Resolução do Trabalho III

Linguagens de Programação

Gabriel Weich e Paulo Aranha

Escola Politécnica – PUCRS

Agosto de 2018

Introdução

A proposta deste estudo é apresentar uma gramática gerada pelo *framework* Xtext para a construção de entidades e um gerador de SQL e Python[1] para a gramática. Iremos descrever as regras aplicadas nas transformações bem como os resultados obtidos.

Gramática

A gramática teve como base o exemplo fornecido pelo tutorial disponível na página do *Xtext* [1]. Fizemos algumas modificações a fim de melhor adaptar a gramática para gerar o script SQL e o código em Python.

Primeiramente removemos os comandos de *import* e de refência a *packages*, mantendo a criação das entidades em primeiro nível. Incluímos os tipos de dados mais comuns presentes no SGBD *Postgres* além de incluir as *constraints* "allowNull"[2] e "unique".

A gramática não permite a criação de chaves primárias, pois estas serão criadas automaticamente para cada estrutura, tal como acontece no *MongoDB*. As chaves estrangeiras serão criadas a partir da referência para uma entidade.

Alguns recursos da gramática são:

- SQL
  - Mapeamento de atributos e relações 1:1, 1:n, n:n
  - Geração automática de Primary Keys
  - Restrições allowNull e unique
  - Valores default para atributos
  - Suporte à herança
- Python
  - Mapeamento de atributos e relações 1:1
  - Geração classes e atributos com *typehints*
  - allowNull são tratados incluindo *None* com default

[1] Compatível com Python 3.7+ e Postgres 9.6+

[2] Todas as colunas são "NOT NULL"por *default*

  - Suporte à herança
  - Todos os *imports* necessários são gerados automaticamente

- .sh: Um script .sh é gerado automaticamente para facilitar a criação do banco e das tabelas. O banco pode ser gerado através do seguinte comando:

```
$ bash nomedobanco.sh
```

Elaborando a lógica

A parte mais complexa na construção do gerador foi o mapeamento das relações nas devidas tabelas. Segue uma explicação de como foram implementos os relacionamentos:

Relacionamento 1:1

Ao adicionar uma referência simples para outra entidade tem-se um relacionamento 1:1, o qual se dá quando cada um dos elementos de uma entidade só pode se relacionar com apenas um elemento da outra entidade, tal referência é implementada através da geração de uma chave estrangeira na tabela pai para a tabela filha. Ao gerar o script Python, um atributo "id_"+«nome da tabela filha» é adicionado à classe Pai.

Se adicionado, o modificador allowNull permitirá uma referência nula no SQL e um valor default *None* para o atributo gerado no Python.

Exemplo

.gesiel

```
entity Conta {
    login: varchar(30)
    senha: varchar(20)
    data_cadastro: timestamp = "now()"
}

entity Pessoa {
    nome: text
    cpf: char(11)
    Conta allowNull
}
```

.sql

```
drop database if exists teste;
create database teste;
\connect teste

CREATE TABLE conta (
  id_conta SERIAL,
  login VARCHAR(30) NOT NULL,
  senha VARCHAR(20) NOT NULL,
  data_cadastro TIMESTAMP NOT NULL DEFAULT now(),
  CONSTRAINT conta_pk PRIMARY KEY (id_conta)
);

CREATE TABLE pessoa (
  id_pessoa SERIAL,
  nome TEXT NOT NULL,
  cpf CHAR(11) NOT NULL,
```

```
  id_conta INTEGER,
  CONSTRAINT pessoa_pk PRIMARY KEY (id_pessoa)
);

ALTER TABLE pessoa ADD CONSTRAINT pessoa_fk0 FOREIGN KEY (id_conta)
    REFERENCES conta(id_conta);
```

.py

```
#conta.py
from dataclasses import dataclass
from datetime import datetime

@dataclass
class Conta:
    id_conta: int
    login: str
    senha: str
    data_cadastro: datetime


#pessoa.py
from dataclasses import dataclass

@dataclass
class Pessoa:
    id_pessoa: int
    nome: str
    cpf: str
    id_conta: int = None
```

Relacionamento 1:n

Quanto uma entidade faz referência à outra entidade incluindo o modificador *many* significa que cada elemento da entidade "A" pode ter um relacionamento com vários elementos da entidade "B". Nesse caso, o identificador da entidade pai migra para a tabela que implementa a entidade filha, gerando uma chave estrangeira para a tabela pai.

Exemplo

.gesiel

```
entity Telefone {
    ddd: char(2)
    numero: varchar(10)
}

entity Pessoa{
    nome: text
    cpf: char(11) unique
    many Telefone
}
```

.sql

```
CREATE TABLE telefone (
  id_telefone SERIAL,
  ddd CHAR(2) NOT NULL,
  numero VARCHAR(10) NOT NULL,
  id_pessoa INTEGER NOT NULL,
  CONSTRAINT telefone_pk PRIMARY KEY (id_telefone)
```

```
);

CREATE TABLE pessoa (
  id_pessoa SERIAL,
  nome TEXT NOT NULL,
  cpf CHAR(11) NOT NULL UNIQUE,
  CONSTRAINT pessoa_pk PRIMARY KEY (id_pessoa)
);

ALTER TABLE telefone ADD CONSTRAINT telefone_fk0 FOREIGN KEY (id_pessoa)
    REFERENCES pessoa(id_pessoa);
```

Relacionamento n:n

Neste tipo de relacionamento cada entidade, de ambos os lados, podem referenciar múltiplas unidades da outra, para uma referência desse tipo usa-se o modificador *many* antes da entidade filha juntamente com zero ou mais submembros entre chaves após a referência à entidade. Nesse caso, uma nova tabela é criada e a chave primária é a composição das chaves primárias das duas tabelas relacionadas.

Os submembros são atributos normais ou relações 1:1 e constituirão colunas adicionais na tabela intermediária.

Exemplo

.gesiel

```
entity Produto{
    nome: text
    preco: numeric(6,2)
}

entity Compra {
    data: date
    many Produto {
        desconto: int allowNull
    }
}
```

.sql

```
CREATE TABLE produto (
  id_produto SERIAL,
  nome TEXT NOT NULL,
  preco NUMERIC(6,2) NOT NULL,
  CONSTRAINT produto_pk PRIMARY KEY (id_produto)
);

CREATE TABLE compra (
  id_compra SERIAL,
  data DATE NOT NULL,
  CONSTRAINT compra_pk PRIMARY KEY (id_compra)
);

CREATE TABLE compra_produto (
  id_compra INTEGER,
  id_produto INTEGER,
  desconto INTEGER,
  CONSTRAINT compra_produto_pk PRIMARY KEY (id_compra,id_produto)
);
```

```
20
21 ALTER TABLE compra_produto ADD CONSTRAINT compra_produto_fk0 FOREIGN KEY (
       id_compra) REFERENCES compra(id_compra);
22 ALTER TABLE compra_produto ADD CONSTRAINT compra_produto_fk1 FOREIGN KEY (
       id_produto) REFERENCES produto(id_produto);
```

Herança

Através da herança uma entidade pode herdar os atributos de outra entidade. A gramática possiblita a herança através do comando *extends* após a definição da entidade.

Exemplo

.gesiel

```
entity Pessoa{
  nome: varchar(50)
  cpf: char(11) unique
}

entity Funcionario extends Pessoa{
  data_contratacao: timestamp
  contrato: text
}
```

.sql

```
CREATE TABLE pessoa (
  id_pessoa SERIAL,
  nome VARCHAR(50) NOT NULL,
  cpf CHAR(11) NOT NULL UNIQUE,
  CONSTRAINT pessoa_pk PRIMARY KEY (id_pessoa)
);

CREATE TABLE funcionario (
  id_funcionario SERIAL,
  data_contratacao TIMESTAMP NOT NULL,
  contrato TEXT NOT NULL,
  CONSTRAINT funcionario_pk PRIMARY KEY (id_funcionario)
) INHERITS (pessoa);
```

.py

```
#pessoa.py
from dataclasses import dataclass

@dataclass
class Pessoa:
    id_pessoa: int
    nome: str
    cpf: str


#funcionario.py
from dataclasses import dataclass
from pessoa import Pessoa
from datetime import datetime

@dataclass
class Funcionario(Pessoa):
    id_funcionario: int
```

```
19      data_contratacao: datetime
20      contrato: str
```

Limitações

Entre as principais limitações e restrições da linguagem destacam-se:

- Não há tratamento para relações 1:n e n:n no Python (Não foi implementado devido a uma maior complexidade de desenvolver tais relações e por não haver uma forma padrão de tratá-las)
- Impossibilidade de gerar uma restrição unique composta.
- Entidades referenciadas por outras entidades devem ser declaradas antes da referência, pois a geração de tabelas se dá na ordem das declarações e uma tabela que referencia outra ainda não criada causa erro no script.
- Devido à construção da gramática todas as relações devem ser declaradas após todos os atributos.

Conclusão

Conseguimos evoluir a gramática desenvolvida no trabalho anterior incluindo recursos que ficaram pendendes naquela versão, tal como o reconhecimento de relações n:n, a quantidade de casas em tipos como varchar e numeric, atributos default, além da refatoração do código da gramática.

A partir do desenvolvimento do trabalho passamos a conhecer mais sobre a construção de gramáticas e do uso das ferramenta Xtext e Xtend para a geração das mesmas.

Na atual versão temos uma gramática simples e consistente. Para uma futura versão pretendemos ampliar a gramática reparando as deficiências que ficaram pendentes nessa primeira versão.

Referências

[1] 15 Minutes Tutorial - Xtext, acesso em (2018, 06 de setembro), `https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html`

[2] 15 Minutes Tutorial - Extended - Xtext (2018, 06 de setembro), `https://www.eclipse.org/Xtext/documentation/103_-domainmodelnextsteps.html`

[3] CREATE TABLE (2018, 06 de setembro), `https://www.postgresql.org/docs/9.1/static/sql-createtable.html`

[4] SQLAlchemy(2018, 06 de setembro), `https://docs.sqlalchemy.org/en/latest/orm/tutorial.html`

[5] dataclasses — Data Classes(2018, 25 de setembro), `https://docs.python.org/3/library/dataclasses.html`