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**

**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 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.

**2.1.1 Analisador LALR**

O analisador LALR (Look-Ahead Left-to-Right parser) foi inventado por Frank DeRemer para a sua dissertação de doutorado titulada *Practical Translators for LR(k) languages* em 1969 como uma forma simplificada do analisador regular LR.

O objetivo dessa analisador era facilitar a implementação de analisadores LR(1) sem que perder o seu poder. DeRemer conseguiu mostrar em seu trabalho que o seu analisador era mais potente que o LR(0), mas que haviam linguagens que pertenciam a LR(1) e que não pertenciam a LALR. Mais tarde foi demostrado que, mesmo sendo mais fraco que o LR(1), o seu analisador era potente o suficiente para muitas das principais linguagens de programação. (não esquecer citação aqui).

Os primeiros algoritmos para gerar analisadores LALR foram feitos em 1972 (citação), hoje em dia os geradores mais comuns são o YACC e o GNU Bison.

* + 1. **YACC**

YACC (Yet Another Compiler Compiler) é um gerador de analisadores sintáticos LALR, 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 dois pontos, dois identificadores quaisquer 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 em nossa estrutura.

**FAZER A IMAGEM DE EXEMPLO.**

**1.3 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 que utiliza 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 parte superior temos a expressão que vai reconhecer qualquer sequência de caracteres que comece com uma letra (maiúscula ou minúscula) e o restante de qualquer letra ou número. Na parte inferior temos o código que deve ser executado quando a expressão for reconhecida, nesse caso só imprime um texto no terminal.

**FAZER IMAGEM 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 séria de *tokens,* eles são utilizados em conjunto, da forma que o Lex serve como um pré-processador do 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.

+-------+

Source -> | Lex | -> yylex

+-------+

+-------+

Input -> | yylex | -> Output

+-------+

3

lexical grammar

rules rules

| |

v v

+---------+ +---------+

| Lex | | Yacc |

+---------+ +---------+

| |

v v

+---------+ +---------+

Input -> | yylex | -> | yyparse | -> Parsed input

+---------+ +---------+

4

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

**3 SEM NOME**

**Conceitual;**

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

**4 IMPLEMENTAÇÃO**

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

**5 CONCLUSÃO**