toolchain*) који омогућавају бољи кориснички доживљај.

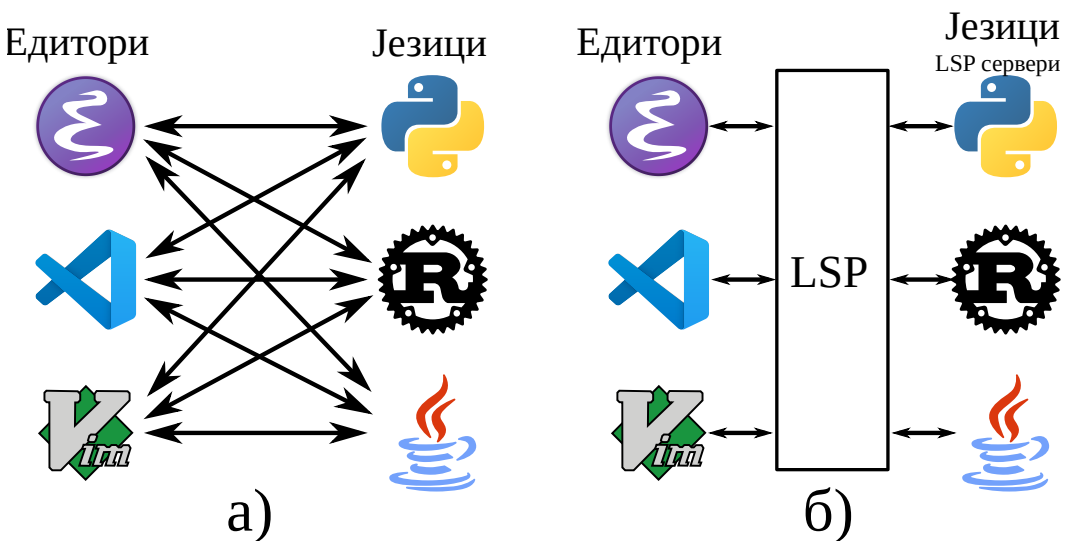
Када су у питању едитори, корисници су навикли последњих деценија да имају подршку за допуну кода (*code completion*), контекстну документацију (енг. *documentation on hover*), одлазак на дефиницију (енг. *goto definition*), рефакторисање и др. Имплементација ових функционалности захтева значајне ресурсе. Додатни проблем је што се за сваки програмски језик мора наново имплементирати језичка подршка за сваки едитор односно интегрисано окружење. Проблем је делимично релаксиран код едитора који деле платформу односно програмски језик на коме су базирани, па самим тим могу делити и део имплементације језичке подршке.

На пример, постоји више популарних едитора базираних на JavaScript Electron[5] развојном оквиру (Atom[6], Visual Studio Code[7]). Тиме се одређене библиотеке и елементи језичке подршке могу делити.

У основи, имамо проблем имплементације `m x n` компоненти. За сваки едитор морамо имплементирати подршку за сваки програмски језик (слика 12а).

Да би се повећала поновна искористивост језичких подршки, фирма Microsoft је у склопу свог *VS Code* едитора издвојила језичку "памет" у посебну серверску компоненту са којом едитор комуницира у улози клијента. Протокол комуникације је данас отворени стандард под називом *протокол језичких сервера* (енг. Language Server Protocol - LSP[8]).

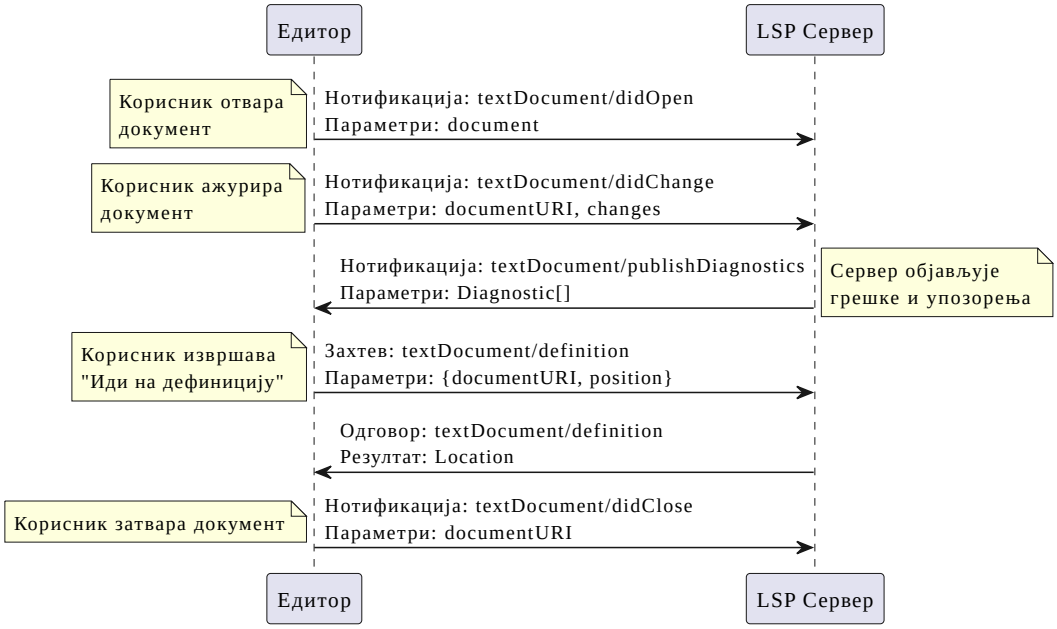
[5] https://www.electronjs.org/
[6] https://atom.io/
[7] https://code.visualstudio.com/
[8] https://microsoft.github.io/language-server-protocol/



Слика 12: Језичка подршка у едиторима: а) без LSP б) употребом LSP језичких сервера.

За сваки програмски језик потребно је имплементирати језичку подршку у виду LSP сервер компоненте једном. Сви едитори који разумеју LSP протокол аутоматски добијају подршку за све језике за које постоји LSP сервер. Овим се ефективно димензија проблема свела са `m x n` на `m + n`, где су `m` и `n` број едитора и број програмских језика (слика 12б). Додатна предност приступа је што се сада сва енергија која се троши на развој језичке подршке за одређени језик концентрише у једну LSP компоненту чиме се добија на квалитету.

LSP протокол је базиран на JSON-RPC, односно позивању удаљених процедура (енг. *Remote Procedure Call - RPC*) разменом JSON порука.



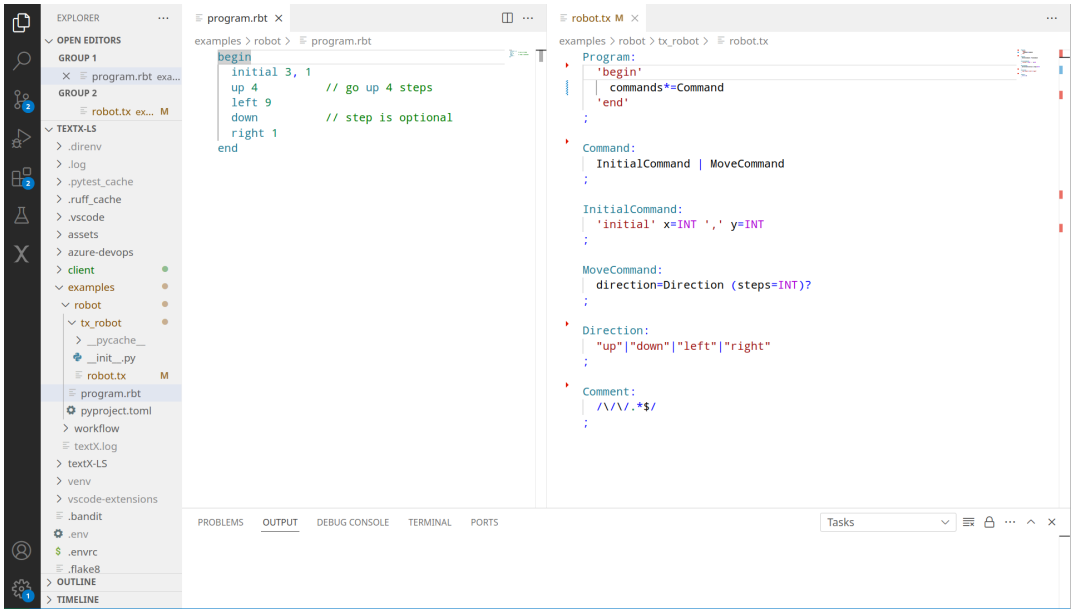
Слика 13: Пример комуникације едитора и LSP сервера.

На слици 13 дат је пример размене порука између едитора (клијента) и LSP сервера при операцијама отварања фајла, ажурирања, захтева за одлазак на дефиницију и затварања.

Глава 16

textX-LS

Алат за креирање ЈСД textX има подршку за LSP кроз пројекат textX-LS[9]. Идеја овог пројекта је аутоматско генерисање језичких сервера за све језике који су креирани употребом textX алата. Библиотека која пружа генеричке сервисе и омогућава брз развој језичких сервера на програмском језику Пајтон је pygls[10].

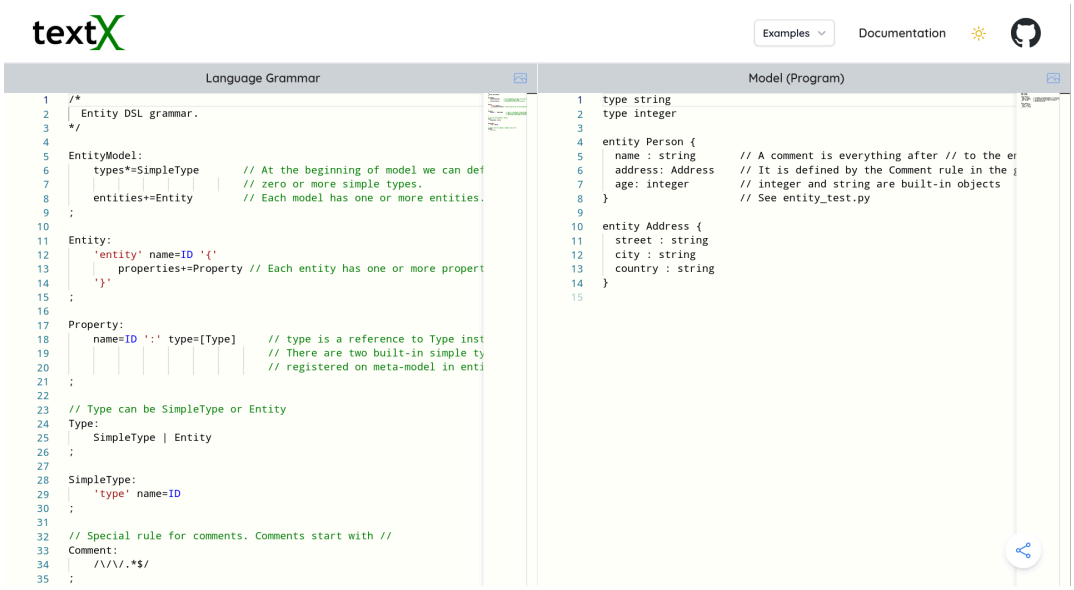


[9] https://github.com/textX/textX-LS
[10] https://github.com/openlawlibrary/pygls

Глава 17

textX игралиште (*textX playground*)

- Веб апликација доступна на адреси https://textx.github.io/textx-playground/
- Омогућава једноставно испробавање и играње са textX језицима без потребе за локалном инсталацијом.



Глава 18

Литература

- Игор Дејановић, *Језици специфични за домен*, Факултет техничких наука, Нови Сад, 2021. (доступно у скриптарници ФТН-а)
- textX dokumentacija[11]

[11] https://textx.github.io/textX/