UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA PARA DEPURAÇÃO DE FUNÇÕES PL/PGSQL

FABIANO BENDER

BLUMENAU
2012

2012/2-10

FABIANO BENDER

FERRAMENTA PARA DEPURAÇÃO DE FUNÇÕES PL/PGSQL

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Profa. Joyce Martins, Mestre – Orientadora

BLUMENAU
2012

2012/2-10

FERRAMENTA PARA DEPURAÇÃO DE FUNÇÕES PL/PGSQL

Por

FABIANO BENDER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Profa. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Alexander Roberto Valdameri, Mestre – FURB
Membro:	Prof. José Roque Voltolini da Silva – FURB

Blumenau, 24 de janeiro de 2013

Dedico este trabalho a minha esposa Cristiane que me ajudou e me incentivou a sempre continuar.

AGRADECIMENTOS

Primeiramente a Deus, pelo seu imenso amor e graça.

À minha família que sempre esteve presente me apoiando.

Aos meus amigos, Bruno, Fernando e Gilson, que me ajudaram nas dificuldades que enfrentei durante o desenvolvimento deste trabalho.

A minha orientadora, Joyce Martins, por ter acreditado em mim e na conclusão deste trabalho.

Se você tem uma maçã e eu tenho outra; e nós trocamos as maçãs, então cada um terá sua maçã. Mas se você tem uma idéia e eu tenho outra, e nós as trocamos: então cada um terá duas idéias.

George Bernard Shaw

RESUMO

Este trabalho apresenta uma ferramenta para depuração de funções PL/PgSQL, cujo principal objetivo é mostrar o valor das variáveis em cada linha de execução da função. As funções são analisadas léxica, sintática e semanticamente para obtenção da lista de variáveis e de comandos. Estas duas listas são utilizadas pelo depurador para determinar o valor das variáveis. A ferramenta foi desenvolvida na linguagem Python utilizando a biblioteca PyQt em conjunto com a ferramenta Qt Designer para criação da interface gráfica, bem como o PLY para implementação dos analisadores léxico e sintático.

Palavras-chave: Sistemas gerenciadores de banco de dados. PostgreSQL. Compilador. Depurador.

ABSTRACT

This paperwork presents a debugger tool for PL/PgSQL functions, wich the main goal is to be able to show variable values on each executed line of the function. The functions goes through lexical analysis, syntatic analysis and semantic analysis to obtain the variables and commands list. The debugger uses those two lists to determine the variables values. The debugger tool was developed using Python and the library PyQt along with Qt Designer for the graphical interface and also PLY for the implementation of the syntatic and lexical analyser.

Key-words: Database managment system. PostgresSQL. Compiler. Debugger.

LISTA DE ILUSTRAÇÕES

Figura 1 – Principais funções e componentes de um SGBD 17
Quadro 1 – Exemplo da estrutura de uma função PL/PgSQL em PostgreSQL 21
Figura 2 – As fases de um compilador 23
Quadro 2 – Exemplo de utilização de expressões regulares na linguagem Python 24
Quadro 3 – Gramática para expressões aritméticas simples 25
Figura 3 – Depuração na ferramenta PL SQL Developer 29
Figura 4 – Edição e depuração na ferramenta SQL Navigator 30
Figura 5 – Edição e depuração na ferramenta SQL Maestro 31
Figura 6 – Protótipo de um compilador para PL/SQL 32
Quadro 4 – Tipos de dados e descrição 34
Quadro 5 – Funções internas do SGBD PostgreSQL 35
Quadro 6 – Regra sintática do comando `SELECT` do SGBD PostgreSQL 36
Figura 7 – Diagrama de casos de uso 36
Quadro 7 – Caso de uso: `Parametrizar banco de dados` 37
Quadro 8 – Caso de uso: `Criar nova função PL/PgSQL` 37
Quadro 9 – Caso de uso: `Abrir função PL/PgSQL` 37
Quadro 10 – Caso de uso: `Manter função PL/PgSQL` 38
Quadro 11 – Caso de uso: `Salvar função PL/PgSQL` 38
Quadro 12 – Caso de uso: `Compilar` 39
Quadro 13 – Caso de uso: `Depurar` 39
Figura 8 – Diagrama de classe 40
Figura 9 – Diagrama de estados 43
Quadro 14 – Exemplo de *Hello World* em Python 45
Figura 10 – Tela da ferramenta Debugres criada no Qt Designer 46
Quadro 15 – Trecho do código gerado pelo Qt Designer para a tela da Figura 10 47
Quadro 16 – Especificação léxica 48
Quadro 17 – Especificação sintática 49
Quadro 18 – Código fonte do depurador 50
Figura 11 – Interface da ferramenta Debugres 52
Figura 12 – Configurando o arquivo `conexao.txt` 53
Figura 13 – Localizando e selecionando uma função em arquivo 53

Figura 14 – Localizando e selecionando funções no banco de dados 54
Figura 15 – Mensagem: `Deseja salvar a função no banco de dados?` 54
Figura 16 – Mensagem: `Função salva com sucesso!` 54
Figura 17 – Selecionado a pasta e informando o nome da função para `Salvar Como` 55
Figura 18 – Mensagem: `Compilado com sucesso!!!` 55
Figura 19 – Listando os tipos dos parâmetros de entrada da função 56
Figura 20 – Depuração com detecção de erro 57
Figura 21 – Depuração sem detecção de erro 57
Quadro 19 – Comparativo entre os trabalhos correlatos e a ferramenta Debugres 59
Quadro 20 – Métodos do analisador léxico 64
Quadro 21 – Métodos do analisador sintático 71

LISTA DE SIGLAS

ACID – Atomicidade, Consistência, Isolamento e Durabilidade

ANSI – *American National Standards Institute*

API – *Application Programming Interface*

ASP – *Active Server Pages*

BD – Banco de Dados

BNF – *Backus Naur Form*

BSD – *Berkeley Software Distribution*

DB2 – DataBase2

DBA – *Data Base Administrator*

DDL – *Data Definition Language*

DML – *Data Manipulation Language*

EA – Enterprise Architect

GPL – *General Public License*

GUI – *Graphical User Interface*

IBM - *International Business Machines*

IDE – *Integrated Development Environment*

ISO – *International Standard Organization*

LEX – *LEXical analyzer generator*

PHP – *Hypertext Preprocessor*

PL – *Procedural Language*

PLY – Python *Lex-Yacc*

RF – Requisitos Funcionais

RNF – Requisitos Não-Funcionais

SGBD – Sistemas Gerenciadores de Banco de Dados

SQL – *Structure Query Language*

SQL/DS – SQL/*Data System*

SSL – *Secure Sockets Layer*

UML – *Unified Modeling Language*

XML –*eXtensible Markup Language*

YACC – *Yet Another Compiler-Compiler*

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS DO TRABALHO 15
1.2 ESTRUTURA DO TRABALHO 15
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 SISTEMAS GERENCIADORES DE BANCO DE DADOS 16
2.1.1 SGBD PostgreSQL 18
2.1.2 Linguagem SQL 19
2.1.3 Linguagem procedural 19
2.1.4 PL/PgSQL 20
2.2 COMPILADORES E DEPURADORES 21
2.2.1 Análise léxica 24
2.2.2 Análise sintática 25
2.2.3 Análise semântica 27
2.3 TRABALHOS CORRELATOS 28
2.3.1 PL SQL Developer 28
2.3.2 SQL Navigator 29
2.3.3 PostgreSQL Maestro 30
2.3.4 Protótipo de um compilador para a linguagem PL/SQL 31
3 DESENVOLVIMENTO DA FERRAMENTA 33
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 33
3.2 ESPECIFICAÇÃO 34
3.2.1 Definição da gramática da linguagem PL/PgSQL 34
3.2.2 Diagrama de casos de uso 36
3.2.3 Diagrama de classes 39
3.2.4 Diagrama de estado 42
3.3 IMPLEMENTAÇÃO 43
3.3.1 Técnicas e ferramentas utilizadas 44
3.3.1.1 Python 44
3.3.1.2 PyQt 45
3.3.1.3 Qt Designer 46
3.3.1.4 PLY 47

3.3.2 Analisador léxico 48
3.3.3 Analisador sintático 49
3.3.4 Depurador 50
3.3.5 Operacionalidade da implementação 52
3.4 RESULTADOS E DISCUSSÃO 58
4 CONCLUSÕES 60
4.1 EXTENSÕES 61
REFERÊNCIAS BIBLIOGRÁFICAS 62
APÊNDICE A – Código fonte do analisador léxico 64
APÊNDICE B – Código fonte do analisador sintático 71

1 INTRODUÇÃO

Louden (2004, p. 2) afirma que a partir do final da década de 40 tornou-se necessário escrever sequências de código para que os computadores efetuassem as computações desejadas. Com o passar do tempo, os programas foram ficando maiores e mais complexos, surgindo a necessidade de usar Sistemas Gerenciadores de Banco de Dados (SGBD). Conforme expõe Milani (2008, p. 328), o SGBD acrescenta ao banco de dados do tipo arquivo uma nova camada que é responsável por garantir a integridade dos dados, além da segurança e da concorrência no acesso aos dados.

Para efetuar o armazenamento e a manutenção dos dados, os SGBDs fazem uso de linguagens. Ao utilizar uma linguagem procedural (*Procedural Language* - PL), o SGBD não está sendo utilizado apenas para armazenamento dos dados. Segundo Oliveira (2007, p. 574), utilizando PL pode-se agrupar sequencialmente uma série de comandos, dentro de uma única função que estará armazenada dentro do SGBD e que poderá ser executada a qualquer momento, executando todos os comandos contidos dentro de si. Desta forma, junta-se a funcionalidade da linguagem PL com a facilidade de uso da linguagem *Structure Query Language* (SQL), assim economizando tempo, porque diminui a quantidade de vezes que a máquina que fez a requisição e o servidor de dados irão se comunicar, aumentando consideravelmente o desempenho do sistema, pois o SGBD irá desviar-se menos para esta tarefa e mais para o processamento da requisição efetuada.

A facilidade do uso de PL com SQL gerou um problema: não é fácil detectar erros nos programas desenvolvidos em PL, principalmente pela falta de ferramentas de depuração. Pode-se afirmar que a depuração de programas é um trabalho árduo nos diversos SGBDs que existem, ainda mais no SGBD PostgreSQL, uma vez que o mesmo dispõe somente de uma ferramenta de depuração, que além de ser paga, possui suporte apenas para a versão 8.3.

Neste sentido é proposto o desenvolvimento de uma ferramenta capaz de depurar funções implementadas em PL/PgSQL[1], independentemente da versão do SGBD PostgreSQL. A ferramenta deverá compilar as funções PL, detectando e diagnosticando erros léxicos, sintáticos e semânticos. Também deverá ser capaz de executar as funções passo a passo (comando a comando), mostrando o valor das variáveis declaradas.

---

[1] Linguagem procedural do SGBD PostgreSQL.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta para depuração de funções PL/PgSQL.

Os objetivos específicos do trabalho são:

a) efetuar as análises léxica, sintática e semântica das funções, para detectar erros de compilação, bem como extrair as informações necessárias para a depuração;
b) depurar algumas funções internas disponíveis no banco de dados;
c) disponibilizar uma interface para mostrar os valores das variáveis da função depurada.

1.2 ESTRUTURA DO TRABALHO

Este trabalho foi dividido em quatro capítulos, sendo que neste capítulo tem-se a introdução e os objetivos do mesmo. No capítulo seguinte é apresentada a fundamentação teórica, onde é explanado sobre SGBD e suas linguagens, assim como sobre o uso e as partes de um compilador e de um depurador. O capítulo é finalizado com a descrição dos trabalhos correlatos.

O terceiro capítulo relata o desenvolvimento do aplicativo proposto, iniciando com os requisitos, seguidos da especificação, da implementação e da operacionalidade do mesmo, finalizando com os resultados obtidos.

O quarto e último capítulo traz as conclusões do desenvolvimento deste trabalho e possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

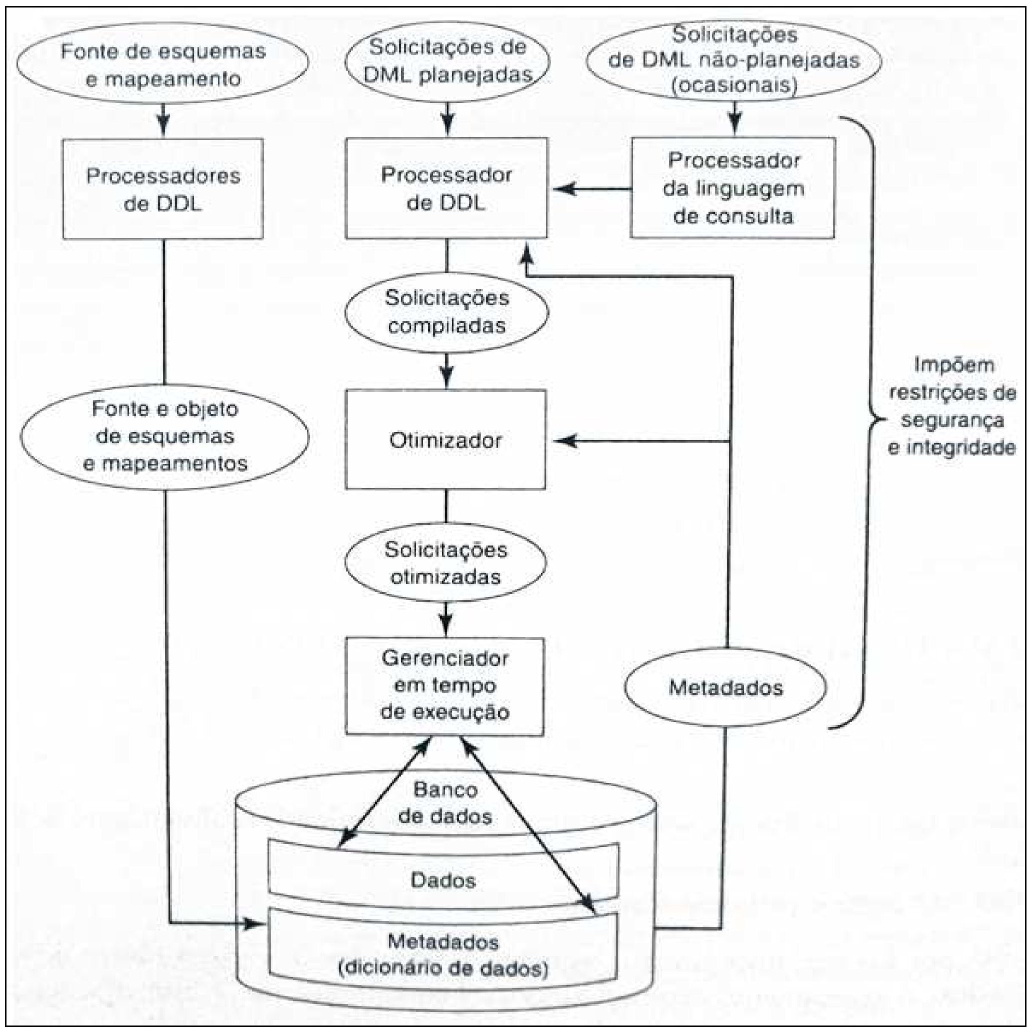
Na seção 2.1 é dada uma visão geral sobre SGBD, sobre o SGBD PostgreSQL e sobre as linguagens utilizadas nos SGBDs. Na seção 2.2 é apresentado o funcionamento de um compilador e de um depurador. Por fim, na seção 2.3 são descritos os trabalhos correlatos.

2.1 SISTEMAS GERENCIADORES DE BANCO DE DADOS

Antes de apresentar a definição de SGBD, deve-se definir o termo Banco de Dados (BD). Segundo Date (2003, p. 10), um banco de dados é uma coleção de dados persistentes, usada por uma aplicação. Um banco de dados é utilizado para armazenar qualquer tipo de dado, desde dígitos e caracteres, até uma sequência de bits e imagens.

Isso posto, segundo Milani (2008, p. 233), o SGBD é um sistema responsável pela segurança e proteção dos dados armazenados no banco de dados de aplicações. Assim, uma das facilidades de se utilizar um SGBD é que as aplicações não precisam se preocupar com a segurança e o acesso aos dados, já que isso fica por conta do próprio SGBD. Conforme Ramakrishnan e Gehrke (2008, p. 7), as vantagens do uso de um SGBD vão além. Pode-se citar: independência dos dados, acesso eficiente aos dados, integridade e segurança dos dados, administração de dados, acesso concorrente, recuperação de falhas e tempo reduzido de desenvolvimento do aplicativo. Mas, “dadas todas essas vantagens, há alguma razão para não se utilizar um SGBD?” (RAMAKRISHNAN; GEHRKE, 2008, p. 8). Devido ao fato de ser complexo e otimizado para alguns tipos de processos, o SGBD pode não ser interessante para algumas aplicações específicas, como, por exemplo, aquelas que dependam de resposta em tempo real.

No entanto, independente da aplicação, Date (2003, p. 38) afirma que um SGBD deve possuir e dar suporte, no mínimo, às funções listadas na Figura 1.



Fonte: Date (2003, p. 38).

Figura 1 – Principais funções e componentes de um SGBD

O SGBD deve ser capaz de aceitar definições de dados e convertê-las para o formato apropriado. Essa é uma tarefa para os processadores de *Data Definition Language* (DDL). Conforme Milani (2008, p. 325), a DDL tem com objetivo efetuar criações e alterações nos formatos de campos e tabelas em um banco de dados. Com o seu uso é possível criar relacionamentos entre entidades e também definir domínio de dados para os atributos.

O SGBD deve possui um processador de *Data Manipulation Language* (DML) para lidar com as requisições do usuário e para buscar, atualizar, excluir ou incluir dados no banco de dados. As requisições de DML devem ser processadas pelo otimizador, que tem por intuito determinar um modo eficiente para executá-las.

O SGBD tem a responsabilidade de fornecer um dicionário de dados. O dicionário de dados contém dados sobre os dados, ou seja, um metadados que possui as definições de todas as estruturas de segurança ou de armazenagem dos dados disponíveis no banco de dados.

O SGBD tem a incumbência de garantir a segurança e a integridade dos dados, monitorando as requisições do usuário e rejeitando toda tentativa de violar as restrições de

segurança e integridade definidas pelo *Data Base Administrator* (DBA).

Atualmente existem muitos sistemas gerenciadores de banco de dados disponíveis no mercado. Entre eles, destaca-se o PostgreSQL.

2.1.1 SGBD PostgreSQL

Entre os vários SGBDs existentes, conforme cita Milani (2008, p. 28), o PostgreSQL destaca-se no mercado por vários motivos, entre os quais: licença de uso *Berkeley Software Distribution* (BSD), que garante uso irrestrito para qualquer finalidade; várias bibliotecas com suporte às principais plataformas e linguagens de programação, como C/C++, PHP[2], ASP[3] e Python; suporte a operações ACID[4], replicação, clusterização, *multithreads* e segurança *Secure Sockets Layer* (SSL), entre vários outros benefícios.

O PostgreSQL é derivado do pacote POSTGRES escrito na Universidade da Califórnia em Berkeley. O projeto POSTGRES iniciou em 1986, sendo que a primeira versão tornou-se operacional em 1987. Porém a mesma foi liberada para alguns poucos usuários externos à universidade em 1989. A segunda versão foi liberada em 1990 devido a uma série de críticas ao sistema de regras do SGBD. Em 1991 foi liberada a terceira versão adicionando suporte a múltiplos gerenciadores de armazenamento, um executor de comandos melhorado e um sistema de regras reescrito. Este mesmo código foi vendido para a empresa Illustra Information Technologies, a qual se fundiu com a Informix, hoje pertencente a IBM, e utilizado como SGBD em um importante projeto científico. A partir da terceira versão até a versão Postgres95 o foco principal estava na portabilidade e na confiabilidade.

Conforme Milani (2008, p. 26), a primeira grande alteração no projeto POSTGRES ocorreu em 1994. Devido à crescente popularidade que a ferramenta estava adquirindo, foi iniciado o projeto Postgres95, trazendo consigo uma grande vantagem em sua primeira versão, a inclusão da linguagem SQL pelos desenvolvedores Andrew Yu e Jolly Chen, substituindo a linguagem PostQUEL utilizada anteriormente. Nesta versão a ferramenta foi totalmente compatibilizada com o padrão ANSI C, tornando-a portável para mais de uma plataforma. Em 1996 surge o nome PostgreSQL, apologia ao POSTGRES original de Berkeley e às versões mais recentes com suporte a linguagem SQL. Com o advento do

---

[2] Acrônimo para *Hypertext Preprocessor*.
[3] Acrônimo de *Active Server Pages.*
[4] Acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade.

PostgreSQL, a ênfase do desenvolvimento foi voltada para a criação de novas funcionalidades e recursos. Com mais de uma década de desenvolvimento, atualmente o PostgreSQL é o mais avançado sistema gerenciador de banco de dados de código aberto disponível (OLIVEIRA, 2007, p. 15).

2.1.2 Linguagem SQL

SQL é a linguagem responsável pela interação com os dados armazenados na maioria dos bancos de dados relacionais (MILANI, 2008, p. 57). Foi desenvolvida originalmente na IBM no início da década de 70 para um protótipo denominado System R e reimplementada em diversos outros produtos comerciais, como o DataBase2 (DB2) e o SQL/*Data System* (SQL/DS). Segundo Manzano (2002, p. 17), com o peso da IBM por trás, a linguagem de consulta estruturada SQL tornou-se o padrão de mercado para linguagem de banco de dados e foi padronizada pelo *American National Standards Institute* (ANSI) em 1982.

A linguagem SQL separa-se basicamente em comandos para definição de dados e comandos para manipulação de dados. Os comandos para definição de dados criam estruturas que serão utilizadas para armazenar os dados e criam a definição das estruturas dos SGBDs. Como exemplo de comandos tem-se `create table` e `create type`.

Os comandos para manipulação de dados, como o próprio termo expressa, servem para manipular os dados, sendo quatro:

a) `select`: utilizado para recuperar e retornar os dados inseridos em uma estrutura do banco de dados;
b) `insert`: utilizado para armazenar os dados em uma estrutura no banco de dados;
c) `update`: utilizado para alterar os dados em uma estrutura no banco de dados;
d) `delete`: utilizado para remover os dados de uma estrutura no banco de dados.

2.1.3 Linguagem procedural

No SGBD PostgreSQL, como em outros SGBDs, é possível que o usuário escreva funções em outras linguagens, as linguagens do tipo procedural. Conforme Oliveira (2007, p. 593), quando o usuário escreve uma função em uma linguagem procedural, o servidor do

banco de dados não possui conhecimento interno sobre como interpretar o texto da função. Em vez disso, a tarefa é passada para um tratador especial embutido no SGBD que conhece os detalhes da linguagem PL. O tratador pode fazer todo o trabalho de analisar e executar a função, ou pode servir como um ligação entre o PostgreSQL e a implementação existente de uma linguagem de programação.

Além de executar comandos de manipulação de dados, nas linguagens PL é possível:

a) criar variáveis e constantes;
b) criar cursores para tratar o resultado do comando `select`;
c) criar estruturas de dados para armazenar o resultado de um cursor ou campos de uma tabela;
d) tratar erros;
e) utilizar comandos de repetição (`for`, `while`, `loop`);
f) utilizar comandos de comparação (`if` e suas variantes).

Segundo Py (2007), é uma boa prática a utilização de PL por programadores, pois, por exemplo, podem deixar armazenados no banco de dados códigos complexos que são utilizados por vários aplicativos, evitando a necessidade de replicá-los.

2.1.4 PL/PgSQL

> PL/PgSQL é uma linguagem procedural carregável desenvolvida para o sistema de banco de dados PostgreSQL. Os objetivos de projeto da linguagem PL/PgSQL foram no sentido de criar uma linguagem procedural carregável que pudesse: ser utilizada para criar procedimentos de funções e de gatilhos; adicionar estruturas de controle à linguagem SQL; realizar processamentos complexos; herdar todos os tipos de dado, funções e operadores definidos pelo usuário; ser definida como confiável pelo servidor; ser fácil de utilizar. (OLIVEIRA, 2007, p. 573).

No PostgreSQL a linguagem padrão para os comandos é a linguagem SQL. Porém, a linguagem SQL possui uma limitação: é executado um comando por vez no banco de dados. Utilizando a linguagem PL/PgSQL, pode-se agrupar vários comandos SQL dentro de uma única função. Além disso, a linguagem PL permite programar funções complexas, iguais as desenvolvidas nas demais linguagens de programação de quarta geração (HIEBERT, 2003, p. 11). A linguagem PL/PgSQL possui uma gama muito grande de tipos de dados, comandos e estruturas de dados. A PL/PgSQL aceita desde os tipo primitivos, como `integer`, `float`, `text` e `boolean`, até tipos complexos tais como `record`, `anyelement`, `macaddr` e `polygon`. Como as demais linguagens de quarta geração, encontram-se na linguagem PL/PgSQL laços

(`for`, `while`, `loop`) e comandos de seleção (`if` e suas variantes). Além disso, PL/PgSQL possui comandos para tratamento de exceções, entre outros.

Uma função em PL/PgSQL (Quadro 1) é formada por um cabeçalho, um escopo e um rodapé.

```
--Cabeçalho da função
CREATE OR REPLACE FUNCTION nome_da_funcao (nome_do_parametro_de_entrada
TIPO_DE_DADO_DO_PARAMETRO) RETURNS TIPO_DE_DADO_DE_RETORNO AS
$$

--Escopo principal da função
DECLARE
--Declaração de variáveis utilizadas no escopo principal
    nome_da_variavel1 TIPO_DE_DADO;
    nome_da_variavel2 TIPO_DE_DADO;
BEGIN
    --Código da função
    --...
    RETURN conteudo_do_tipo_de_dado_de_retorno; --opcional
END;

--Rodapé da função
$$
LANGUAGE plpgsql;
```

Quadro 1 – Exemplo da estrutura de uma função PL/PgSQL em PostgreSQL

No cabeçalho identifica-se o nome da função, os parâmetros de entrada e o tipo do retorno. O escopo é a parte principal de uma função PL, onde são declaradas as variáveis e escritos os comandos que serão executados pela função. No rodapé é especificada a linguagem de programação utilizada na função. De acordo com Barros Neto (2012), no SGBD PostgreSQL, normalmente é utilizada a linguagem PL/PgSQL por ser nativa do SGBD, mas pode-se programar em diversas linguagens como: C, Perl, Java, Python, entre outras.

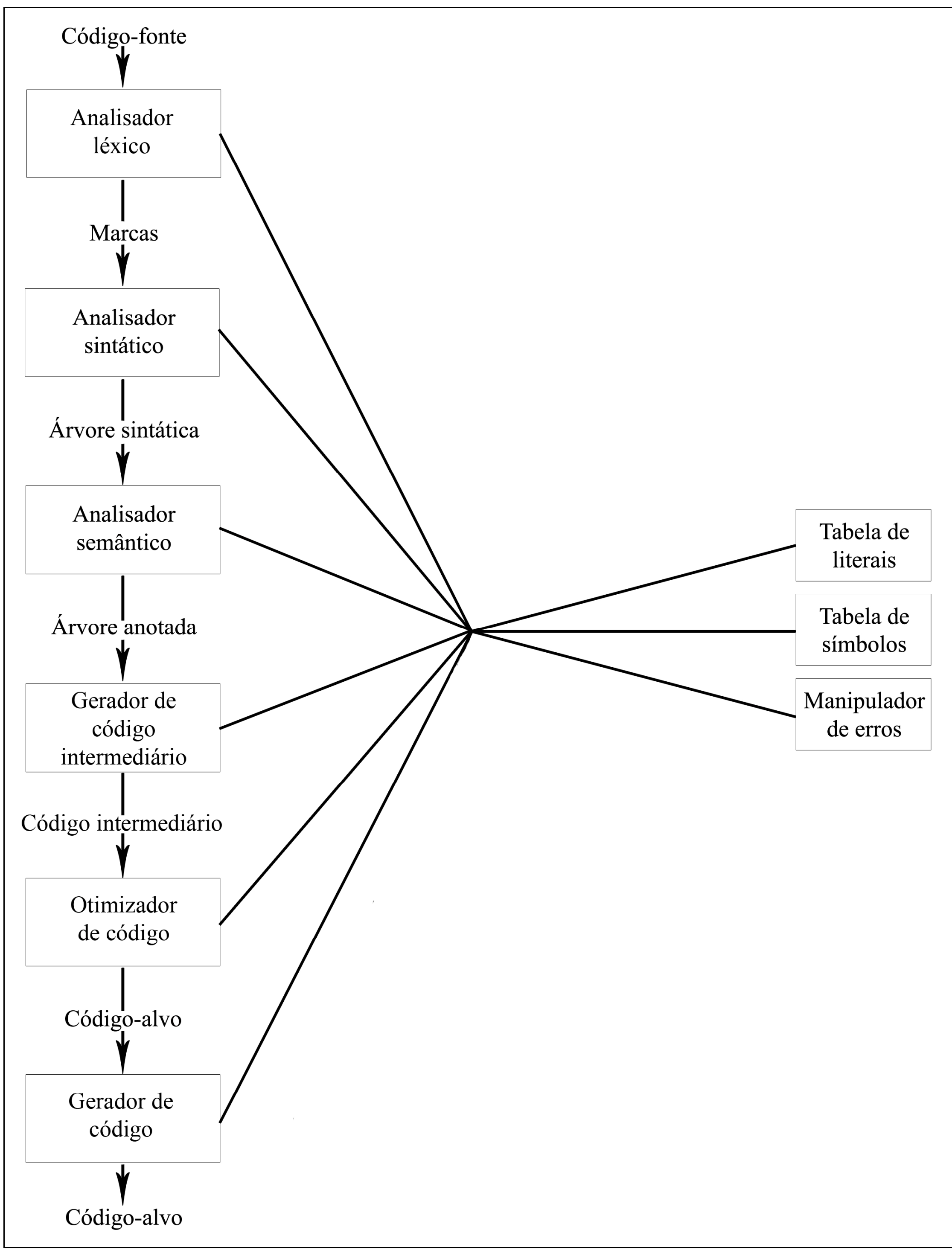
2.2 COMPILADORES E DEPURADORES

No final da década de 40, com o surgimento do computador de John von Neumann, surgiu a necessidade de criar sequências de códigos para que este computador as executasse. No começo, estes códigos eram escritos em linguagem de máquina, com códigos numéricos representando as operações que a máquina efetivamente iria executar. Como enfatiza Louden (2004, p. 2), escrever programas desta forma consumia muito tempo e era um trabalho entediante. Rapidamente esta linguagem foi substituída pela linguagem de montagem (*Assembly*), onde as instruções são codificadas de forma simbólica e posteriormente traduzidas para instruções de linguagem de máquina. Ainda segundo Louden (2004, p. 2), as

linguagens de montagem fizeram com que os programas fossem implementados com maior velocidade e com maior precisão. Ainda hoje as linguagens de montagem são utilizadas, principalmente quanto é necessário uma grande velocidade ou concisão no código. O próximo passo para a evolução foi a criação de uma linguagem mais próxima da linguagem natural ou da notação matemática e de um compilador para a mesma.

Conforme Delamaro (2004, p. 2), a principal característica de um compilador é “transformar um programa escrito numa linguagem de alto nível – que chamamos de linguagem fonte – em instruções executáveis por uma determina máquina – que chamamos de código objeto”. Em geral um compilador recebe como entrada um fonte de um sistema e o transforma em um código que pode ser executado por uma máquina.

De acordo com Price e Toscani (2001, p. 7), um compilador é um programa que pode tornar-se muito complexo, por isso é dividido em partes menores que estão interconectadas entre si. As partes ou fases de um compilador, mostradas na Figura 2, são: analisador léxico, analisador sintático, analisador semântico, gerador de código intermediário, otimizador de código e gerador de código.



Fonte: adaptado de Louden (2004, p. 7).

Figura 2 – As fases de um compilador

Para construção de um depurador devem ser desenvolvidos os três analisadores de um compilador. A diferença entre um depurador e um compilador é que o depurador tem como objetivo reduzir erros em um programa, isso é, erros que impossibilitam o programa de ser executado e erros que produzem um resultado inesperado na execução do mesmo. Segundo Louden (2004, p. 6), um depurador é um programa que pode ser utilizado para verificar a existência de erros de execução em um programa compilado.

2.2.1 Análise léxica

A primeira fase para a construção de um compilador é a análise léxica. De acordo com Price e Toscani (2001, p. 17), a principal função do analisador léxico, também denominado *scanner*, é “fazer a leitura do programa fonte, caractere a caractere, e traduzi-lo para uma sequência de símbolos léxicos, também chamados de *tokens*”.

Há diversas categorias de *tokens* como palavras reservadas, identificadores, símbolos especiais, constantes, operadores da linguagem, entre outras. Cada *token* é representado internamente no analisador léxico por três informações: classe (categoria do *token* reconhecido), valor (sequência de caracteres que compõem o *token*) e posição (linha e coluna onde o *token* se encontra).

A especificação dos *tokens* é feita através de expressões regulares. Conforme Grune et al. (2001, p. 54), “uma expressão regular é uma fórmula que descreve um conjunto de strings possivelmente infinito”. A expressão regular mais básica é aquela que define apenas um caractere. Além deste padrão, existem mais dois padrões básicos de expressões regulares, uma que define um conjunto de caracteres e um que define todos os caracteres possíveis de se utilizar na linguagem (Quadro 2).

```
#biblioteca para operações com expressões regulares
>>> import re
>>> token = "Fabiano 123 Bender"

#pesquisa pelo caractere 'a' na variável token
>>> re.findall('a', token)
['a', 'a']

#pesquisa por uma sequência de dígitos na variável token
>>> re.findall('\d+', token)
['123']

#pesquisa por uma sequência de letras na variável token
>>> re.findall('[a-zA-Z\_]+', token)
['Fabiano', 'Bender']
```

Quadro 2 – Exemplo de utilização de expressões regulares na linguagem Python

A implementação de analisadores léxicos é feita através da implementação de autômatos finitos correspondentes às expressões regulares especificadas. Esta implementação pode ser gerada de forma automática através de um gerador de analisadores léxicos, mais conhecido como *LEXical analyzer generator* (LEX), ou através da construção da tabela de transição correspondente em uma linguagem de programação. Em ambos os casos, a saída será uma lista de *tokens* que será utilizada pelo analisador sintático.

2.2.2 Análise sintática

Na estrutura de um compilador, o analisador sintático está interligado com o analisador léxico recebendo do mesmo uma lista de *tokens* que representa o programa fonte. O analisador sintático verifica se estes *tokens* estão de acordo com a sequência determinada pela gramática.

Normalmente, as estruturas sintáticas válidas são especificadas através de uma gramática livre de contexto. De acordo com Price e Toscani (2001, p. 30), as gramáticas livres de contexto, popularizadas pela notação *Backus Naur Form* (BNF), formam a base para a análise sintática, pois permitem descrever a maioria das linguagens de programação utilizadas. No Quadro 3 é apresentada uma gramática livre de contexto na notação BNF para expressões aritméticas simples.

```
<expressão>  ::= <expressão> + <termo>
               | <expressão> - <termo>
               | <termo>
<termo>      ::= <termo> * <fator>
               | <termo> / <fator>
               | <fator>
<fator>      ::= ( <expressão> )
               | identificador
```

Fonte: adaptado de Aho et al. (2008, p. 126).

Quadro 3 – Gramática para expressões aritméticas simples

Existem duas estratégias básicas para realizar a análise sintática: descendente (*top-down*) e ascendente (*bottom-up*).

Os métodos de análise baseados na estratégia descendente constroem a árvore de análise sintática analisando a lista de *tokens* e substituindo os nós internos da árvore (símbolos não terminais) pelo lado direito das regras de produção (derivação). Segundo Louden (2004, p. 143), o nome descendente vem da forma como a árvore sintática é percorrida, em pré-ordem, ou seja, a partir do símbolo inicial da gramática (raiz da árvore), fazendo a árvore crescer até atingir as suas folhas. Há dois tipos de analisadores sintáticos descendentes: analisadores com retrocesso e analisadores preditivos. Um analisador sintático com retrocesso testa diferentes possibilidades para os *tokens* da entrada, retrocedendo se alguma possibilidade falhar. Já um analisador sintático preditivo tenta prever a derivação seguinte com base em um ou mais *tokens* da entrada.

Os métodos de análise baseados na estratégia ascendente constroem a árvore de análise sintática analisando a lista de *tokens* e substituindo os nós da árvore na forma inversa à estratégia descendente. A partir dos *tokens* (folhas da árvore sintática) constrói a árvore até o

símbolo inicial da gramática (raiz da árvore de derivação). Isso é, em cada passo de execução, o lado direito das regras de produção é substituído por um símbolo não-terminal (redução).

De acordo com Price e Toscani (2001, p. 54), os analisadores sintáticos normalmente são implementados por autômatos de pilha. A implementação é feita de forma automática utilizando um gerador de analisadores sintáticos. Um gerador de analisadores sintáticos é um programa que recebe como entrada uma especificação de sintaxe de uma determinada linguagem e produz como saída um programa para análise sintática da linguagem especificada. Um gerador de analisador sintático amplamente utilizado é chamado *Yet Another Compiler-Compiler* (YACC). O YACC recebe como entrada um arquivo de especificação e produz como saída um arquivo para o analisador sintático. Um arquivo de especificação YACC é divido em três seções: definições, regras e rotinas auxiliares. A seção de definições contém informações sobre os *tokens*, tipos de dados e as regras gramaticais para que o YACC gere o analisador sintático. A seção de regras contém as regras gramaticais junto com ações semânticas codificadas na linguagem que está sendo utilizada para implementar o compilador. A seção de rotinas auxiliares contém declarações de rotinas que podem ser necessárias para completar o analisador sintático e o compilador.

Por fim, de acordo com Aho et al. (2008, p. 122), o analisador sintático deve ser projetado para emitir mensagens para quaisquer erros de sintaxe encontrados no programa fonte e também para se recuperar destes erros, a fim de continuar processando o restante do programa.

> Quando um compilador detecta um erro de sintaxe, é desejável que ele tente continuar o processo de análise de modo a detectar outros erros que possam existir no código ainda não analisado. Isso envolve realizar o que se chama de recuperação ou reparação de erros. (PRICE; TOSCANI, 2001, p. 74).

Segundo Aho et al. (2008, p. 124), a técnica mais simples é interromper a análise sintática e emitir uma mensagem logo após localizar o primeiro erro. Para a recuperação de erros de sintaxe são utilizadas quatro estratégias (AHO et al., 2008, p. 125):

a) modo pânico: ao encontrar um erro o analisador sintático vai descartando os símbolos de entrada até que um conjunto de *tokens* de sincronismo seja encontrado;
b) nível de frase: ao encontrar um erro o analisador sintático pode realizar a correção local sobre o restante da entrada, utilizando um símbolo que permita a continuação da análise;
c) produções de erro: estende-se a gramática da linguagem com produções que geram construções sintáticas erradas, antecipando os erros mais comuns que poderiam

ocorrer na análise do programa;

d) correção global: determina uma sequência mínima de mudanças a fim de obter uma correção com custo global menor.

2.2.3 Análise semântica

Na construção de um compilador o analisador sintático trabalha conjuntamente com o analisador semântico, cuja principal atividade é determinar se a árvore de sintaxe resultante da análise sintática faz sentido, ou seja, verificar se um identificador declarado como variável é usado como tal, se existe compatibilidade entre operandos e operadores, entre muitas outras validações (PRICE; TOSCANI, 2001, p. 9).

De acordo com Aho et al. (2008, p. 6), uma parte importante da análise semântica é a verificação de tipo, em que o compilador verifica se cada operador possui operandos compatíveis. Se durante a análise semântica for identificado algum erro de compatibilidade de tipos, o analisador deve retornar uma mensagem de erro.

A análise semântica pode ser dividida em duas categorias, conforme Louden (2004, p. 259). A primeira é análise do texto de entrada verificando se o mesmo atende as regras da linguagem de programação, para verificar sua correção e garantir a sua execução. A segunda categoria da análise semântica é efetuar a melhora da eficiência de execução do programa traduzido.

Diferentemente da análise sintática que pode ser especificada através de uma gramática livre de contexto, na análise semântica não há nenhum método padrão para especificação. Um método para escrever um analisador semântico é identificar atributos ou propriedades de entidades da linguagem alvo que precisem ser examinadas e escrever regras semânticas que expressem como a computação de atributos e propriedades se relacionam com as regras gramaticais da linguagem.

Outra diferença da análise semântica para as análises léxica e sintática, é que ambas possuem uma forma de automatizar o desenvolvimento dos analisadores. Para a análise léxica tem-se o LEX e para a análise sintática tem-se o YACC. Na análise semântica não há nenhum artifício porque normalmente a situação não é tão clara e em parte por não existir uma forma padrão para a especificação da semântica como as expressões regulares para a análise léxica e a BNF para a análise sintática.

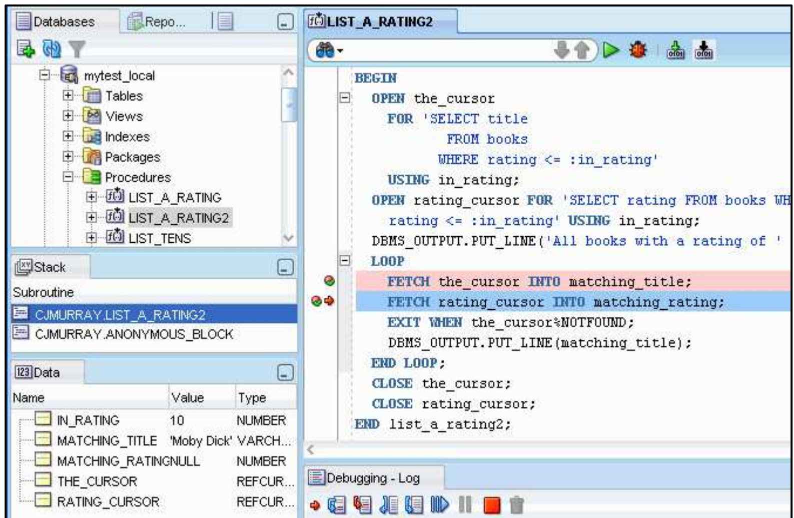
2.3 TRABALHOS CORRELATOS

Algumas ferramentas desempenham papel semelhante ao proposto no presente trabalho, cada qual com as suas peculiaridades e para determinados SGBDs. Dentre elas, pode-se citar: PL SQL Developer (REZENDE, 2004), SQL Navigator (QUEST SOFTWARE, 2011), PostgreSQL Maestro (SQL MAESTRO GROUP, 2010) e o protótipo de um compilador desenvolvido por Hiebert (2003).

2.3.1 PL SQL Developer

Segundo Rezende (2004), desenvolver aplicações ou executar administração em um banco de dados Oracle não é uma tarefa simples, pois requer um alto grau de conhecimento do banco de dados, bem como da linguagem procedural. O PL SQL Developer é uma *Integrated Development Environment* (IDE) para desenvolvimento de *functions*, *procedures* e *triggers* armazenados em um banco de dados Oracle. O PL SQL Developer oferece uma interface limpa e extremamente produtiva para as tarefas de edição, compilação, correção, testes, depuração e otimização, além de outras funcionalidades como execução de *scripts* SQL, criação e modificação de definições de tabelas e relatórios.

O PL SQL Developer é uma ferramenta que permite um alto grau de produtividade e confiabilidade nas informações e resultados. Ela inclui opções para edição de funções PL, possui funcionalidade para se construir *querys* de forma dinâmica e possui um depurador integrado que permite rastrear erros nas unidades de programa. O depurador permite a utilização de *breakpoints*, *step in* e *step out*, bem como a visualização do valor das variáveis das funções. A Figura 3 ilustra uma função sendo depurada com a ferramenta PL SQL Developer.



Fonte: Sun Software (2011).

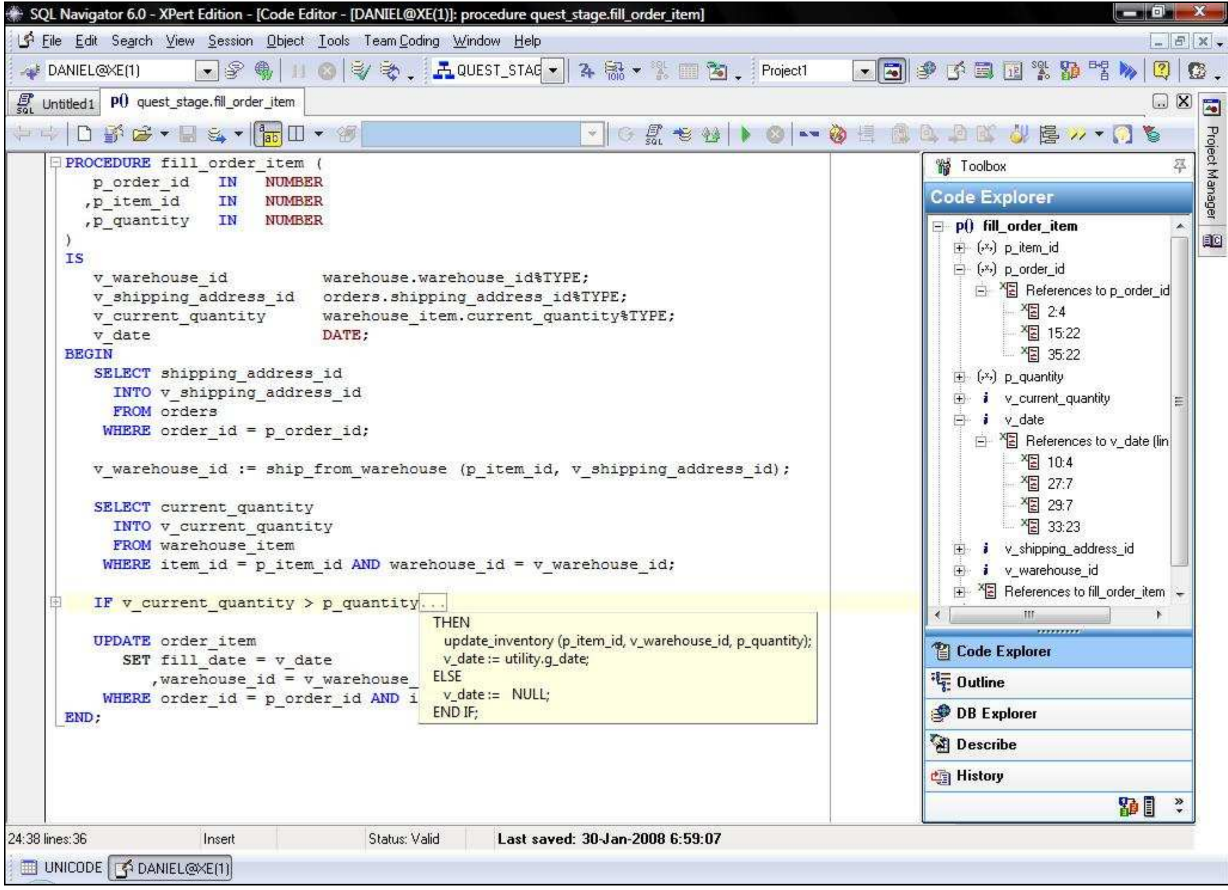
Figura 3 – Depuração na ferramenta PL SQL Developer

2.3.2 SQL Navigator

Segundo a Quest Software (2011), o SQL Navigator é uma solução completa para desenvolvimento e gerenciamento para o banco de dados Oracle. Ela proporciona um ambiente integrado para desenvolver e testar *procedures*, esquemas, *scripts* SQL e programas, a partir de uma interface gráfica de fácil utilização. Possui bibliotecas de codificação integradas que possibilitam um desenvolvimento rápido e livre de erros.

O SQL Navigator está divido em vários módulos, sendo um desses o PL SQL Debugger. O PL SQL Debugger ajuda a depurar *procedures*, *triggers* e tipos de dados. O SQL Navigator conta praticamente com os mesmos recursos citados na ferramenta anterior, porém sua principal diferença é que o depurador permite usar o método de apontar e clicar para identificar e corrigir problemas lógicos em funções PL armazenadas no banco de dados. Além disso, com o SQL Navigator pode-se: definir pontos de interrupção, executar o código PL SQL passo a passo ou até um ponto de interrupção especificado e avaliar ou modificar os valores das variáveis do programa durante a execução da depuração.

Conforme a Quest Software (2011), os desenvolvedores podem interativamente codificar e depurar o código linha por linha, assim visualizando como o código será executado pelo servidor em tempo real. A Figura 4 ilustra uma função sendo depurada na ferramenta SQL Navigator através do módulo PL SQL Debugger.



Fonte: Quest Software (2011).

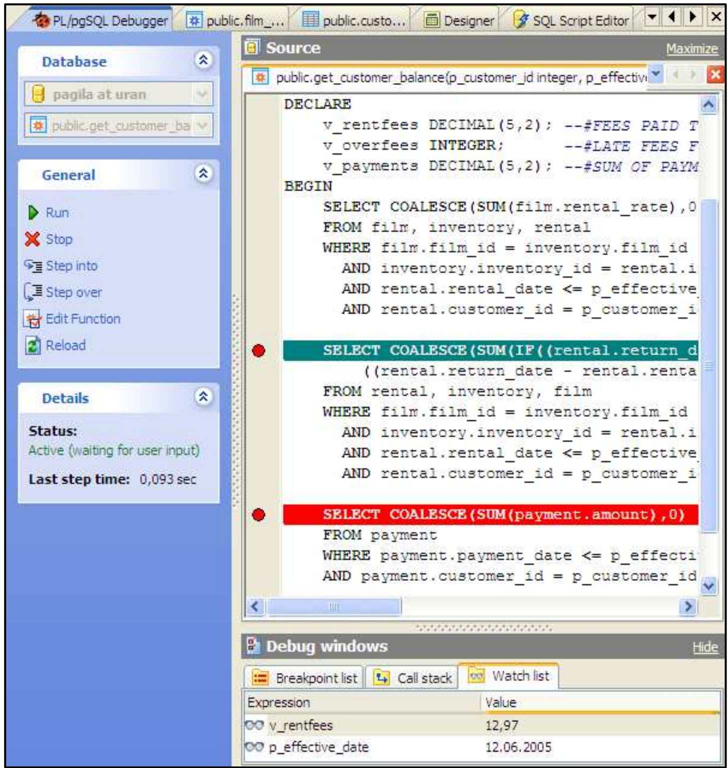
Figura 4 – Edição e depuração na ferramenta SQL Navigator

2.3.3 PostgreSQL Maestro

Conforme o SQL Maestro Group (2010, p. 1), o “PostgreSQL Maestro é a principal ferramenta de administração para o desenvolvimento e gestão do PostgreSQL. Ele permite que você faça todas as operações do banco de dados de forma fácil e rápida”. O depurador do PostgreSQL Maestro permite depurar funções escritas em PL/PgSQL usando recursos de depuração tradicionais, como a utilização de *breakpoints*, a visualização de valores de variáveis e a verificação da pilha de chamadas. O processo de depuração dentro do depurador do PostgreSQL Maestro é semelhante ao das demais ferramentas desta mesma linha.

A janela do depurador consiste em três partes (Figura 5): *Source*, *Debug windows* e barra de navegação. A guia *Source* contém editores de *script* para cada unidade de programa, permitindo alternar facilmente entre elas para ver o fonte da função PL, acrescentar ou remover pontos de interrupção e assim por diante. Quando ocorre uma exceção, o depurador mostra a linha que gerou a exceção e os valores das variáveis no momento da exceção. Isso é

útil para descobrir a causa da exceção. A *Debug windows* mostra características do processo de depuração, como informações dos *tokens* nos pontos de interrupção e uma pilha de execução com as chamadas de métodos que o depurador usou para chegar até a linha atual.

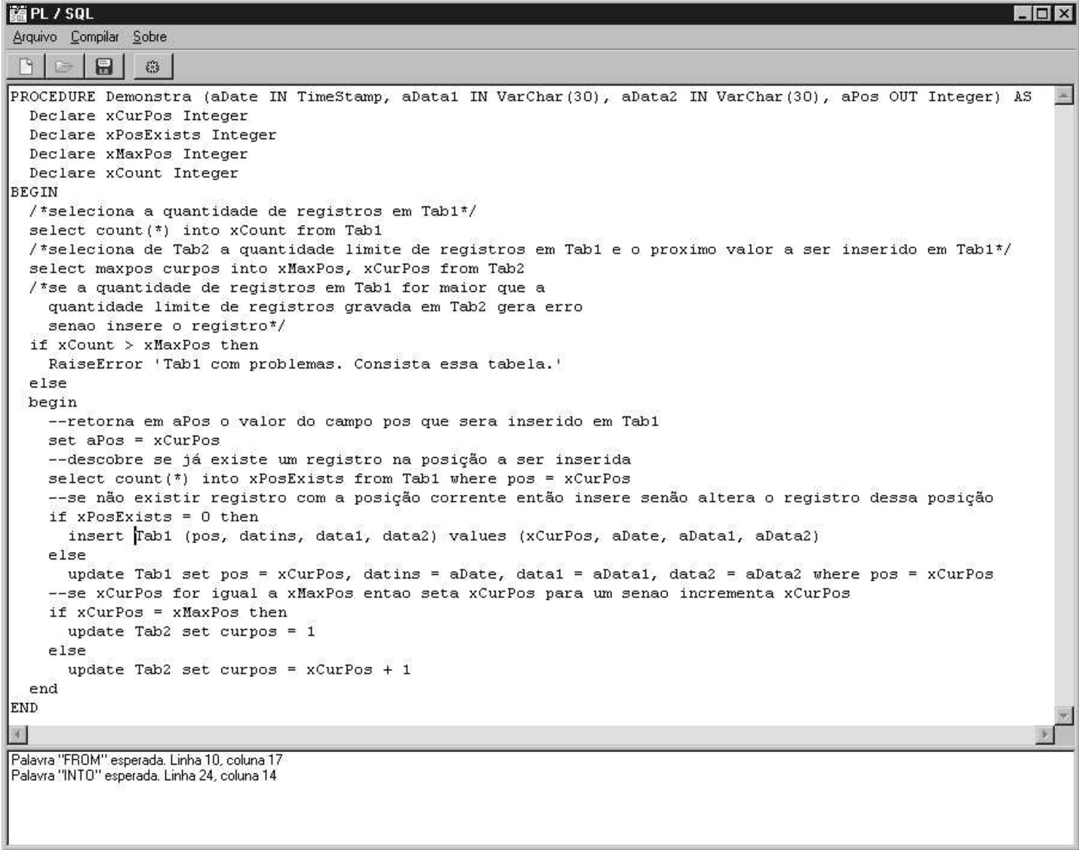


Fonte: SQL Maestro Group (2010, p. 309).

Figura 5 – Edição e depuração na ferramenta SQL Maestro

2.3.4 Protótipo de um compilador para a linguagem PL/SQL

Segundo Hiebert (2003, p. 4), para o desenvolvimento do protótipo foi definido um padrão segundo as normas definidas pela *International Standard Organization* (ISO) para um determinado conjunto de comandos da PL, a partir do qual foi especificada uma gramática e implementado um compilador capaz de realizar checagem de erros e gerar código específico para os SGBDs Oracle e SQL Server. A ferramenta possui a interface apresentada na Figura 6.



Fonte: Hiebert (2003, p. 41).

Figura 6 – Protótipo de um compilador para PL/SQL

Com este protótipo, o usuário pode editar e salvar funções PL ou abrir arquivos com extensão SQL contendo uma função. Ao compilar uma função, caso a mesma possua algum erro, o protótipo mostra os erros detectados. Caso a função compilada não possua erros, é apresentada uma tela para que o usuário escolha o SGBD para o qual deseja gerar código.

3 DESENVOLVIMENTO DA FERRAMENTA

Neste capítulo é descrito o desenvolvimento da ferramenta proposta bem como sua utilização, abordando os seguintes itens:

a) a especificação dos requisitos;
b) a especificação da ferramenta, apresentando a gramática da linguagem PL/PgSQL, os diagramas de casos de uso, de classes e de estado;
c) a implementação da ferramenta.

Ainda são relatadas as dificuldades encontradas e os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF) da ferramenta proposta são:

a) compilar as funções PL/PgSQL (RF);
b) depurar as funções PL/PgSQL (RF);
c) permitir conectar em um banco de dados PostgreSQL (RF);
d) possuir um editor para criar e manter funções PL/PgSQL (RF);
e) permitir salvar funções (do banco de dados ou de arquivos texto) em arquivo texto com extensão SQL (RF);
f) permitir salvar funções no banco de dados ou em arquivos texto (RF);
g) executar no sistema operacional Windows e Linux (RNF);
h) ser implementada utilizando a linguagem de programação Python (RNF);
i) utilizar a ferramenta Qt Designer para desenho da interface gráfica (RNF);
j) utilizar a biblioteca PyQt para interligação da interface com a ferramenta (RNF);
k) utilizar a biblioteca PLY para geração dos analisadores léxicos e sintáticos (RNF).

3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação da ferramenta, detalhando a gramática definida para desenvolvimento da mesma, bem como descrevendo os diagramas de casos de uso, de classes e de estado da *Unified Modeling Language* (UML). Os diagramas foram especificados utilizando a ferramenta Enterprise Architect (EA).

3.2.1 Definição da gramática da linguagem PL/PgSQL

Como a linguagem PL/PgSQL possui uma gama muito grande de tipos de dados, de funções internas e de comandos, na ferramenta proposta nem todas as estruturas sintáticas foram implementadas, delimitando a gramática utilizada. Assim, o Quadro 4 traz os tipos de dados suportados pela ferramenta desenvolvida. No Quadro 5 são relacionadas as funções internas do SGBD PostgreSQL suportadas pela ferramenta. Por fim, no Quadro 6 tem-se as regras sintáticas do comando `SELECT`. A gramática encontra-se completamente especificada nos Apêndices A e B.

tipo SQL	**aliases**	**descrição**
`bigint`	`int8`	inteiro de oito bytes com sinal
`bigserial`	`serial8`	inteiro de oito bytes com auto incremento
`boolean`	`bool`	booleano lógico (verdade/falso)
`bytea`		dados binários ("matriz de bytes")
`date`		data de calendário (ano, mês, dia)
`double precision`	`float8`	número de ponto flutuante de precisão dupla
`integer`	`int, int4`	inteiro de quatro bytes com sinal
`interval`		espaço de tempo
`numeric`	`decimal`	numérico exato com precisão selecionável
`serial`	`serial4`	inteiro de quatro bytes com auto incremento
`smallint`	`int2`	inteiro de dois bytes com sinal
`text`		cadeia de caracteres de comprimento variável
`timestamp`	`timestamptz`	data e hora

Fonte: adaptado de Oliveira (2007, p. 83).

Quadro 4 – Tipos de dados e descrição

função	tipo de entrada	tipo de retorno	descrição
upper	text	text	converte as letras do texto em maiúsculas
lower	text	text	converte as letras do texto em minúsculas
coalesce	qualquer tipo de dado	qualquer tipo de dado	retorna o primeiro dos argumentos que não for nulo
Length	text	integer	retorna o número de caracteres da cadeia de caracteres
substr	text, integer, integer	text	retorna uma parte de uma cadeia de caracteres
max	diversos[5]	diversos	retorna o valor máximo de um conjunto de linhas
min	diversos	diversos	retorna o valor mínimo de um conjunto de linhas
avg	diversos	diversos	retorna o valor médio de um conjunto de linhas
round	numeric | double precision, integer	numeric | double precision	retorna o valor arredondado em n casas
trunc	numeric | double precision, integer	numeric | double precision	retorna o valor truncado em n casas
now		timestamp	retorna a data e hora corrente
replace	text, text, text	text	substitui todas as ocorrências no texto, do texto de origem pelo texto de destino
count	diversos	diversos	retorna a quantidade de um conjunto de linhas
r_pad	text, integer, text	text	preenche o texto até o comprimento adicionando os caracteres de preenchimento à direita
l_pad	text, integer, text	text	preenche o texto até o comprimento adicionando os caracteres de preenchimento à esquerda
r_trim	text, text	text	remove a cadeia de caracteres mais longa contendo apenas os caracteres do final do texto
l_trim	text, text	text	remove a cadeia de caracteres mais longa contendo apenas os caracteres do início do texto
trim	text, text	text	remove a cadeia de caracteres mais longa contendo apenas os caracteres da extremidade inicial, final e ambas do texto
to_char	timestamp| interval | integer | double precision| numeric, text	text	converte um tipo em text
current_date		date	retornar a data corrente
current_time		time	retorna a hora corrente
current_timestamp		timestamp	retorna a data e hora corrente

Fonte: adaptado de Oliveira (2007, p. 11, p. 127, p. 130, p. 132, p. 133, p. 167, p. 173, p. 174, p. 196).

Quadro 5 – Funções internas do SGBD PostgreSQL

[5] Nesse caso, os tipos possíveis são: bigint, date, double precision, integer, interval, numeric, smallint, text, time e timestamp.

```
expression_select :
  SELECT expression_coluna INTO ID SEMICOLON
| SELECT expression_coluna INTO ID FROM expression_tabela SEMICOLON
| SELECT expression_coluna INTO ID FROM expression_tabela WHERE expression_where
                                                                    SEMICOLON
| SELECT expression_coluna INTO ID FROM expression_tabela ORDER expression_order
                                                                    SEMICOLON
| SELECT expression_coluna INTO ID FROM expression_tabela WHERE expression_where
                                              ORDER expression_order SEMICOLON
| PERFORM expression_coluna SEMICOLON
| PERFORM expression_coluna FROM expression_tabela SEMICOLON
| PERFORM expression_coluna FROM expression_tabela WHERE expression_where
                                                                    SEMICOLON
| PERFORM expression_coluna FROM expression_tabela ORDER expression_order
                                                                    SEMICOLON
| PERFORM expression_coluna FROM expression_tabela WHERE expression_where
                                              ORDER expression_order SEMICOLON
```

Quadro 6 – Regra sintática do comando `SELECT` do SGBD PostgreSQL

3.2.2 Diagrama de casos de uso

A ferramenta especificada possui sete casos de usos (Figura 7): `Parametrizar banco de dados`, `Criar nova função PL/PgSQL`, `Abrir função PL/PgSQL`, `Manter função PL/PgSQL`, `Salvar função PL/PgSQL`, `Compilar` e `Depurar`.

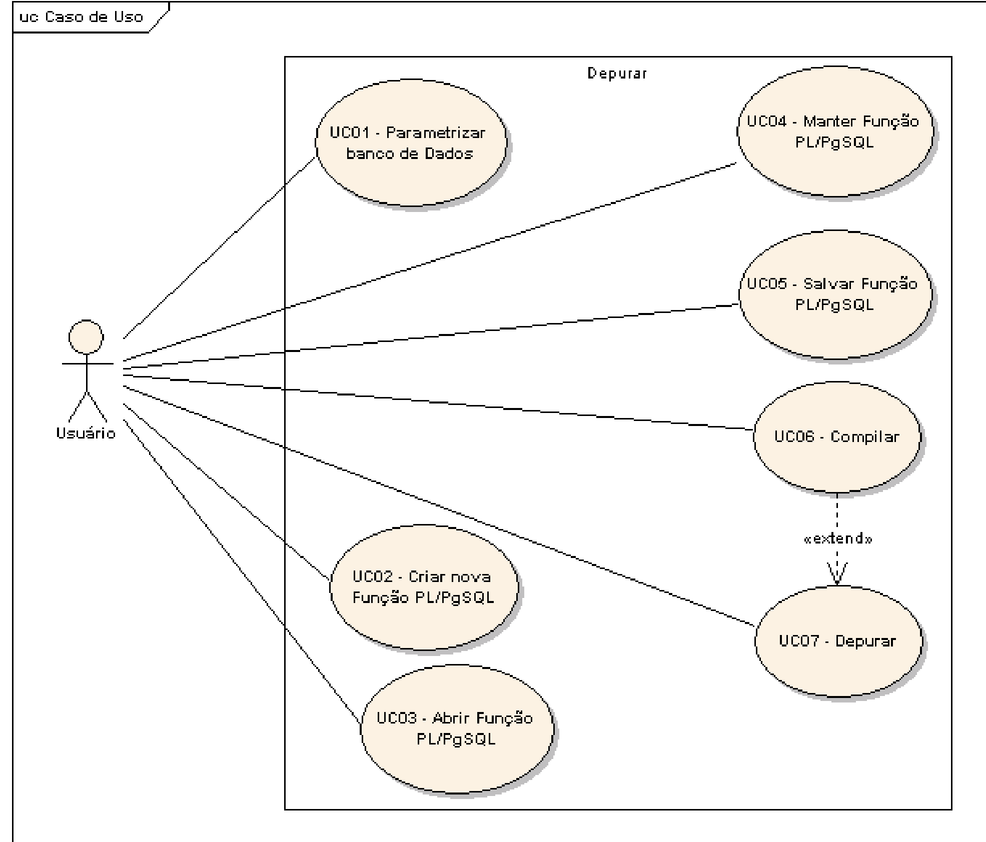


Figura 7 – Diagrama de casos de uso

O caso de uso `Parametrizar banco de dados`, apresentado no Quadro 7, é a ação

inicial do usuário da ferramenta.

UC01 - Parametrizar banco de dados	
Descrição	Permite parametrizar o banco de dados que será utilizado.
Atores	Usuário
Pré-condições	A parametrização deve ser feita em um arquivo chamado `conexao.txt`.
Cenário principal	1. O usuário inclui uma linha que identifica o *host* do servidor onde se encontra o banco de dados (por exemplo: `host=localhost`). 2. O usuário inclui uma linha que identifica o nome do banco de dados (por exemplo: `database=nome_da_bd`). 3. O usuário inclui uma linha que identifica o usuário de acesso ao banco de dados (por exemplo: `user=usuario_do_bd`). 4. O usuário inclui uma linha com a senha de acesso ao banco de dados (por exemplo: `password=senha_do_bd`). 5. O usuário salva o arquivo `conexao.txt` no mesmo diretório em que encontra-se o executável da ferramenta.
Pós-condições	As quatro informações (`host`, `database`, `user`, `password`) devem estar preenchidas para poder acessar a ferramenta.

Quadro 7 – Caso de uso: `Parametrizar banco de dados`

Após o usuário parametrizar o banco de dados, a ferramenta pode ser utilizada. São seis funcionalidades, apresentadas nos casos de usos `Criar nova função PL/PgSQL` (Quadro 8), `Abrir função PL/PgSQL` (Quadro 9), `Manter função PL/PgSQL` (Quadro 10), `Salvar função PL/PgSQL` (Quadro 11), `Compilar` (Quadro 12) e `Depurar` (Quadro 13).

UC02 – Criar nova função PL/PgSQL	
Descrição	Permite criar uma nova função.
Atores	Usuário
Pré-condições	Deve ter sido executado o caso de uso UC01.
Cenário principal	1. O usuário clica no botão `Novo`.
Pós-condições	O editor da ferramenta deve ficar em branco e com o cursor posicionado na primeira linha.

Quadro 8 – Caso de uso: `Criar nova função PL/PgSQL`

UC03 – Abrir função PL/PgSQL	
Descrição	Permite abrir uma função salva em um arquivo texto ou no banco de dados.
Atores	Usuário
Pré-condições	Deve ter sido executado o caso de uso UC01. Deve existir pelo menos uma função em um arquivo texto ou no banco de dados.
Cenário principal	Existem duas formas de abrir uma função: 1. Abrir uma função que se encontra em um arquivo texto: 1.1. O usuário clica no botão `Abrir`. 1.2. A ferramenta permite localizar e selecionar uma função em um arquivo. 1.3. O usuário seleciona o diretório e o arquivo desejados e clica no botão `Abrir`. 2. Abrir uma função que se encontra no banco de dados: 2.1. O usuário clica no botão `Abrir BD`. 2.2. A ferramenta lista as funções disponíveis no banco de dados. 2.3. O usuário seleciona a função desejada e clica no botão `OK`.
Pós-condições	A função escolhida deve ser carregada no editor da ferramenta.

Quadro 9 – Caso de uso: `Abrir função PL/PgSQL`

UC04 – Manter função PL/PgSQL	
Descrição	Permite alterar uma função já aberta ou uma nova função.
Atores	Usuário
Pré-condições	Deve ter sido executado o caso de uso UC02 ou o caso de uso UC03.
Cenário principal	1. O usuário implementa uma nova função ou altera uma função existente no editor da ferramenta, utilizando a linguagem PL/PgSQL.
Pós-condições	

Quadro 10 – Caso de uso: `Manter função PL/PgSQL`

UC05 – Salvar função PL/PgSQL	
Descrição	Permite salvar uma função em um arquivo texto ou no banco de dados.
Atores	Usuário
Pré-condições	Deve ter sido executado o caso de uso UC02 ou o caso de uso UC03 ou o caso de uso UC04.
Cenário principal	Existem três formas de salvar uma função: 1. Salvar uma função existente em um arquivo texto: 1.1. O usuário clica no botão `Salvar`. 1.2. A ferramenta exibe uma mensagem questionando se a função deve ser salva no banco de dados ou em um arquivo texto. 1.3. O usuário pressiona a opção correspondente a arquivo texto. 1.4. A ferramenta emite uma mensagem informando que a função foi salva com sucesso, isso é, o arquivo texto foi atualizado. 2. Salvar uma função existente no banco de dados: 2.1. O usuário clica no botão `Salvar`. 2.2. A ferramenta exibe uma mensagem questionando se a função deve ser salva no banco de dados ou em um arquivo texto. 2.3. O usuário pressiona a opção correspondente a banco de dados. 2.4. Caso a função tenha sido carregada do banco de dados, a ferramenta emite uma mensagem informando que a função foi salva com sucesso. 2.5. Caso a função seja nova, a ferramenta emite uma mensagem informando que a função foi inserida com sucesso no banco de dados. 3. Salvar uma função nova ou uma função existente em um arquivo texto: 3.1. O usuário clica no botão `Salvar como` ou no botão `Salvar`, para funções novas, ou clica no botão `Salvar como`, para funções existentes. 3.2. A ferramenta permite selecionar o local e informar o nome da função. 3.3. O usuário seleciona o diretório e informa o nome que deseja para a função e clica no botão `Salvar`. 3.4. A ferramenta emite uma mensagem informando que a função foi salva com sucesso.
Pós-condições	A função deve ser salva em um arquivo texto ou no banco de dados, conforme o caso.

Quadro 11 – Caso de uso: `Salvar função PL/PgSQL`

UC06 – Compilar	
Descrição	Permite verificar se não existe erro de compilação na função que está sendo analisada.
Atores	Usuário
Pré-condições	Deve ter sido executado o caso de uso UC02 ou o caso de uso UC03 ou o caso de uso UC04.
Cenário principal	1. O usuário clicar no botão `Compilar`. 2. A ferramenta executa a função do editor da ferramenta no banco de dados a fim de verificar se está léxica, sintática e semanticamente correta. 3. A ferramenta informa que a função foi compilada com sucesso.
Cenário de exceção	No passo 2, a ferramenta detecta algum erro. Neste caso, é informado para o usuário o erro ocorrido e em qual linha. O usuário deve obrigatoriamente voltar para o caso de uso `UC03 – Manter função PL/PgSQL`.
Pós-condições	A função do editor da ferramenta deve ser compilada com sucesso.

Quadro 12 – Caso de uso: `Compilar`

UC07 – Depurar	
Descrição	Permite analisar semanticamente uma função e retornar a lista de *tokens* e seus valores em cada linha.
Atores	Usuário
Pré-condições	Deve ter sido executado o caso de uso UC02 ou o caso de uso UC03 ou o caso de uso UC04.
Cenário principal	1. O usuário clica no botão `Depurar`. 2. O usuário informa valores para os parâmetros de entrada. 3. O usuário escolhe a(s) variável(eis) que será(ão) analisada(s). 4. A ferramenta executa o caso de uso `UC06 – Compilar`. 5. A ferramenta mapeia todas as variáveis declaradas na função e executa as análises léxica, sintática e semântica, verificando o correto uso das variáveis. 6. A ferramenta lista as variáveis escolhidas pelo usuário no passo 3 e seus valores em cada linha da função até alcançar o final da função.
Cenário de exceção	No passo 4 ou no passo 5, a ferramenta detecta algum erro. Neste caso, o processo de depuração é abortado e é informado para o usuário o erro ocorrido e em qual linha. O usuário deve obrigatoriamente voltar para o caso de uso `UC03 – Manter função PL/PgSQL`.
Pós-condições	A função do editor da ferramenta deve ser depurada com sucesso e os valores das variáveis em cada linha da função devem ser apresentados.

Quadro 13 – Caso de uso: `Depurar`

3.2.3 Diagrama de classes

Na Figura 8 é mostrado o diagrama de classe com as classes da ferramenta, incluindo a forma como estas estão estruturadas e interligadas.

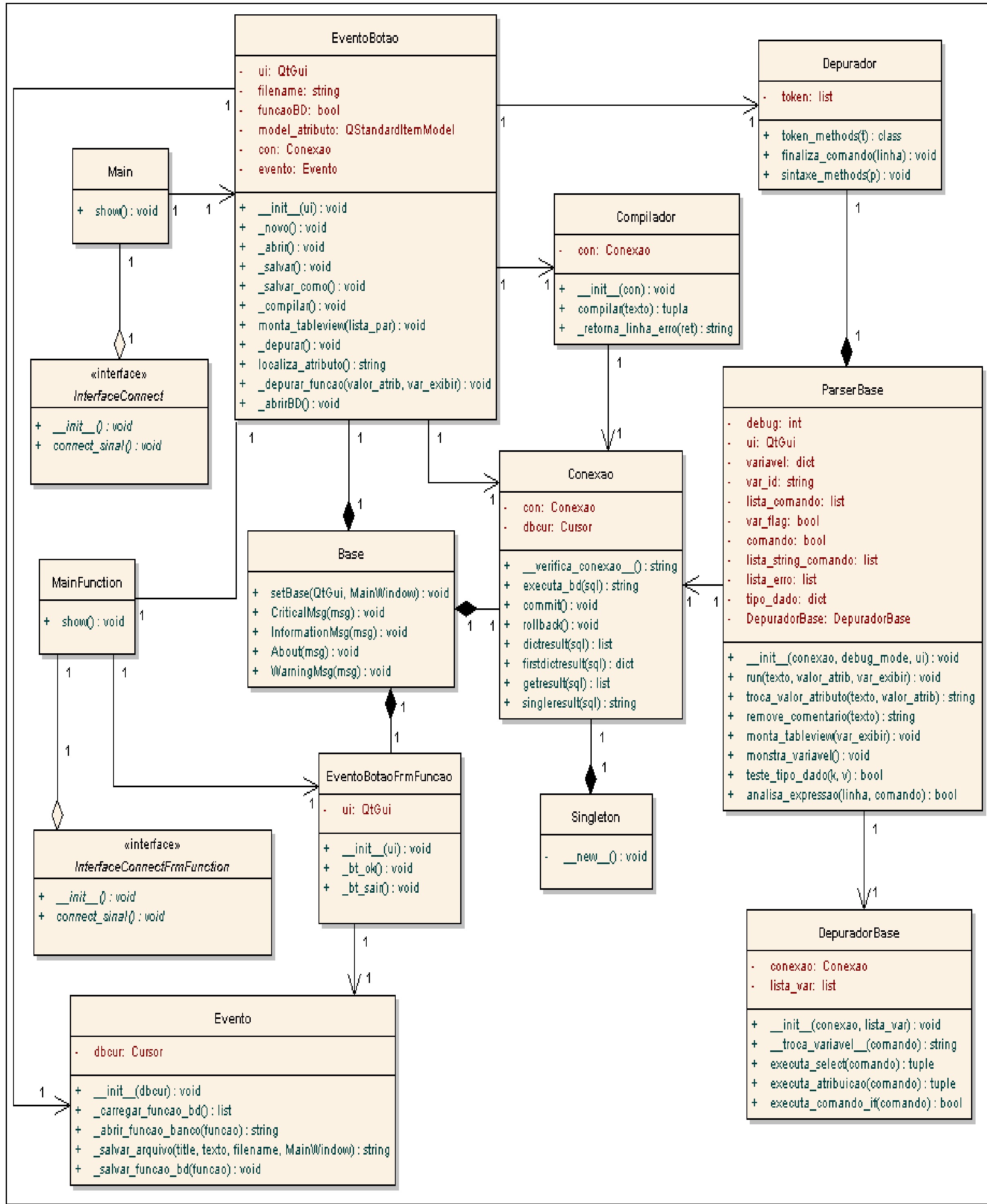


Figura 8 – Diagrama de classe

A classe `Main` abre o formulário principal da ferramenta, ficando também responsável por conectar os eventos dos botões através da interface `InterfaceConnect`.

A responsabilidade pela usabilidade do formulário principal é delegada à classe `EventoBotao`, que possui todos os eventos que estão disponíveis nesta tela, bem como é responsável por interligar as principais classes da ferramenta. O método `_abrirBD` instancia a classe `MainFunction`, que é responsável por recuperar a lista de todas as funções disponíveis

no banco de dados, apresentando-as no formulário secundário da ferramenta, além de retornar para o formulário principal a estrutura da função escolhida pelo usuário. O método `_compilar` é responsável por chamar a classe `Compilador`. Na classe `Compilador` é feita uma primeira análise para verificar qualquer inconsistência que exista na função que está sendo trabalhada. Nesta análise, se for detectado, será retornado para o usuário qualquer erro que impeça a execução da função. O método `_depurar_funcao` chama a principal classe da ferramenta, a classe `Depurador`. Nela são feitas as análises léxica e sintática e separadas as variáveis disponíveis na função. A classe `Depurador` ainda é responsável por retornar os erros léxicos e os erros sintáticos não localizados pela classe `Compilador`.

A classe `EventoBotao` possui uma instância da classe `Conexao`, classe essa que efetua a conexão com o banco de dados a partir da leitura do arquivo `conexão.txt`, além de possuir os métodos para manipulação dos comandos no banco de dados. A classe `Conexao`, por possuir um ponteiro para o banco de dados, é uma classe `Singleton`. Ou seja, qualquer classe que for instanciar a classe `Conexao` receberá sempre a mesma instância desta classe. Foi implementada esta funcionalidade a fim de evitar que sejam criados inúmeros cursores[6] no banco de dados, deixando a ferramenta mais lenta. A classe `EventoBotao` possui também uma instância da classe `Evento`.

A classe `ParserBase` é a classe que determina os valores em cada linha de execução para a lista de variáveis localizadas pela classe `Depurador`, montando uma lista de variáveis e valores, além de fazer a análise semântica do depurador, através do método `analisa_expressao`. No método `testa_tipo_dado` é feita a análise semântica da ferramenta. Neste método é verificado se o valor da variável corresponde ao tipo de dado definido para a mesma. A classe `ParserBase` possui uma instância da classe `DepuradorBase` que é responsável por executar, através dos métodos `executa_select`, `executa_atribuicao` e `executa_comando_if`, cada comando da função extraído durante as analises léxica e sintática.

A classe `MainFunction` possui a responsabilidade de localizar, listar e recuperar todas as funções presentes no banco de dados, além de abrir a tela secundária da ferramenta. A classe `MainFunction` está interligada a classe `EventoBotaoFrmFuncao`. Esta classe possui os eventos dos botões desta tela e uma instância para a classe `Evento`.

Na classe `Evento` o método `_carregar_funcao_bd` busca no banco de dados todas as

[6] Um cursor é o resultado de uma consulta armazenado em memória, sendo possível apenas acessar as informações deste resultado. É uma espécie de "conjunto de dados" onde o armazenamento fica restrito apenas em memória e não no disco rígido (SILVA, 2013).

funções que estão inseridas no banco de dados. O método `_abrir_funcao_banco` possui a responsabilidade de abrir a função escolhida pelo usuário, que se encontra no banco de dados, através da leitura das tabelas bases do banco, remontando a função. Os métodos `_salvar_arquivo` e `_salvar_funcao_bd` irão salvar a função em um arquivo texto ou no banco de dados, respectivamente.

A classe `Depurador` é responsável por fazer as análises léxica e sintática da ferramenta. O método `token_methods` representa todos os métodos que possuem a especificação das expressões regulares (Apêndice A) para a identificação dos *tokens*. Já o método `sintaxe_methods`, por sua vez, representa todos os métodos que possuem a especificação das regras sintáticas (Apêndice B) da linguagem PL/PgSQL.

3.2.4 Diagrama de estado

Na Figura 9 é mostrado o diagrama de estados que é utilizado como um complemento para o diagrama de classes. Neste diagrama é possível verificar o estado em que o objeto se encontra naquele momento e a sequência lógica de utilização da ferramenta desenvolvida.

A ferramenta inicia com o editor sem nenhum texto. A partir deste estado, é possível que o usuário crie uma função nova ou abra uma função existente em um arquivo texto ou no banco de dados. Para abrir uma função a partir do banco de dados, deve-se selecionar uma a partir de uma relação com todas as funções inseridas no banco. Para abrir uma função a partir de um arquivo de texto, deve-se localizar a função desejada no sistema de arquivos.

Após a função estar disponível no editor da ferramenta, a mesma pode ser alterada, salva, compilada ou depurada. Ao compilar uma função, a ferramenta verifica a possível existência de erros léxicos, sintáticos e alguns erros semânticos. Ao depurar uma função, a ferramenta efetua as análises léxica, sintática e semântica e exibe uma lista de variáveis com os seus valores.

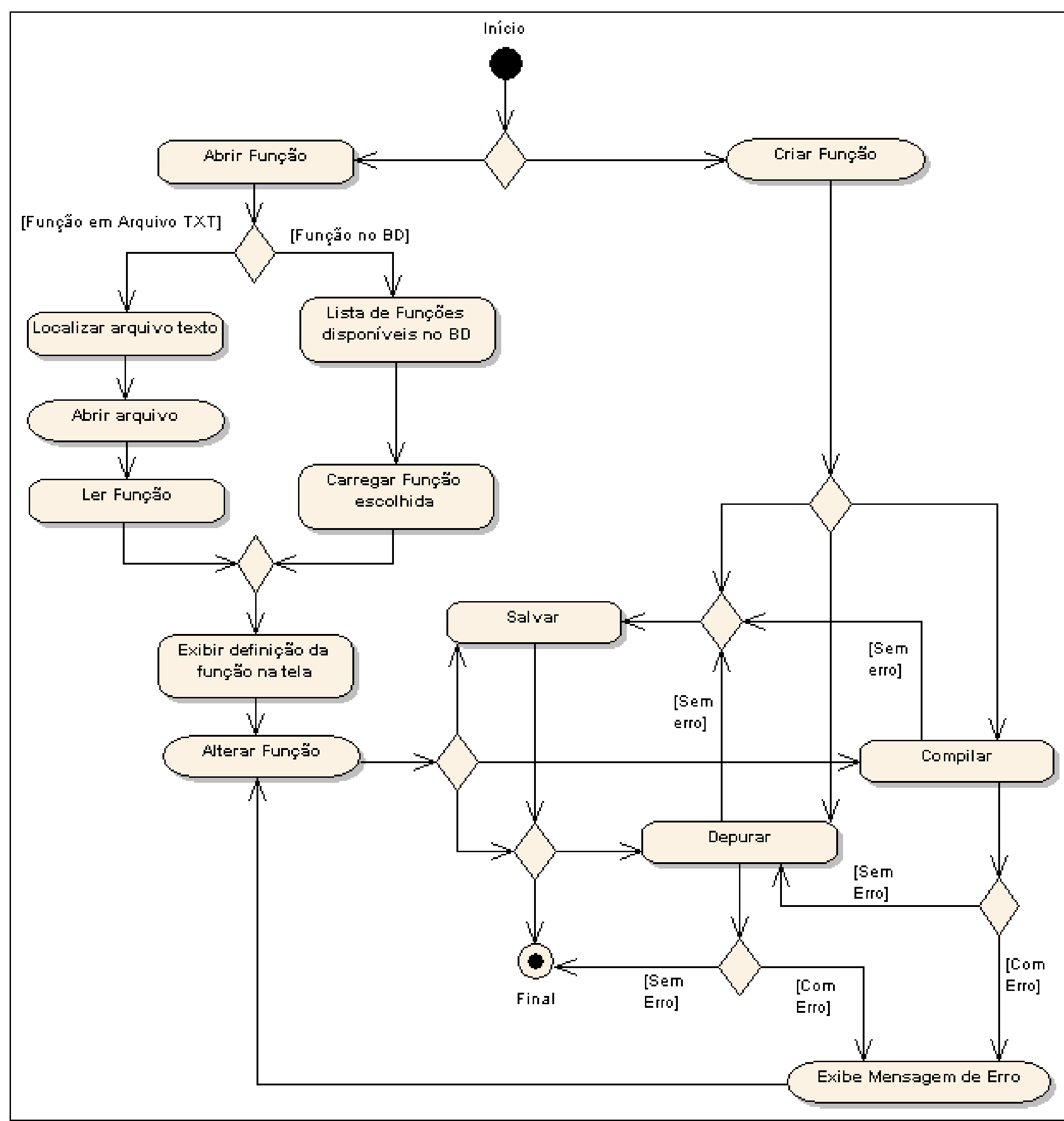


Figura 9 – Diagrama de estados

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Nesta seção são descritas a linguagem, as ferramentas e as bibliotecas utilizadas na implementação. Inicialmente a linguagem Python é brevemente apresentada, seguida da biblioteca PyQt, da ferramenta Qt Designer e, por fim, da biblioteca PLY.

3.3.1.1 Python

Python é uma linguagem de programação que foi desenvolvida por Guido van Rossum no início dos anos 90. A linguagem Python surgiu com o intuito de ser mais poderosa que as linguagens de *shell scripting* e mais alto nível que a linguagem C, para programar sistemas operativos. Utilizando um modelo de desenvolvimento colaborativo, em que qualquer pessoa pode contribuir com ideias e resolução dos problemas, o Python evoluiu até o estado atual.

Conforme Morais e Pires (2002, p. 1), o nome Python não possui um relacionamento com a selvagem, feroz e gigantesca cobra Python, no Brasil retratado pela cobra Anaconda. Na verdade este nome surgiu por Guido van Rossum ser fã incondicional dos comediantes britânicos da década de 70, Monty Python. Morais e Pires (2002, p. 3) destacam como as principais características da linguagem Python:

a) é fácil de utilizar, visto que pessoas com pouco conhecimento técnico conseguem desenvolver programas em Python;
b) incentiva a utilização de metodologias de programação corretas, tornando os programas escritos em Python altamente legíveis e de fácil compreensão, manutenção e extensão;
c) possui vasta biblioteca de objetos e funções, evitando o "reinventar da roda" tão comum em outras linguagens;
d) é orientada a objetos;
e) é interpretada, não sendo necessária a compilação do código-fonte para execução do programa;
f) é multiplataforma, estando disponível na maioria dos sistemas operacionais que possuem como base o Unix, como Linux, MAC OS e Free BSD, assim como para o Windows e plataformas PalmOS.

No Quadro 14 são apresentados dois exemplos de *Hello World* em Python, o primeiro

um *script* e o segundo com orientação a objetos.

```
#Hello World em script
print "Saída: Hello World"
Saída: Hello World

#Hello World utilizando orientação a objetos
class OlaMundo:
    def getOlaMundo(self):
        return "Hello World"

ola = OlaMundo()
print "Saída: %s" % ola.getOlaMundo()

Saída: Hello World
```

Quadro 14 – Exemplo de *Hello World* em Python

3.3.1.2 PyQt

Diferente de algumas ferramentas como o Visual Basic e Delphi, Python não possui uma interface gráfica nativa, mas existem inúmeras bibliotecas GUI[7] disponíveis para o mesmo, entre elas, o PyQt. PyQt é um módulo para Python desenvolvido pela *Computing Riverbank Limited* para a plataforma Qt, que é um *toolkit* desenvolvido pela Nokia Frameworks. O PyQt possui licença *dual*, *General Public License* (GPL) e comercial. Segundo Rempt (2002), PyQt é uma biblioteca muito avançada e oferece um grande conjunto de objetos para desenvolvimento de telas, incluindo: um conjunto substancial de *widgets*, classes para acesso a bancos de dados, *parser eXtensible Markup Language* (XML) e suporte a ActiveX no Windows. Além disso, segundo Werneck (2010), PyQt possui como vantagens:

a) facilidade de aprendizado: a biblioteca possui uma ótima documentação desenvolvida pelo Trolltech;
b) *Application Programming Interface* (API): a API do PyQt utiliza um sistema de sinais, similar aos utilizados por outras bibliotecas;
c) IDE: possui uma IDE boa para criação de interfaces, chamada Qt Designer;
d) portabilidade: os programas rodam com o visual nativo nas plataformas Windows, Unix e Mac OS.

De acordo com Rempt (2002), a combinação de Python e Qt é extremamente eficiente, podendo ser usado em uma ampla variedade de aplicações. É utilizado, por exemplo, para gerar *script* OpenGL; criar modelos 3D complexos; desenvolver aplicações com animação;

[7] Acrônimo de *Graphical User Interface*.

escrever programas de banco de dados, jogos, utilitários e aplicativos de monitoramento de hardware. Python e Qt são usados em projetos de código aberto, mas também por grandes empresas, como a Disney, Globo e Google.

3.3.1.3 Qt Designer

Para utilizar o PyQt pode-se criar todas as telas manualmente ou utilizar o Qt Designer (Figura 10).

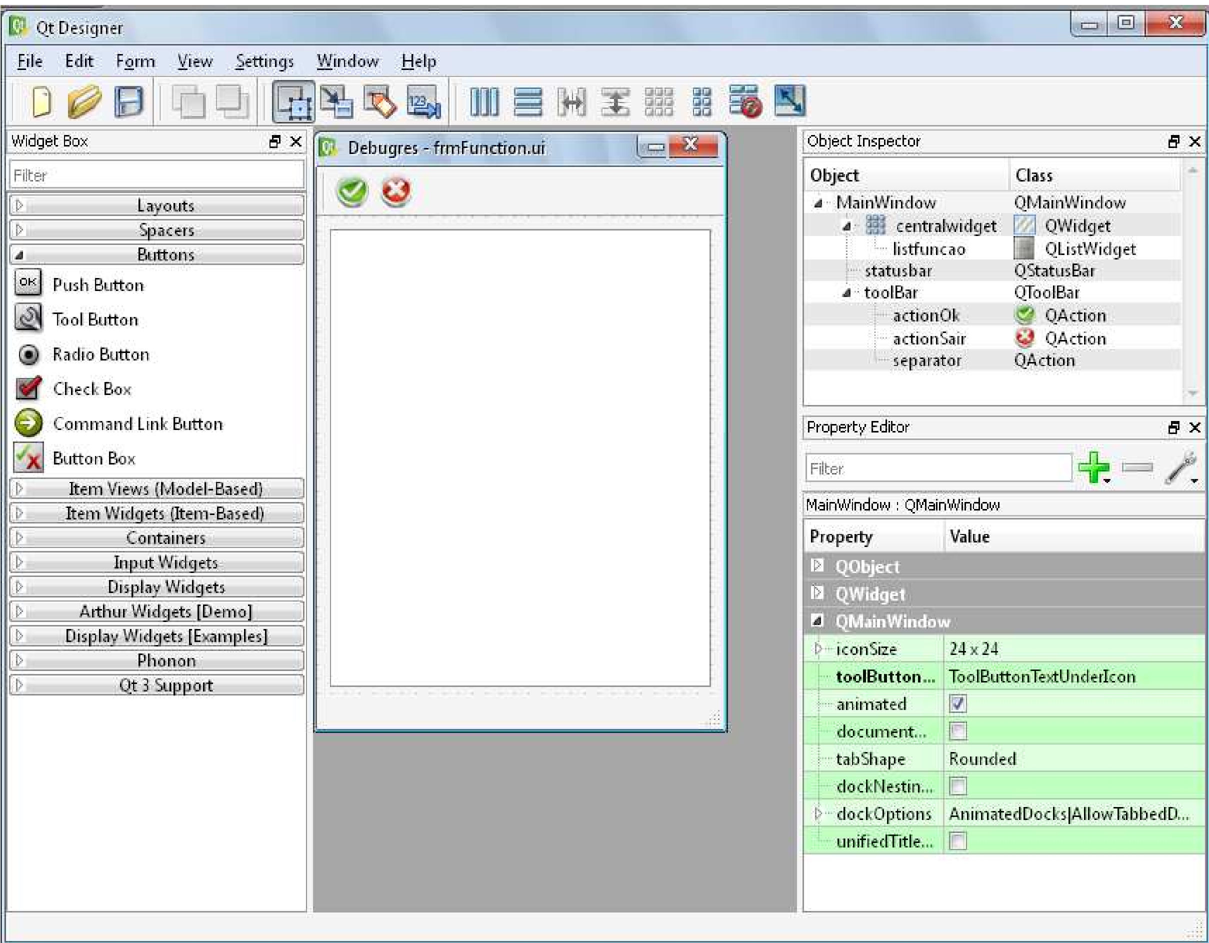


Figura 10 – Tela da ferramenta Debugres criada no Qt Designer

O aplicativo Qt Designer é distribuído juntamente com o *toolkit* Qt, sobre o nome *designer*. Com o Qt Designer é possível clicar em um objeto e arrastar para a tela, bem como conectar sinais aos objetos para utilizá-los no código fonte para associar os eventos aos objetos. De acordo com Rempt (2002), o Qt Designer gera arquivos XML (Quadro 15) que podem ser compilados para Python ou C++.

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>MainWindow</class>
  <widget class="QMainWindow" name="MainWindow">
    <property name="geometry">
      <rect><x>0</x><y>0</y><width>289</width><height>389</height></rect>
    </property>
    <property name="windowTitle">
      <string>Debugres</string>
    </property>
    <property name="windowIcon">
      <iconset><normaloff>icone.ico</normaloff>icone.ico</iconset>
    </property>
    ...
    <action name="actionOk">
      <property name="icon">
        <iconset resource="icons.qrc">
        <normaloff>:/imagens/ok.png</normaloff>:/imagens/ok.png</iconset>
      </property>
      <property name="text"><string/></property>
      <property name="iconText"><string/></property>
      <property name="toolTip"><string>Ok(Ctrl+O)</string></property>
      <property name="shortcut"><string>Ctrl+O</string></property>
    </action>
    ...
    <resources>
      <include location="icons.qrc"/>
    </resources>
    ...
</ui>
```

Quadro 15 – Trecho do código gerado pelo Qt Designer para a tela da Figura 10

Summerfield (2007, p. 206) afirma que o benefício de usar o Qt Designer, além da conveniência de criar a interface visualmente, é que se o projeto for alterado só será necessário recompilar o arquivo resultante do Qt Designer, não sendo necessário alterar o código fonte. Só será necessário mudar o código fonte se for acrescentado, apagado ou renomeado algum objeto que é ou será referenciado. Isso significa que usar o Qt Designer é muito mais rápido e mais fácil do que codificar os *layouts* da interface em "código fonte puro", ajudando a manter a separação entre a concepção visual e o comportamento implementado no código.

3.3.1.4 PLY

PLY é uma implementação das ferramentas LEX e YACC. Foi inteiramente escrito em Python e sua primeira versão foi desenvolvida por Beazley (2011). O módulo LEX é usado para quebrar o texto de entrada em uma coleção de *tokens* especificados por uma coleção de expressões regulares. Já o YACC é usado para reconhecer a sintaxe da linguagem que tenha sido especificada na forma de uma gramática livre de contexto.

Os módulos LEX e YACC foram desenvolvidos para trabalharem em conjunto, onde o

módulo LEX fornece uma interface externa na forma de uma função que retorna o próximo *token* válido no fluxo de entrada do YACC. O módulo YACC fornece vários recursos para tratamento dos *tokens* retornados pelo LEX, incluindo verificação de erros, validação da gramática, apoio a produções vazias, erro nos *tokens* e resolução de ambiguidade através de regras de precedência.

Conforme Beazley (2011), o PLY depende de reflexão para construir os analisadores léxico e sintático. Ao contrário das ferramentas tradicionais que exigem um arquivo de entrada especial que é convertido em um arquivo separado, as especificações dadas para o PLY são retiradas diretamente dos programas Python. Isso significa que não existem arquivos fonte extra, nem há um passo de construção especial do compilador.

3.3.2 Analisador léxico

O primeiro passo para a construção do depurador foi a implementação do analisador léxico. Foram definidos todos os *tokens*, tratados pela ferramenta, presentes na linguagem PL/PgSQL. A definição dos *tokens* foi feita através da especificação de expressões regulares. Com as expressões regulares definidas (Quadro 16, Apêndice A), o módulo LEX da biblioteca PLY se encarrega de extrair e identificar os *tokens* presentes nas funções analisadas.

```
#Método com a expressão regular que identifica se o token é um tipo
def t_TYPE(self, t):
    r'bigint|bigserial|boolean|bytea|date|double\
precision|integer|interval|numeric|serial|smallint|text|timestamp|void|time'
    if self.var_flag:
        self.variavel[self.var_id] = [t.value, None]

    return t

#Método com a expressão regular que identifica se o token é um identificador
def t_ID(self, t):
    r'[a-zA-Z\_][a-zA-Z0-9\_]*'

    if self.var_flag:
        self.var_id = t.value.strip()

    return t

#Método com a expressão regular que identifica se o token é um decimal
def t_DECIMAL(self, t):
    r'\d+(\.\d+)?'
    return t
```

Quadro 16 – Especificação léxica

Foram criados métodos, sendo que a primeira linha de cada método possui a expressão regular que identifica um *token* e sua classe. Nesse caso, a expressão regular, diferentemente das expressões regulares em Python, são definidas por uma *string* tendo como primeiro

caractere a letra `r`.

Para o funcionamento da ferramenta, os *tokens* que identificam as variáveis declaradas em uma função foram armazenados em uma lista. Além da classe do *token*, foi armazenado o valor que representa o tipo da variável. Estas duas informações são importantes para que seja possível depurar a função e mostrar o valor de cada variável declarada.

3.3.3 Analisador sintático

O segundo passo para a implementação do depurador foi a implementação do analisador sintático. Nesta fase foram escritas as regras que definem a sintaxe de uma função em PL/PgSQL. O analisador sintático recebe como parâmetro de entrada a lista de *tokens* reconhecidos pelo analisador léxico. A partir desta lista e das regras sintáticas, o módulo YACC da biblioteca PLY faz a análise sintática das funções analisadas.

Na especificação das regras sintáticas (Quadro 17, Apêndice B) foram utilizados três tipos de anotações. Cada regra sintática foi definida através de um método que deve começar com `statament`, `expression` ou `empty`. A regra sintática anotada com a palavra `statament` é utilizada como a regra inicial do analisador sintático. As regras sintáticas do tipo `expression` são utilizadas para definir as principais regras sintáticas da linguagem. Já a regra do tipo `empty` define uma regra vazia, que é utilizada como finalização de outras regras. Normalmente, as regras dos tipos `statament` e `empty` são declaradas apenas uma vez na construção das regras sintáticas.

```
#Regra inicial
def p_statement_assign(self, p):
    """statement : CREATE OR REPLACE FUNCTION ID LPAREN expression_type RPAREN
RETURNS TYPE LANGUAGE PLPGSQL AS DOLLAR DECLARE expression_variavel BEGIN
expression_comando END SEMICOLON DOLLAR empty"""
    pass

#Regra do tipo empty
def p_empty(self, p):
    "empty :"
    pass

#Regra que define um tipo
def p_expression_type(self, p):
    """expression_type : TYPE
                       | TYPE COMMA expression_type"""
```

Quadro 17 – Especificação sintática

As anotações possuem um padrão onde as palavras em letras maiúsculas são os *tokens* e as palavras em letras minúsculas são regras sintáticas. Para cada *token* declarado em uma regra sintática, deve haver um *token* correspondente na lista de *tokens* extraídos pelo

analisador léxico, e para cada regra sintática definida em outra regra sintática, deve haver uma anotação correspondente.

O analisador sintático começa a análise pela regra sintática do tipo `statament`. Em seguida, analisa as regras sintáticas na ordem que as mesmas foram definidas, até que não localize mais nenhuma regra e/ou até que todos os *tokens* da lista reconhecida pelo analisador léxico sejam analisados. Caso o analisador sintático analise todas as regras sem processar toda a lista de *tokens*, processe toda a lista de *tokens* sem analisar todas as regras sintáticas ou não encontre a regra sintática adequada ao *token* da lista, será gerado um erro de sintaxe.

3.3.4 Depurador

A terceira e última parte da implementação da ferramenta é o depurador. Esta parte contempla o analisador semântico e o depurador. O analisador semântico desenvolvido utiliza a lista de *tokens* para verificar se o valor da variável corresponde ao tipo definido para a mesma em cada comando que a variável sofre alguma alteração. Se durante a análise semântica ocorrer um erro, o processo de depuração será interrompido, exibindo para o usuário uma mensagem informando o erro e listando as variáveis com os seus valores, em cada linha, até a linha que ocasionou o erro.

O depurador basicamente percorre a lista de comandos da função obtida durante a análise sintática, verificando que tipo de comando que está sendo tratado e executando o mesmo. Após a execução, o valor obtido é enviado para o analisador semântico juntamente com a variável que irá receber este valor para verificar a compatibilidade de tipos entre ambos (Quadro 18).

```
#método responsável por verificar a compatibilidade da variável
#com o dado que lhe será atribuído
def testa_tipo_dado(self, k, v):
    v = str(v)
    if v:
        tipo = self.tipo_dado[self.variavel[k][0]]
        dado = "%s(%s)" % (tipo, v)

        if self.debug: print " ==> %s - %s - %s" % (v, tipo, dado)
        try:
            if not tipo in ('date', 'time', 'datetime'):
                eval(dado)

            if tipo == 'int' and v.find('.') > -1: raise
            elif tipo == 'date' and '0000-00-00' != re.sub('\d', '0', v): raise
            elif tipo == 'time' and '00:00:00' != re.sub('\d', '0', v[:8]): raise
            elif tipo == 'datetime' and '0000-00-00 00:00:00' != re.sub('\d',
'0', v[:19]): raise
```
Quadro 18 – Código fonte do depurador

```
                return True
            except Exception, e:
                print e
                return False

    #método responsável por determinar o tipo de comando que será tratado
    #e chamar o método adequado para execução do mesmo
    def analisa_expressao(self, linha, comando):
        ret_teste = True

        try:
            if comando.startswith('if'):
                if not self.cmd_if:
                    self.ret_if = self.DepuradorBase.executa_comando_if(comando)
                    print "Linha: %s <> Token: IF <> Valor: %s " % (linha,
self.ret_if)
                    self.cmd_if = True
                else:
                    self.cmd_if = False

            if comando.startswith('else') and self.cmd_if:
                print "Linha: %s <> Token: ELSE" % linha
                self.ret_if = not self.ret_if

            if self.cmd_if and not self.ret_if:
                return True

            if comando.startswith('return'):
                print "Linha: %s <> Token: RETURN" % linha
                return False

            if comando.startswith('select'):
                k, v = self.DepuradorBase.executa_select(comando)
                print "Linha: %s <> Token: %s <> Valor: %s" % (linha, k, v)
                self.token_valor.append( (linha, k, v) )
                self.variavel[k][1] = v or 'None'
                ret_teste = self.testa_tipo_dado(k, v)

            if comando.find(':=') > -1:
                k, v = self.DepuradorBase.executa_atribuicao(comando)
                print "Linha: %s <> Token: %s <> Valor: %s" % (linha, k, v)
                self.token_valor.append( (linha, k, v) )
                self.variavel[k][1] = v
                ret_teste = self.testa_tipo_dado(k, v)

            if not ret_teste:
                msg = "Erro na linha %s: esperado tipo %s" % (linha,
self.variavel[k][0])
                self.lista_erro.append(msg)
                return False

            return True
        except Exception, e:
            msg = "ERRO na linha %s: %s" % (linha, e.__str__().replace('ERRO:', ''))
            msg = msg.split('\n')[0]

            self.lista_erro.append(msg)
            return False
```

Quadro 18 – Código fonte do depurador (continuação)

3.3.5 Operacionalidade da implementação

A operacionalidade da ferramenta desenvolvida, denominada Debugres, é bastante simplificada. Na Figura 11 pode ser vista a interface da ferramenta. Para utilizá-la é necessário possuir o SGBD PostgreSQL instalado na máquina onde a ferramenta será executada ou ser possível conectar-se a um SGBD PostgreSQL.

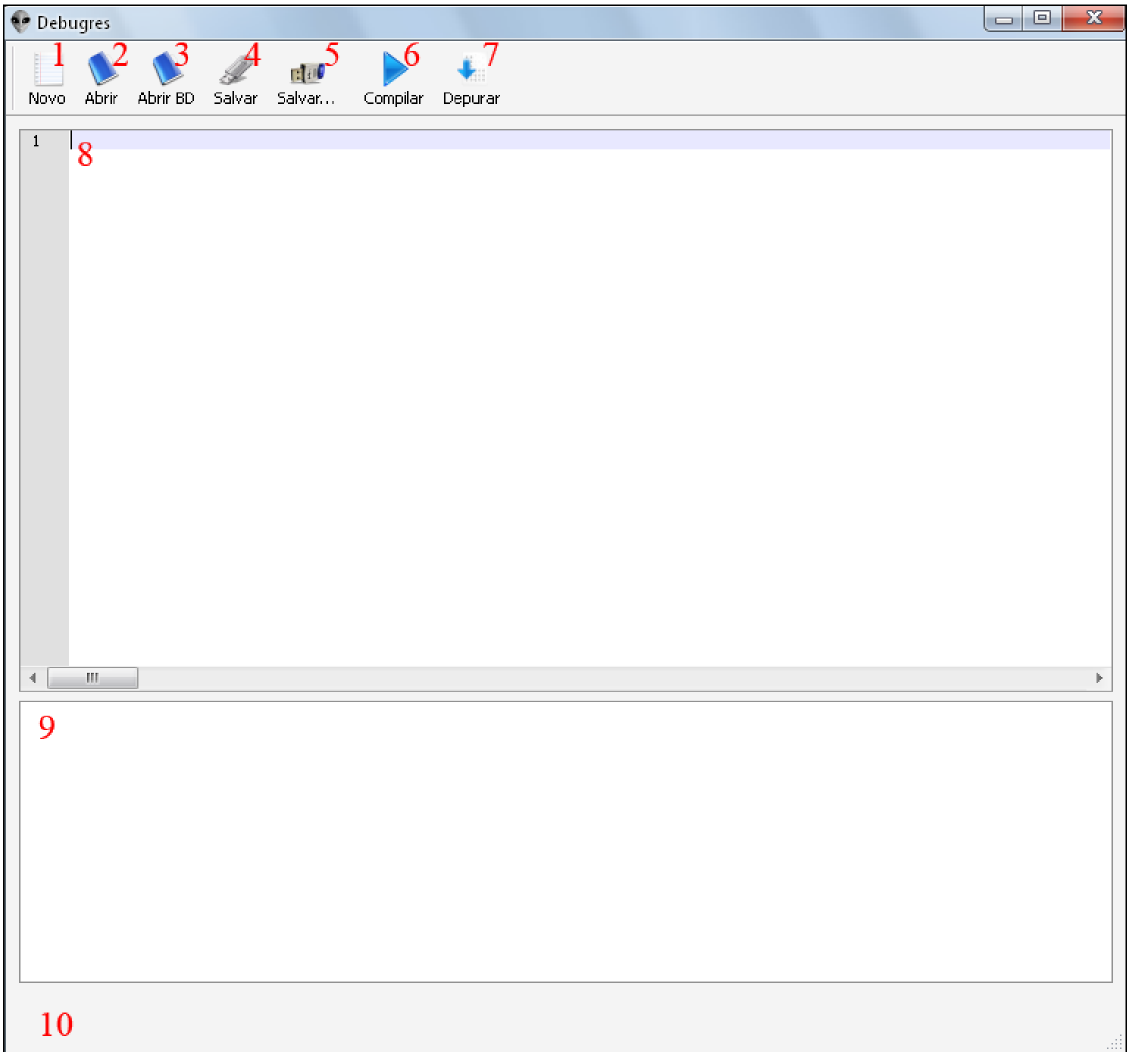


Figura 11 – Interface da ferramenta Debugres

Para iniciar o uso da ferramenta Debugres é necessário configurar o arquivo `conexao.txt` (Figura 12).

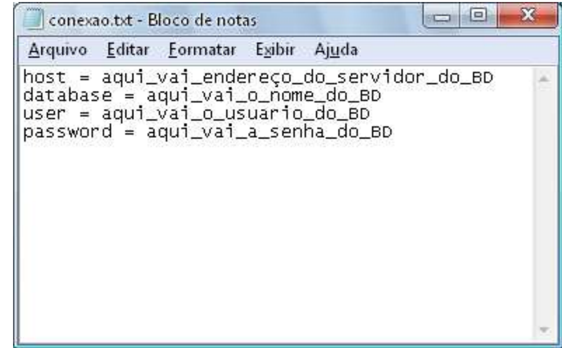


Figura 12 – Configurando o arquivo `conexao.txt`

A primeira linha deste arquivo deve conter o *host* do servidor do banco de dados, precedido pelo termo `host=`. A segunda linha identifica o nome da base de dados e deve iniciar com o termo `database=`. Na terceira e na quarta linhas devem ser inseridos, respectivamente, o nome do usuário para conexão no banco de dados (`user=`) e a senha do usuário (`password=`).

A ferramenta Debugres possui botões com ações para manipulação de arquivos e para compilação e depuração de funções. A ação associada ao botão `Novo` (Figura 11-1) limpa o editor da ferramenta (Figura 11-8) e permite criar uma nova função. Ao clicar no botão `Abrir` (Figura 11-2), é possível localizar e selecionar uma função existente (Figura 13). Uma vez selecionada, após pressionado o botão `Abrir`, a função é carregada no editor do Debugres.

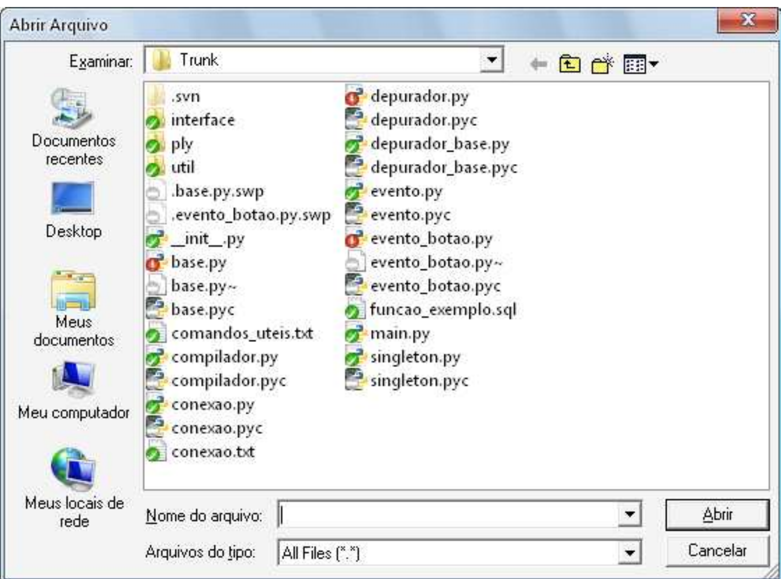


Figura 13 – Localizando e selecionando uma função em arquivo

Ao pressionar no botão `Abrir BD` (Figura 11-3), são listadas as funções disponíveis no banco de dados (Figura 14). O usuário deve selecionar a função desejada e clicar no botão para que a função seja carregada no editor do Debugres.

Figura 14 – Localizando e selecionando funções no banco de dados

Assim, o usuário pode criar uma função nova ou alterar uma função já existente. Clicando no botão `Salvar` (Figura 11-4), é exibida uma mensagem solicitando se o usuário deseja salvar a função no banco de dados (Figura 15) e após é mostrada uma mensagem informando que a função foi salva com sucesso (Figura 16).

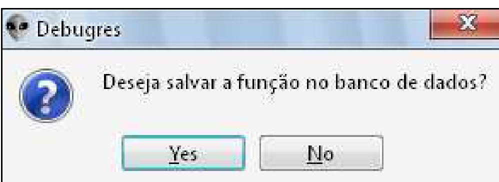


Figura 15 – Mensagem: `Deseja salvar a função no banco de dados?`

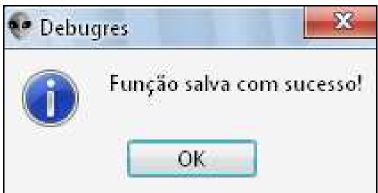


Figura 16 – Mensagem: `Função salva com sucesso!`

A outra forma de salvar uma função é utilizando o botão `Salvar Como` (Figura 11-5). Ao utilizar este botão é possível selecionar o local e informar o nome da função (Figura 17). Esta opção permite somente salvar uma função em um arquivo texto. Após é mostrada uma mensagem informando que a função foi salva com sucesso (Figura 16).

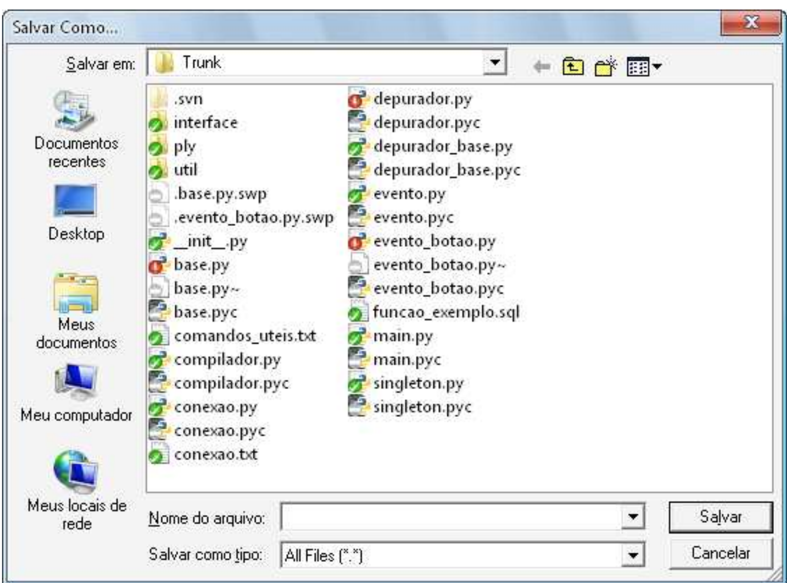


Figura 17 – Selecionado a pasta e informando o nome da função para `Salvar Como`

Qualquer função pode ser compilada clicando no botão `Compilar` (Figura 11-6). Uma função pode ser compilada com sucesso ou apresentar erro, informado através de mensagem com o erro detectado (Figura 18).

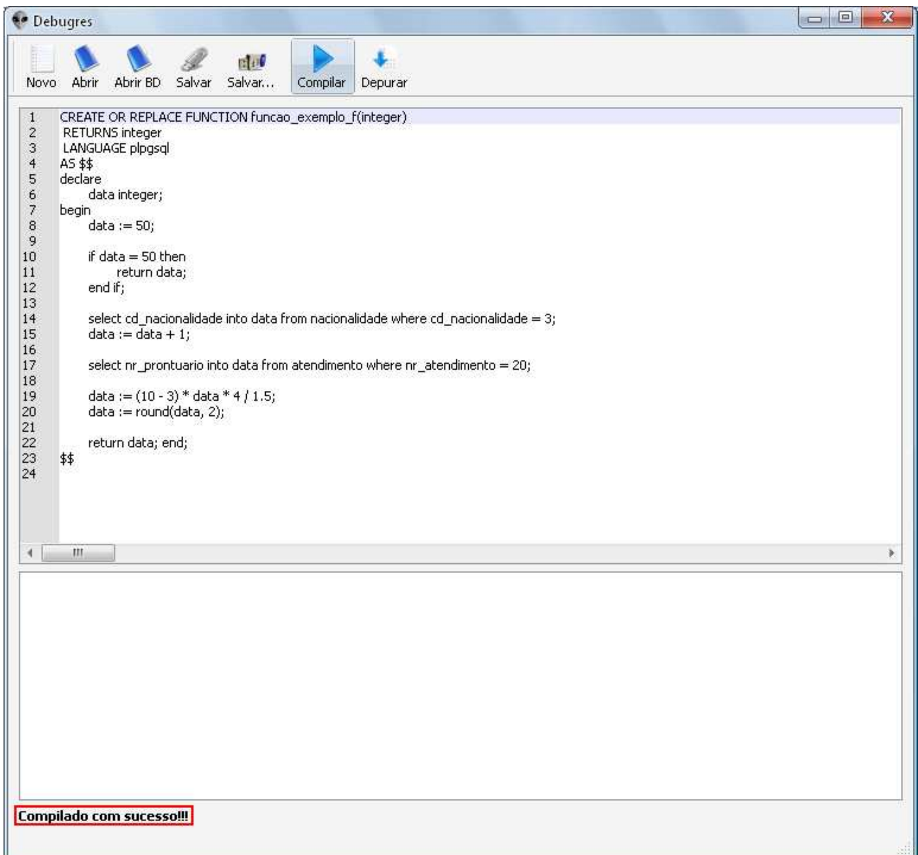


Figura 18 – Mensagem: `Compilado com sucesso!!!`

Ao pressionar o botão `Depurar` (Figura 11-7) é acionado o evento mais importante do Debugres. É a este botão que está associada a ação correspondente ao processo em estudo deste trabalho. Ao clicar no botão `Depurar`, obtém-se a relação dos tipos dos parâmetros de

entrada da função para que o usuário informe o valor de cada um (Figura 19). Também é possível escolher a(s) variável(eis) que será(ão) analisada(s) através do campo `Variáveis`. Deve-se escrever cada variável separada por vírgula ou deixar este campo em branco caso queira que o valor de todas as variáveis seja mostrado.

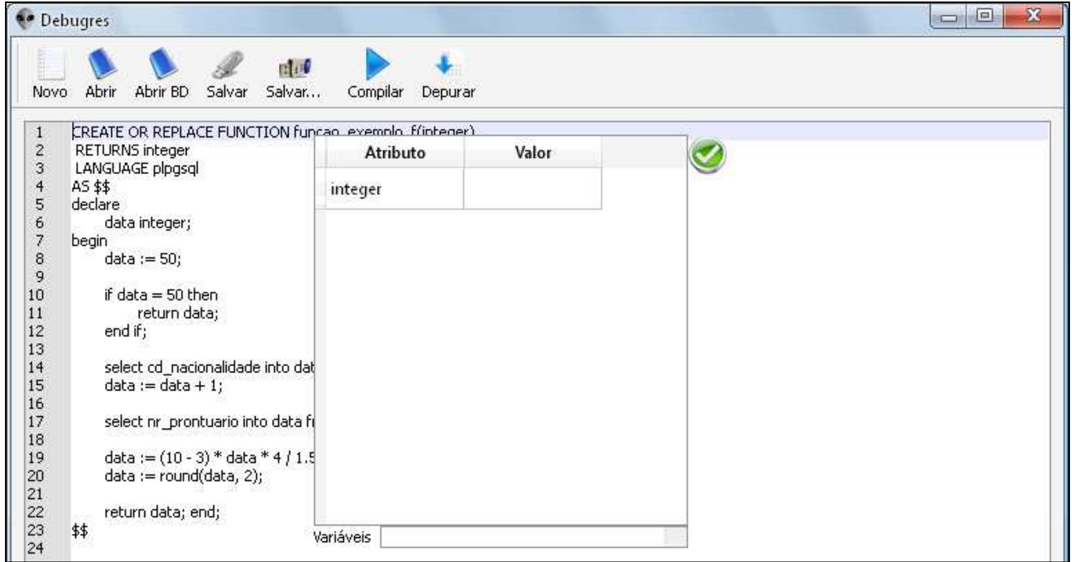


Figura 19 – Listando os tipos dos parâmetros de entrada da função

O resultado do processo de depuração exibe ao final as variáveis com o seu respectivo valor a cada linha de execução da função. Na Figura 20 é mostrado o resultado final da depuração com a detecção de um erro semântico e na Figura 21 tem-se o resultado final sem a detecção de nenhum erro.

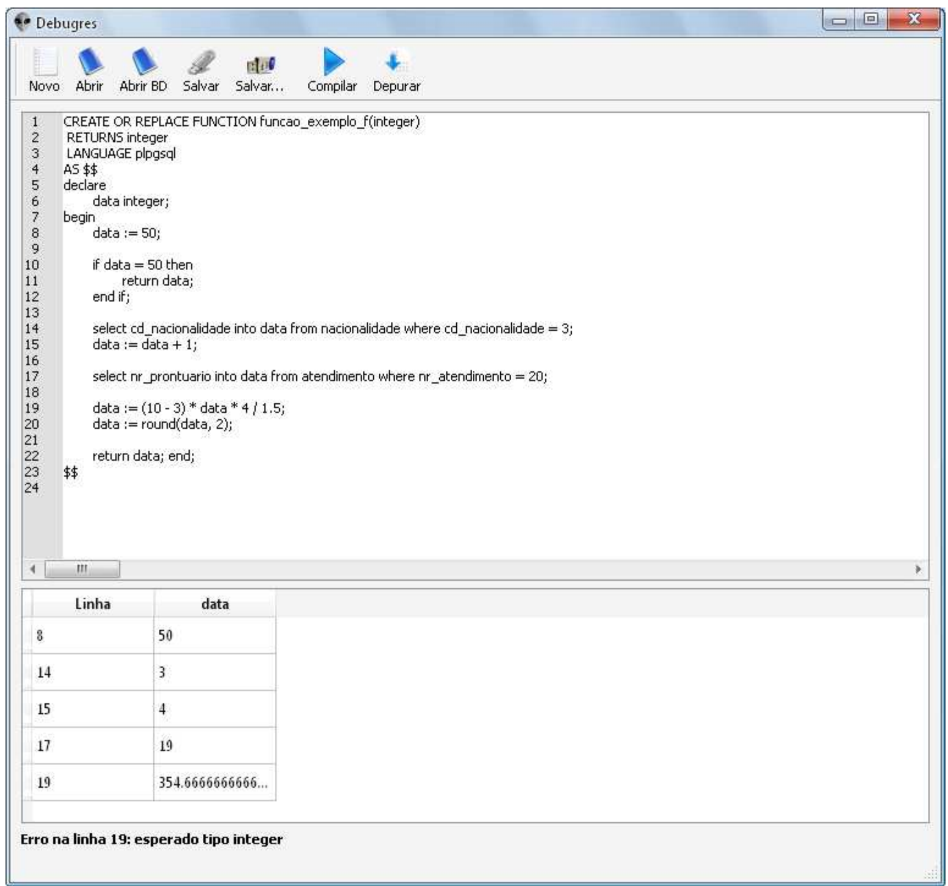


Figura 20 – Depuração com detecção de erro

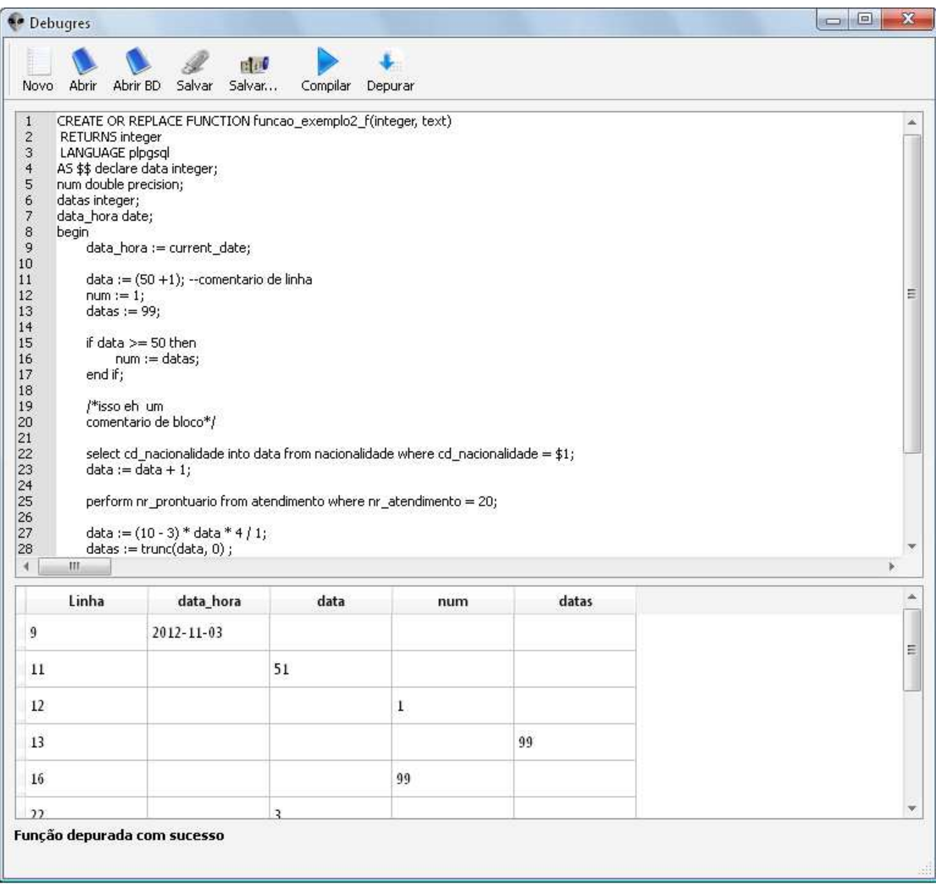


Figura 21 – Depuração sem detecção de erro

3.4 RESULTADOS E DISCUSSÃO

O presente trabalho apresentou o desenvolvimento de uma ferramenta para depuração de funções PL/PgSQL, linguagem procedural do SGBD PostgreSQL. Os resultados obtidos foram satisfatórios, sendo possível afirmar que o objetivo de depurar uma função mostrando o valor das variáveis em cada linha de execução da função foi atingido. No entanto, a gramática usada no desenvolvimento apresenta limitações. Estas limitações foram necessárias para conseguir desenvolver a ferramenta em tempo hábil, pois a linguagem PL/PgSQL possui muitos tipos de dados e comandos.

A ferramenta foi desenvolvida em Python usando as bibliotecas PLY e PyQt, bem como a ferramenta Qt Desinger. Por suas características, o uso de Python facilitou toda a implementação, principalmente no desenvolvimento do analisador semântico e do depurador. Já a biblioteca PLY mostrou-se eficiente para o desenvolvimento dos analisadores léxico e sintático. Mas, apesar do grande poder desta biblioteca, a mesma não é muito utilizada e possui pouco material de apoio, sendo que isso foi um problema para o entendimento inicial da utilização da mesma. O grande percalço da implementação foi com o uso da biblioteca PyQt, para interligação da interface com o Debugres. Mesmo sendo amplamente utilizada, principalmente nos sistemas Unix, foram encontrados alguns problemas com a manipulação de múltiplas janelas, incluindo incompatibilidade com o sistema Windows. Esta incompatibilidade impediu a utilização do comando `pdb` do Python, utilizado para depuração dos códigos fontes escritos nesta linguagem. Mesmo com estes problemas, a biblioteca PyQt ajudou bastante no desenvolvimento do Debugres, apoiada principalmente com a utilização da ferramenta Qt Designer para o desenvolvimento das janelas.

Com relação aos trabalhos correlatos, o que mais se assemelha com a ferramenta desenvolvida é a ferramenta PostgreSQL Maestro, sendo a principal diferença que a ferramenta PostgreSQL Maestro não suporta todas as versões do SGBD PostgreSQL, enquanto a ferramenta Debugres não possui tal limitação. No Quadro 19 é apresentada uma comparação entre os trabalhos correlatos e a ferramenta desenvolvida.

	PL SQL Developer	**SQL Navigator**	**PostgreSQL Maestro**	**Hiebert (2003)**	**Debugres**
é multiplataforma	✗	✗	✗	✗	✓
tem suporte ao SGBD PostgreSQL	✗	✗	✓	✗	✓
tem suporte a todas as versões do SGBD PostgreSQL	✗	✗	✗	✗	✓
permite salvar diretamente no banco de dados	✓	✓	✗	✗	✓
permite abrir diretamente do banco de dados	✓	✓	✗	✓	✓
possui *break point*	✓	✓	✓	✗	✗
tem execução linha a linha	✓	✓	✓	✗	✓
mostra os erros da função	✓	✓	✓	✓	✓

Quadro 19 – Comparativo entre os trabalhos correlatos e a ferramenta Debugres

4 CONCLUSÕES

O trabalho aqui apresentado discorre sobre o desenvolvimento de uma ferramenta para a depuração de funções desenvolvidas na linguagem PL/PgSQL. Funções escritas em linguagem procedural são usadas nos mais diversos SGBDs. Desta forma, tem-se, cada vez mais, a necessidade de ferramentas para compilar e depurar estas funções, pois, como afirmam Geschwinde e Schönig (2002, p. 132), ao inserir uma função PL no banco de dados não se tem garantia que a mesma irá funcionar corretamente, pois o SGBD não efetua toda a análise semântica.

O estudo do SGBD PostgreSQL, dos seus comandos e dos seus tipos de dados, possibilitou um melhor entendimento de como as ferramentas de depuração devem funcionar. Foi este estudo que possibilitou a delimitação do objetivo proposto.

Durante o estudo dos analisadores presentes nos compiladores, viu-se a necessidade de delimitar a gramática a ser atendida, pois sem esta delimitação a ferramenta não seria finalizada em tempo. Aho, Sethi e Ullman (1995, p. 1) afirmam que, “ao longo dos anos 50, os compiladores foram considerados programas difíceis de escrever”. Desde então, foram desenvolvidas várias técnicas para facilitar e agilizar a construção de compiladores e depuradores. Porém, dependendo da linguagem alvo, este processo continua sendo muito trabalhoso: uma ferramenta para depurar funções PL é por si só complexa, tendo em vista que é necessário efetuar as análises léxica, sintática e semântica das funções. Para tanto, foi utilizada a biblioteca PLY. Sem o estudo desta biblioteca não seria possível a obtenção da lista de variáveis e de comandos presentes em uma função.

Com o grande número de comandos e tipos de dados que uma linguagem procedural suporta e a complexidade que as funções desenvolvidas podem alcançar, a probabilidade de ocorrer algum erro durante a compilação ou durante a execução é muito alta. Além disso, no PostgreSQL quando uma função é compilada com sucesso não significa que a mesma será executada sem problemas. Sendo assim, uma ferramenta para depuração auxilia em muito o desenvolvedor a evitar e a localizar possíveis erros que a função desenvolvida poderá conter. Ainda, um depurador pode ajudar o desenvolvedor saber como a função está se comportando a cada linha de execução. Em função dos pontos levantados, a ferramenta foi desenvolvida buscando melhorar o processo de depuração de funções PL/PgSQL por profissionais que utilizam o SGBD Postgres.

A ferramenta desenvolvida possui vantagens em relação as demais presentes no

mercado atualmente. Pode-se destacar a possibilidade de utilização da ferramenta em Linux e em Windows, principais sistemas operacionais nos quais são utilizados o SGBD Postgres. Outra vantagem é que a ferramenta não possui nenhuma incompatibilidade com as várias versões do PostgreSQL, suportando desde as primeiras versões lançadas até a versão atual 9.2.1.

4.1 EXTENSÕES

A ferramenta alcançou o seu objetivo, mas mesmo assim existem pontos que podem ser melhorados e incrementados, sendo eles:

a) aperfeiçoar a gramática, não restringindo os tipos de dados e os comandos disponíveis para o SGBD PostgreSQL;
b) melhorar as mensagens dos erros identificadas pelos analisadores, deixando mais específicas as correções a serem efetuadas;
c) integrar o depurador diretamente ao SGBD PostgreSQL, eliminando a parte gráfica e facilitando a depuração das funções;
d) incluir a funcionalidade de *breakpoint*.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V. et al. **Compiladores**: princípios, técnicas e ferramentas. 2. ed. Tradução Daniel Vieira. São Paulo: Pearson Addison-Wesley, 2008.

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores**: princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: LTC, 1995.

BARROS NETO, Pedro H. R. Stored functions do PostgreSQL. **Revista SQL Magazine**, [S.l.], n. 100, [2012?]. Disponível em: <http://www.devmedia.com.br/stored-functions-do-postgresql-revista-sql-magazine-100/24708#>. Acesso em: 13 out. 2012.

BEAZLEY, David M. **PLY (Python Lex-Yacc)**. [S.l.], [2011?]. Disponível em: <http://www.dabeaz.com/ply/ply.html>. Acesso em: 30 out. 2011.

DATE, Christopher J. **Introdução a sistemas de banco de dados**. Tradução Daniel Vieira. 4. ed. Rio de Jainero: Elsevier, 2003.

DELAMARO, Márcio E. **Como construir um compilador utilizando ferramentas Java**. São Paulo: Novatec, 2004.

GESCHWINDE, Ewald; SCHÖNIG, Hans-Jürgen. **PostgreSQL developer's handbook**. Indianapolis: Sams, 2002.

GRUNE, Dick et al. **Compiladores**: projeto e implementação. Tradução Vandenderg D. de Souza. Rio de Janeiro: Campus, 2001.

HIEBERT, Dennis. **Protótipo de um compilador para a linguagem PL/SQL**. 2003. 52 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

LOUDEN, Kenneth C. **Compiladores**: princípios e práticas. Tradução Flávio Soares Corrêa da Silva. São Paulo: Pioneira Thomson Learning, 2004.

MANZANO, José A. N. G. **Estudo dirigido de SQL (ANSI/89)**. São Paulo: Érica, 2002.

MILANI, André. **PostgreSQL**: guia do programador. São Paulo: Novatec, 2008.

MORAIS, Pedro; PIRES, José N. **Python**: curso completo. Lisboa: FCA, 2002.

OLIVEIRA, Halley P. **Documentação do PostgresSQL 8.0.0**: projeto de tradução para o português do Brasil. [S.l.], 2007. Disponível em: <http://sourceforge.net/projects/pgdocptbr>. Acesso em: 10 set. 2011.

PRICE, Ana M. A.; TOSCANI, Simão S. **Implementação de linguagens de programação**: compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

PY, Hesley S. **Stored procedure no PostgreSQL**: introdução. [S.l.], 2007. Disponível em: <http://www.devmedia.com.br/stored-procedure-no-postgresql-introducao/6550>. Acesso em: 13 out. 2012.

QUEST SOFTWARE. **SQL Navigator**: solução de desenvolvimento PL/SQL para Oracle. [S.l.], [2011?]. Disponível em: <lh.com.br/portal/suporte/arquivos/quest/SQL-Navigator/SQL%20Navigator%20BR.pdf>. Acesso em: 10 set. 2011.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes**. Sistemas de bancos de dados**. 3. ed. Tradução Acauan Pereira Fernandes, Celia Taniwaki, João Tortello. São Paulo: McGraw-Hill, 2008.

REMPT, Boudewijn. **GUI programming with Python**: Qt edition. [S.l.], 2002. Disponível em: <http://www.commandprompt.com/community/pyqt/>. Acesso em: 30 out. 2011.

REZENDE, Ricardo. PL/SQL Developer. **SQL Magazine**, [Rio de Janeiro], n. 18, [2004?]. Disponível em: <www.devmedia.com.br/post-7676-Artigo-SQL-Magazine-18-PL-SQL-Developer.html>. Acesso em: 11 set. 2011.

SILVA, Rodolpho. Cursores em Oracle. **SQL Magazine**, [Rio de Janeiro], n. 65, [2013?]. Disponível em: <http://www.devmedia.com.br/artigo-sql-magazine-65-cursores-em-oracle/13376>. Acesso em: 29 jan. 2013.

SQL MAESTRO GROUP. **PostgreSQL Maestro**: user's guide. [S.l.], [2010]. Disponível em: <http://www.sqlmaestro.com/products/postgresql/maestro>. Acesso em: 10 set. 2011.

SUMMERFIELD, Mark. **Rapid GUI programming with Python an Qt**: the definitive guide to PyQt programming. Michigan: Pearson Education, 2007.

SUN SOFTWARE. **PL/SQL Developer**: solução para ambiente de desenvolvimento integrado Oracle. São Paulo, [2011?]. Disponível em: <http://www.sunsoftware.com.br/solucoes/ambiente.desenvolvimento/plsql.pdf>. Acesso em: 10 set. 2011.

WERNECK, Pedro. **Comparação de GUIs**. [S.l.], [2010]. Disponível em: <http://www.python.org.br/wiki/ComparacaoDeGUIs#PyQT>. Acesso em: 04 out. 2011.

APÊNDICE A – Código fonte do analisador léxico

No Quadro 20 é apresentado o código fonte dos métodos com a especificação do analisador léxico. A maioria dos métodos possui apenas a expressão regular que é utilizada para identificar um *token*. Alguns métodos, como `t_TYPE` e `t_SEMICOLON`, possuem a função de armazenar a lista de variáveis e a lista de comandos, respectivamente.

```
def t_LPAREN(self, t):
    r'\('
    if self.debug: print 'LParen:    ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_RPAREN(self, t):
    r'\)'
    if self.debug: print 'RParen:    ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_OPERACAO(self, t):
    r'\+|\-|\*|\/'
    if self.debug: print 'Operacao:   ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_OPERADOR(self, t):
    r'<=|>=|<>|!=|<|>|='
    if self.debug: print 'Operador:   ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_JOIN(self, t):
    r'join|left\ join|right\ join|left\ outer\ join|right\ outer\ join|inner\
join|full\ outer\ join'
    if self.debug: print 'Join:       ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_WHERE(self, t):
    r'where'
    if self.debug: print 'Where:      ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_UPPER(self, t):
    r'upper'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t
```

Quadro 20 – Métodos do analisador léxico

```
def t_LOWER(self, t):
    r'lower'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_COALESCE(self, t):
    r'coalesce'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_LENGTH(self, t):
    r'length'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_SUBSTR(self, t):
    r'substr'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_MAX(self, t):
    r'max'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_MIN(self, t):
    r'min'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_NOW(self, t):
    r'now'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_AVG(self, t):
    r'avg'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_ROUND(self, t):
    r'round'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t
```

Quadro 20 – Métodos do analisador léxico (continuação)

```
def t_TRUNC(self, t):
    r'trunc'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_REPLACE(self, t):
    r'replace'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_COUNT(self, t):
    r'count'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_R_PAD(self, t):
    r'r_pad'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_L_PAD(self, t):
    r'l_pad'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_R_TRIM(self, t):
    r'r_trim'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_L_TRIM(self, t):
    r'l_trim'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_CURRENT_DATE(self, t):
    r'current_date'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_CURRENT_TIMESTAMP(self, t):
    r'current_timestamp'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t
```

Quadro 20 – Métodos do analisador léxico (continuação)

```
def t_CURRENT_TIME(self, t):
    r'current_time'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_TRIM(self, t):
    r'trim'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_TO_CHAR(self, t):
    r'to_char'
    if self.debug: print 'Funcao:     ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_SATRIB(self, t):
    r':='
    if self.debug: print 'Atrib:      ' + t.value

    self.comando = True
    self.lista_string_comando.append(t.value)

    return t

def t_RETURNS(self, t):
    r'returns'
    if self.debug: print 'Returns:    ' + t.value
    return t

def t_RETURN(self, t):
    r'return'
    if self.debug: print 'Return:     ' + t.value
    return t

def t_INTO(self, t):
    r'into'
    if self.debug: print 'Into:       ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_FROM(self, t):
    r'from'
    if self.debug: print 'From:       ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_PERFORM(self, t):
    r'perform'
    if self.debug: print 'Perform:     ' + t.value

    self.comando = True
    self.lista_string_comando.append(t.value)

    return t
```

Quadro 20 – Métodos do analisador léxico (continuação)

```
def t_SELECT(self, t):
    r'select'
    if self.debug: print 'Select:     ' + t.value

    self.comando = True
    self.lista_string_comando.append(t.value)

    return t

def t_BEGIN(self, t):
    r'begin'
    if self.debug: print 'Begin:      ' + t.value

    self.var_flag = False
    return t

def t_END(self, t):
    r'end'
    if self.debug: print 'End:        ' + t.value
    return t

def t_DECLARE(self, t):
    r'declare'
    if self.debug: print 'Declare:    ' + t.value

    self.var_flag = True
    return t

def t_CREATE(self, t):
    r'create'
    if self.debug: print 'Create:     ' + t.value
    return t

def t_ORDER(self, t):
    r'order\ by'
    if self.debug: print 'Order:      ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_AND(self, t):
    r'and'
    if self.debug: print 'And:        ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_OR(self, t):
    r'or'
    if self.debug: print 'Or:         ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_AS(self, t):
    r'as'
    if self.debug: print 'As:         ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t
```

Quadro 20 – Métodos do analisador léxico (continuação)

```
def t_IF(self, t):
    r'if'
    if self.debug: print 'If:  ' + t.value

    self.comando = True
    self.lista_string_comando.append(t.value)

    return t

def t_THEN(self, t):
    r'then'
    if self.debug: print 'Then:  ' + t.value

    if self.comando:
        self.lista_string_comando.append(t.value)
        self.finaliza_comando(t.lineno)

    return t

def t_ELSE(self, t):
    r'else'
    if self.debug: print 'Else:  ' + t.value

    self.lista_string_comando.append(t.value)
    self.finaliza_comando(t.lineno)

    return t

def t_FUNCTION(self, t):
    r'function'
    if self.debug: print 'Function:  ' + t.value
    return t

def t_LANGUAGE(self, t):
    r'language'
    if self.debug: print 'Language:  ' + t.value
    return t

def t_PLPGSQL(self, t):
    r'plpgsql'
    if self.debug: print 'PlPgsql:   ' + t.value
    return t

def t_COMMA(self, t):
    r'\,'
    if self.debug: print 'Virgula:   ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_SEMICOLON(self, t):
    r'\;'
    if self.debug: print 'Semicolon: ' + t.value

    if self.comando:
        self.lista_string_comando.append(t.value)
        self.finaliza_comando(t.lineno)

    return t

def t_DOLLAR(self, t):
    r'\$\$'
    if self.debug: print 'Dollar:    ' + t.value
    return t
```

Quadro 20 – Métodos do analisador léxico (continuação)

```
def t_TYPE(self, t):
    r'bigint|bigserial|boolean|bytea|date|double\
precision|integer|interval|numeric|serial|smallint|text|timestamp|void|time'
    if self.debug: print 'Type:       ' + t.value

    if self.var_flag:
        self.variavel[self.var_id] = [t.value, None]

    return t

def t_ID(self, t):
    r'[a-zA-Z\_][a-zA-Z0-9\_]*'

    if self.var_flag:
        self.var_id = t.value.strip()
        if self.debug: print 'Variavel:   ' + t.value
    else:
        if self.debug: print 'ID:         ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_DECIMAL(self, t):
    r'\d+(\.\d+)?'
    if self.debug: print 'Decimal:    ' + t.value

    if self.comando: self.lista_string_comando.append(t.value)
    return t

def t_newline(self, t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")

def t_error(self, t):
    msg_erro = "ERRO: Caracter invalido '%s'\n" % t.value[0].__repr__()
    print msg_erro
    self.lista_erro.append(msg_erro)
    t.lexer.skip(1)
```

Quadro 20 – Métodos do analisador léxico (continuação)

APÊNDICE B – Código fonte do analisador sintático

No Quadro 21 é apresentado o código fonte dos métodos com a especificação do analisador sintático. Os métodos do analisador sintático possuem apenas as regras sintáticas, utilizadas para a validação da função verificando se as mesmas estão implementas conforme a especificação da linguagem PL/PgSQL.

```
def p_statement_assign(self, p):
    """statement : CREATE OR REPLACE FUNCTION ID LPAREN expression_type RPAREN
RETURNS TYPE LANGUAGE PLPGSQL AS DOLLAR DECLARE expression_variavel BEGIN
expression_comando END SEMICOLON DOLLAR empty"""
    pass

def p_expression_type(self, p):
    """expression_type : TYPE
                       | TYPE COMMA expression_type"""
    pass

def p_empty(self, p):
    "empty :"
    pass

def p_expression_variavel(self, p):
    """expression_variavel : ID TYPE SEMICOLON
                  | ID TYPE SEMICOLON expression_variavel"""
    pass

def p_expression_funcaobd(self, p):
    """expression_funcaobd : UPPER LPAREN ID RPAREN
                           | LOWER LPAREN ID RPAREN
                           | COALESCE LPAREN ID COMMA ID RPAREN
                           | LENGTH LPAREN expression_operacao RPAREN
                           | SUBSTR LPAREN ID COMMA expression_operacao COMMA
DECIMAL RPAREN
                           | MAX LPAREN expression_operacao RPAREN
                           | MIN LPAREN expression_operacao RPAREN
                           | AVG LPAREN expression_operacao RPAREN
                           | ROUND LPAREN expression_operacao COMMA DECIMAL
RPAREN
                           | TRUNC LPAREN expression_operacao COMMA DECIMAL
RPAREN
                           | REPLACE LPAREN expression_operacao COMMA ID COMMA ID
RPAREN
                           | COUNT LPAREN expression_operacao RPAREN
                           | R_PAD LPAREN expression_operacao COMMA DECIMAL COMMA
ID RPAREN
                           | L_PAD LPAREN expression_operacao COMMA DECIMAL COMMA
ID RPAREN
                           | R_TRIM LPAREN expression_operacao COMMA ID RPAREN
                           | L_TRIM LPAREN expression_operacao COMMA ID RPAREN
                           | TRIM LPAREN expression_operacao COMMA ID RPAREN
                           | TO_CHAR LPAREN expression_operacao COMMA ID RPAREN
                           | NOW LPAREN RPAREN
                           | CURRENT_DATE
                           | CURRENT_TIME
                           | CURRENT_TIMESTAMP"""
```

Quadro 21 – Métodos do analisador sintático

```
def p_expression_where(self, p):
    """expression_where : ID OPERADOR ID
                        | ID OPERADOR DECIMAL
                        | DECIMAL OPERADOR ID
                        | expression_where AND expression_where
                        | expression_where OR expression_where"""
    pass

def p_expression_order(self, p):
    """expression_order : ID
                        | ID COMMA expression_order
                        | DECIMAL
                        | DECIMAL COMMA expression_order"""
    pass

def p_expression_tabela(self, p):
    """expression_tabela : ID
                         | JOIN ID LPAREN expression_where RPAREN
expression_tabela"""
    pass

def p_expression_coluna(self, p):
    """expression_coluna : ID
                         | expression_funcaobd
                         | ID AS ID
                         | ID COMMA expression_coluna
                         | ID AS ID COMMA expression_coluna"""
    pass


def p_expression_select(self, p):
    """expression_select : SELECT expression_coluna INTO ID SEMICOLON
                         | SELECT expression_coluna INTO ID FROM
expression_tabela SEMICOLON
                         | SELECT expression_coluna INTO ID FROM
expression_tabela WHERE expression_where SEMICOLON
                         | SELECT expression_coluna INTO ID FROM
expression_tabela ORDER expression_order SEMICOLON
                         | SELECT expression_coluna INTO ID FROM
expression_tabela WHERE expression_where ORDER expression_order SEMICOLON
                         | PERFORM expression_coluna SEMICOLON
                         | PERFORM expression_coluna FROM expression_tabela
SEMICOLON
                         | PERFORM expression_coluna FROM expression_tabela WHERE
expression_where SEMICOLON
                         | PERFORM expression_coluna FROM expression_tabela ORDER
expression_order SEMICOLON
                         | PERFORM expression_coluna FROM expression_tabela WHERE
expression_where ORDER expression_order SEMICOLON"""
    pass

def p_expression_id(self, p):
    """expression_id : ID
                     | DECIMAL"""
    pass

def p_expression_operacao(self, p):
    """expression_operacao : expression_id
                           | expression_id RPAREN
                           | expression_id RPAREN OPERACAO expression_operacao
                           | expression_id OPERACAO expression_operacao
                           | LPAREN expression_id OPERACAO expression_operacao
                           | expression_funcaobd
                           | expression_funcaobd OPERACAO expression_operacao
    """
    pass
```

Quadro 21 – Métodos do analisador sintático (continuação)

```
def p_expression_atribuicao(self, p):
    """expression_atribuicao : ID SATRIB expression_operacao SEMICOLON"""

    comando = self.lista_comando[len(self.lista_comando)-1][1]
    self.lista_comando[len(self.lista_comando)-1][1] = p[1] + comando

def p_expression_return(self, p):
    "expression_return : RETURN ID SEMICOLON"

    comando = []
    for r in p.slice:
        if r.value:
            comando.append(r.value)

    self.lista_comando.append( [p.lineno(1), " ".join(comando)])
    pass

def p_expression_if(self, p):
    """expression_if : IF expression_operacao OPERADOR expression_operacao THEN
expression_comando END IF SEMICOLON
                     | IF expression_operacao OPERADOR expression_operacao THEN
expression_comando ELSE expression_comando END IF SEMICOLON"""
    pass

def p_expression_comando(self, p):
    """expression_comando : expression_select expression_comando
                          | expression_atribuicao expression_comando
                          | expression_return expression_comando
                          | expression_if expression_comando
                          | empty"""
    pass

def p_error(self, p):
    if p:
        msg_erro =  "ERRO: Linha %s possui erro de sintaxe no caracter '%s'\n" %
(p.lineno, p.value)
        print msg_erro
        self.lista_erro.append(msg_erro)
```

Quadro 21 – Métodos do analisador sintático (continuação)