GREGORY SHOLL E SANTOS

GUILHERME CARLOS POLITTA

TRABALHO SEM TITULO

Trabalho apresentado como requisito parcial à Obtenção de grau de Bacharel em Ciências da Computação no curso de graduação em Bacharelado em Ciências da Computação, Departamento de Informática da Universidade Federal do Paraná.

Orientador: Prof. Bruno Muller Junior

CURITIBA

2014

1 INTRODUÇÃO

Empresas com mais tempo no mercado e que possuem sistemas criados a muitos anos, têm dificuldades de migra-los para tecnologias mais recentes. Essa dificuldade se dá principalmente pelo custo da migração manual de uma banco de dados para a tecnologia mais contemporânea.

Vinculado a este custo estão o aprendizado de ambas tecnologias, a original e a que se deseja assumir, e desafio de manter o banco de dados novo consistente com o atual. Ao contar todos esses recursos, o financeiro, o temporal e o humano, o custo chega a tal que pode vir a ser inviável para a empresa fazer esse processo. Sendo assim, muitas empresas optam por não trocar de tecnologia, mantendo sistemas de difícil manutenção e que não possuem recursos modernos para desenvolvimento e gerência.

Um processo automatizado de migração de um modelo de dados entre duas tecnologias pode facilitar e reduzir os custos envolvidos, incentivando está mudança. Esse processo precisa ser capaz de analisar o código de criação do banco de dados original e interpretá-lo para seu formato equivalente na tecnologia desejada, mantendo todas as relações entre os dados.

Existem hoje vários *frameworks* que são capazes de criar uma aplicação a partir do modelo do banco de dados. Um deles é o *Ruby on Rails*, que apresenta uma forma de criar um esqueleto da aplicação através de linhas de comando que descrevem o modelo de dados. Outros *frameworks* que operam de forma semelhante são o ASP.NET, o JavaEE e o OpenACS.

Portanto, se esse processo automatizado for capaz fazer a migração para um desses *frameworks,* a adoção destas tecnologias por parte das empresas seria muito mais ampla.

Este trabalho propõe um *software* que gere a estrutura básica de uma aplicação em *Ruby on Rails*, dado o modelo de dados de uma aplicação já existente. É importante reforçar que a proposta é da geração da estrutura básica da aplicação, ou seja, toda e qualquer lógica envolvida na manipulação e utilização ainda terá de ser implementada manualmente pelo usuário.

O trabalho está organizado da seguinte maneira: conceitos e ferramentas de *Rails*, banco de dados e analisadores léxicos e semânticos no capítulo 2, proposta de um software que gere os códigos Rails no capitulo 3, uma implementação da proposta no capítulo 4 e conclusão no capítulo 5.

2 Revisão Bibliográfica

Esse capítulo apresenta as ferramentas e conceitos utilizados no trabalho. A seção **X** apresenta a linguagem declarativa SQL, a seção **X** os Sistemas de Gerenciamento de Banco de Dados (SGBD), a seção **X** o *framework Ruby on Rails*, a seção **X** os analisadores sintáticos e a seção **X** os analisadores léxicos.

2.1 Linguagem SQL

O SQL (do inglês, *Structured Query Language*) é uma linguagem de programação feita para gerenciar sistema de banco de dados relacionais que foi desenvolvida no começo dos anos 70 por Donald D. Chamberlin e Raymond F. Boyce. Atualmente, SQL é muito utilizado em gerenciamento e desenvolvimento de banco de dados *Web*.

A linguagem é originalmente baseada em álgebra relacional, facilitando o seu entendimento. Ela é dividida nas linguagens de definição e de manipulação de dados.

2.1.2 Linguagem de definição de dados

A linguagem de definição de dados é a que altera as estruturas e restrições do dados, no caso do SQL essas estruturas são as tabelas do banco de dados. Esse subconjunto na linguagem SQL é denomiado de SQL Schema. A figura 1 mostra um exemplo de criação de uma tabela Pessoa.

FAZER IMAGEM

```
CREATE TABLE Pessoa (
    id          INTEGER         PRIMARY_KEY,
    nome        VARCHAR(50)     NOT NULL,
    sobrenome   VARCHAR(100)    NOT NULL,
    telefone    INTEGER         NULL
);
```

Além do CREATE para criação de tabelas, pode-se destruir parcial ou totalmente os dados, permitir ou revogar acesso às tabelas (todas funções consideram que já foram criadas as tabelas previamente). As chamadas para essas funções são DROP, GRANT e REVOKE, respectivamente.

A função DROP pode, por exemplo, tirar somente a coluna de data de nascimento da tabela, assim como também pode remover a tabela inteira. Já a função GRANT permite acesso a dados do objeto alterado a um grupo de usuários. A função REVOKE é a que revoga acesso a dados do objetoa certos usuários.

2.1.3 Linguagem de manipulação de dados

A linguagem de manipuação de dados é aquela que lida com os dados, propriamente dito. Essas linguagem são as que inserem, apaguam, atualizam e selecionam os dados desejados.

A manipulação de dados incluem funções como INSERT (inserção), DELETE (remoção) e SELECT (seleção) dos dados dos banco de dados. Essa parte da linguagem não é relevante para o domínio deste trabalho e, portanto, não será detalhada.

2.2 SGBD

SGBD vem da sigla Sistema de Gerenciamento de Banco de Dados (em inglês, *Database Management System*) e é um conjunto de programas que gerenciam o acesso e manutenção de um banco de dados. Os SGBDs de banco de dados relacionais são geralmente acompanhados de uma interface simples e intuitiva para o usuário, retirando a necessidado do usuário intervir na funcionalidade do gerenciamento.

Um banco de dados geralmente não é portável para outro SGBD que não seja aquele onde foi criado, mas SGBDs distintos podem se comunicar através de algum protocolo para fornecerem dados entre si.

Existem três características fundamentais que todos SGBDs devem ter: linguagem de definição de dados, linguagem de manipulação de dados e processamento eficaz de consultas. As duas primeiras já foram definidas na seção 2.1.

INSERIR ALGUMA IMAGEM PARA ESCLARECER

Um SGBD precisa ter integridade semântica, ou seja, manter os seus dados sempre correto em relação ao domínio da aplicação. Caso contrário, perde-se o principal propósito de se utilizar um banco de dados.

Os SGBDs também possuem outras características importantes como a segurança no armazenamento dos dados e sua concorrência. Entretanto, estas características não fazem parte do escopo deste trabalho e não serão aprofundadas.

2.3 *Framework* Ruby on Rails

Ruby on Rails, ou somente *Rails*, é um *framework* gratuito e de código aberto destinado a aumentar a velocidade e facilitado com que *sites* orientados a bancos de dados são feitos.

Esse *framework* utiliza diversos padrões de engenharia de *software* e paradigmas, como convenção sobre configuração (do inglês, *Convention over Configuration*), *Don't Repeat Yourself* (traduz para Não Se Repita), o padrão *Active Record* e o modelo MVC.

Como em muitos *frameworks web*, o *Rails* utiliza o modelo *model-view-controller* (também conhecido como MVC) para organizar a programação do projeto. Esse modelo é dividido em três componentes: modelo, visão e controlador.

O componente modelo é responsável por gerenciar diretamente os dados, lógicas e regras da aplicação. Ele também notifica os outros dois componentes que mudanças foram feitas no seu estado, permitindo mudança na saída da visão e no conjunto de comandos disponível no controlador.

A visão é responsável por gerar alguma forma de representação dos dados armazenados, seja por texto, diagrama, tabelas etc.

O controlador pode enviar comandos para o modelo para que mudanças sejam feitas nos dados, assim como também pode mandar comandos para a visão associadas para mudar a forma de exibição dos dados (por exemplo, rolar por um documento mais extenso).

A figura 4 abaixo mostra um diagrama da interação desse componentes entre si e com o usuário. Note que o usuário não sabe de que forma os dados estão armazenados, pois para ele isso não importa, geralmente. Ele visualiza apenas as respostas dos comandos gerados por ele. Os comandos são interpretados pelo controlador que os traduz para um comando que o modelo possa executar. Ao realizar o comando, o modelo adquire um retorno e o passa para a visão atualizar a sua mostragem.

INSERIR IMAGEM AQUI

O *Active Record* é um padrão de projeto de *softwares* que utilizam bancos de dados relacionais. É uma abordagem para o acesso ao banco de dados. Uma tabela desse banco de dados é embrulhada (do inglês, *wrapped*) em uma classe de programação orientada a objetos. Assim, um objeto dessa classe fica ligado a somente uma tupla dessa tabela.

Ao criar um novo objeto, uma nova tupla também é criada na tabela. Mudanças como atualização e seleção de objetos utilizam o banco de dados, porém de uma forma mais intuitiva para os programadores.

A figura 5 mostra um exemplo de um pseudo-código do padrão *Active Record* e também a sua versão equivalente em SQL.

INSERIR CÓDIGO ACTIVERECORD/SQL

A biblioteca *Active Record* do Ruby implementa ORM (do inglês, mapeamento objeto-relacional), que é uma técnica de desenvolvimento utilizada para diminuir os conflitos entre programação orientadas a objetos e banco de dados relacionais. Isso cria um modelo de domínio persistente, onde a lógica e os dados são apresentados como um pacote unificado.

O *Rails* possui uma série de linhas de comandos para facilitar o seu uso no dia-a-dia. A estrutura básica de uma aplicação *Rails* pode ser criada inteiramente por essas linhas de comandos. Dentre esses comandos, dois possuem maior importância nesse trabalho que é o *new* e o *generate*.

O *rails new* cria todo o esqueleto da aplicação desejada. No caso deste trabalho, toda a construção da aplicação final é feita por linhas de comando. As

próximas figuras demonstram esse esqueleto criado para uma aplicação qualquer.

INSERIR IMAGENS DA ESTRUTURA FEITA PELO RAILS NEW

O *rails generate* possui várias utilizações, como criar controladores, visões, etc. O *generate* executa as suas tarefas dentro da estrutura criada pelo *rails new*, supondo que este comando já tenha sido executado anteriormente. A importância do *generate* neste trabalho é para criação de modelos.

É possível passar os atributos que compõem o modelo e quais as suas relações com outros modelos. Ao final de tudo, ao executar o modelo, será criada uma tabela em um banco de dados relacional com as informações passadas pelo *generate*.

A figura 6 mostra um exemplo de utilização do *rails generate* e a figura 7 retrata a tabela que será gerada ao final da execução.

INSERIR FIGURAS DO GENERATES E DA TABELA FINAL

2.4 Analisadores sintáticos

Análise sintática é uma técnica emprega no estudo da estrutura sintática de uma linguagem (citação), conforme as regras de uma gramática formal. Essa análise resulta em uma árvore ou outra estrutura hierárquica que mostra o relacionamento entre cada símbolo reconhecido.

Um analisador sintático tem a função de determinar como a entrada pode ser derivada a partir do símbolo inicial da gramática. Existe duas maneiras de realizar essa tarefa:

1. *Top-down* ou descendente. O analisador começa do símbolo mais alto da hierarquia de análise e tenta transforma-lo na entrada
2. *Bottom-up* ou Ascendente. O analisador começa com a entrada de dados e tenta reescreve-la até chegar ao símbolo inicial mais alto da hierarquia.

O analisador LL (Left-to-right, Leftmost derivation) faz a análise *Top-down* indo da direita pra esquerda, preferindo sempre derivar o símbolo mais à esquerda. Já o analisador LR (Left-to-right, Rightmost derivation) faz a análise

Botton-up, indo da direita pra esquerda, preferindo derivar o símbolo mais à direita. Ambos são normalmente acompanhados de um número k para indicar que são autorizados a olhar k entradas a frente para evitar *backtracking* ou adivinhação.

2.4.1 YACC

YACC (do inglês, *Yet Another Compiler Compiler*) é um gerador de analisadores sintáticos LALR (*Look-Ahead Left-to-Right parser*), desenvolvido no começo dos anos 1970 por Stephen C. Johnson.

Normalmente utilizado para a construção de compiladores, o YACC proporciona uma ferramenta na qual o usuário específica uma estrutura de entrada junto com um código para ser invocado cada vez que uma estrutura é reconhecida.

A figura 1 mostra o YACC sendo utilizado. No painel A coluna temos um trecho de código em Pascal que será utilizado como entrada para o código YACC, no painel B temos a estrutura definida por um usuário com o YACC e no painel C a saída gerada em Java. Como pode ser visto, o código YACC detecta quando uma linha começa com a palavra-chave *var* seguida de um identificador qualquer, dois pontos, um segundo indentificador e um ponto e vírgula, gerando a saída em Java "*String x;*". Em blocos entre colchetes é possível escrever qualquer código em C, portanto poderíamos, por exemplo, fazer a verificação de que se *String* é um tipo suportado para a estrutura.

FAZER A IMAGEM DE EXEMPLO.

2.5 Analisadores Léxicos

Análise léxica é o processo de converter uma sequência de caracteres em símbolos ou *tokens*, permitindo que seja feita a verificação de que esses caracteres pertencem ao alfabeto de análise. Portanto, um analisador léxico implementa um autômato finito que reconhece símbolos como sendo válidos ou não a uma certa linguagem.

A implementação desses analisadores requerem a descrição do autômato que reconhece a gramática ou expressão regular desejada. A sequência de caracteres de entrada é estruturada como uma lista de símbolos, que o analisador vai utilizar como entradas para o autômato.

Se o analisador terminar de consumir os símbolos em um estado final, a entrada é dado como válida, se acabar em um estado não final ou não houver estado para o qual possa ir, a entrada é considerada inválida para a gramática.

2.5.1 Lex

Lex é um gerador de analisadores léxicos, escrito por Mike Lesk e Eric Schmidt em 1975. Ajuda a escrever programas cujo fluxo de controle é dirigido por instâncias de expressões regulares, ou seja, as entradas podem ser interpretadas por essas expressões.

Um código Lex pode ser dividido em duas partes: a declaração da expressão regular e a sequência de ações de devem ser executadas quando essa expressão é reconhecida.

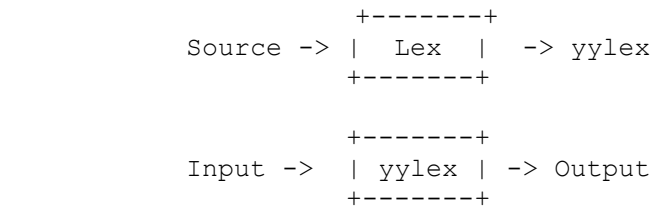
A figura 2 mostra um trecho de código que utiliza Lex. Na linha 5 temos uma expressão que reconhece sequências de caracteres que começam com uma letra (maiúscula ou minúscula) seguida de zero ou mais letras e números, em qualquer ordem. Já a linha 9 reconhece quando a expressão previamente declarada é reconhecida e execute um código qualquer, nesse caso só imprime um texto no terminal.

FAZER IMAGEM (ou talvez código mesmo) COM O CODIGO ABAIXO:

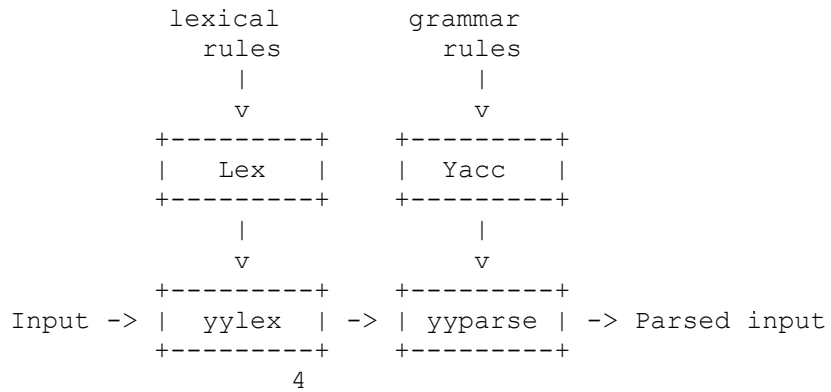
```
ident [a-zA-Z][a-zA-Z1-9_]*


{ident} { printf(“encontrei um identificador válido!” ) }
```

Como o Lex só consegue trabalhar com máquinas de estado finito e o YACC não consegue ler simples entradas de dados, trabalhando apenas com

uma série de *tokens,* as duas ferramentas são utilizadas em conjunto, de forma que o Lex serve como um pré-processador para o YACC, gerando os *tokens* que ele necessita. A figura 3 e 4 explicam essa relação. O Lex recebe a entrada de dados e as expressões regulares, gerando a rotina chamada *yylex,* a saída gerada serve como entrada para a rotina *yyparse* criada pelo YACC usando as regras gramaticais. Portanto, cada vez que o YACC precisa de um novo *token* ele invoca o Lex, que processa os dados de entrada e retorna a primeira expressão identificada.



3



4

Esse trabalho tem como objetivo propor uma aplicação que recebe um SQL *schema* e utiliza o YACC e o Lex para traduzi-lo para um conjunto de comandos *Rails,* que juntos constroem o banco de dados e um esqueleto de telas para a entrada.

Parte referente a: Capitulo 2: Rails (generates), SGBD/SQL (sintaxe), Gramáticas / Parser, YACC (mais sintaxe). **Reescrever objetivos com nova "linguagem"**

Faltou essa parte de reescrever os objetivos, acho melhor escrever a introdução antes.

3 Proposta do Trabalho

A implementação proposta nesse trabalho é da construção de uma ferramenta que, a partir de um SQL *Schema* de entrada, gere comandos *rails generate*, com o intuito de facilitar a migração do banco de dados de um projeto legado para uma aplicação em *Ruby on Rails.*

Para isso, utilizamos as ferramentas YACC e Lex, como interpretadores do SQL *Schema*, e a linguagem de programação C, para unir as duas ferramentas e gerar os comandos de saída.

Através do YACC criamos uma estrutura que reconhece as diferentes formas de criação de tabelas que podem ser construídas utilizando o SQL *Schema.* O Lex, por outro lado, reconhece os símbolos, ou *tokens,* válidos que podem aparecer nessa criação. Quando um criação de tabela é detectada com sucesso, criamos o comando *rails generate* equivalente a ela.

A figura X mostra um exemplo da entrada da nossa ferramenta. Essa é composta de uma ou mais tabelas descritas em SQL *Schema* e separadas por uma linha em branco. A saída esperada para dita entrada, exemplificada na figura X, são comandos *rails generate* que descrevem o mesmo modelo passado como entrada, mantendo as mesmas relações entre as tabelas. Podemos notar que os campos customer_id e order_id não são explicitamente declarado, já que o *rails generate* cria esses campos automaticamente.

```
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_name VARCHAR(100)
);


CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    amount DOUBLE,
```

```
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

rails generate scaffold customers customer_name:string --force

rails generate scaffold orders customers:references amount:decimal{42.6} --force

Conceitual, proposta do trabalho;

Gramática de entrada -> Parser -> Saida (comando rails)

4 IMPLEMENTAÇÃO

A implementação da ferramenta proposta foi dividida em três partes: a leitura do arquivo do SQL *Schema*, a intepretação desse arquivo e a geração dos comandos *Rails.*

A primeira foi feita com o Lex, onde criamos uma séries de expressões regulares que definem todos os *tokens* possíveis para o SQL *Schema*, portanto ele não só lê a entrada de dados, mas também verifica se os símbolos lidos são válidos. A lista de expressões regulares, assim como os símbolos que elas definem, estão disponíveis no apêndice 1.

CRIAR UMA TABELA COM AS COLUNAS: EXPRESSÃO REGULAR NO LEX E SÍMBOLOS RECONHECIDOS PELO COMANDO COMO APENDICE

Já a interpretação dos símbolos lidos foi feita com o YACC. Com ele criamos uma estrutura que reconhece as sequências possíveis de símbolos e, com isso, verificando se a entrada está de acordo com o padrões do SQL *Schema*, reconhecendo os identificadores das tabelas, como nome da tabela e de atributos, e também a relação de atributos da tabela com outras tabelas.

Com essas informações, o terceiro passo cria registros em memória que guardam as relações entre tabelas, seus nomes, os atributos que compõem cada tabela, os tipos de cada um desses atributos e quais outras características eles têm, por exemplo, se é chave primária ou estrangeira. Ao final é feita uma avaliação de todas as tabelas, que por fim gera os comandos *Rails* adequados.

A IDEIA AGORA É COLOCAR UM EXEMPLO DE USO UTLIZANDO O PAJE

Como funciona, exemplos, aplicações, desempenho, possíveis problemas (**todos os tipos de relacionamento funcionam? 1- 1, 1-n, n-n? etc**)

5 CONCLUSÃO