CENTRO UNIVERSITÁRIO FEEVALE

ANA TRINDADE WINCK

O DESENVOLVIMENTO DE UM COMPILADOR PARA TRADUÇÃO DE UM SISTEMA LEGADO

Novo Hamburgo, novembro de 2005

ANA TRINDADE WINCK

O DESENVOLVIMENTO DE UM COMPILADOR PARA TRADUÇÃO DE UM SISTEMA LEGADO

CENTRO UNIVERSITÁRIO FEEVALE
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

Professor Orientador: Ricardo Ferreira de Oliveira

Novo Hamburgo, novembro de 2005

Winck, Ana Trindade
O desenvolvimento de um compilador para tradução de um sistema legado / Ana Trindade Winck – 2005.
97 f. ; 28cm.

Trabalho de conclusão do curso de Ciência da computação do Centro Universitário Feevale.
. "Orientador: Ricardo Ferreira de Oliveira".
Inclui bibliografia.

1. Sistemas Legados 2. Compiladores(Computadores) 3. Clipper(Programa de computador) 4. Java(Linguagem de programação de computador) I. Título.

CDU 004.41

Agradecimentos

Agradeço a meus pais por todo seu amor, carinho e confiança empregados em mim, e por seu investimento para a realização dessa etapa de nossas vidas. Ao meu noivo por seu amor, compreensão e companheirismo. À minha vó e à minha dinda por todo carinho e admiração. Essas pessoas são, sem dúvida, o meu alicerce. A vocês, muito obrigada!

Agradeço, também, aos colegas da Feevale pelo apoio e encorajamento.

Agradeço, ainda, à Quatro Informática, pela sua estrutura e por suas pessoas, por ter me propiciado a realização e concretização deste trabalho.

RESUMO

Traduzir um sistema escrito em uma linguagem antiga para uma linguagem contemporânea pode tornar-se um processo lento. Visando diminuir o tempo da etapa de tradução propõe-se a construção de um compilador para efetuar a maior parte deste decurso. O foco deste projeto concentra-se na tradução de um sistema legado, através do desenvolvimento de um compilador. Este sistema vem a ser traduzido para uma nova plataforma de desenvolvimento, buscando, assim, a atualização do mesmo. Para tanto, este trabalho busca propor uma solução de reengenharia de sistemas legados, econômica em termos de desenvolvimento, definindo soluções de tradução a partir do desenvolvimento de um compilador. Este deve converter códigos de uma linguagem procedural para uma linguagem orientada a objetos. Este novo código, então, assumirá características próprias e atualizadas, atendendo aos requisitos originais e otimizando, inclusive, partes significantes dele.

Palavras-chave: Sistemas Legados, Compiladores, Clipper, Java.

ABSTRACT

Translating a system written in an old language to a contemporary language can be a slow process. To diminish this time it is considered the construction of a compiler to automatize most of this job. The focus of this project is the translation of a legacy system, through the development of a compiler. This legacy system is translated to a new development platform, becoming then, an upgraded system. To make so, this project proposes a legacy systems reengineering solution, economic development-wise, defining translation solutions through the development of a compiler. The compiler must translate procedural code into object-oriented language. The new code will assume its own, updated characteristics, attending all the original requirements, and even optimizing significant parts of the original code.

Key-Words: Legacy Systems, Compilers, Clipper, Java.

LISTA DE FIGURAS

Figura 1.1 – Modelo em camadas de um sistema legado. (SOMMERVILLE, 2001, p.501)..... 19

Figura 1.2 – Modelo clássico do ciclo de vida. (PRESSMAN, 1995, p.33). 21

Figura 1.3 – Distribuição do esforço da manutenção. (SOMMERVILLE , 2001, p.518) 22

Figura 1.4 – O ciclo de vida da modernização do software (SEACORD, et al, tradução nossa). 25

Figura 2.1 – Um compilador. (AHO et al, 1995, p.1) 32

Figura 2.2 – Estrutura de um compilador (AHO, et al, 1995, p.5)..... 34

Figura 2.3 – Gramática simples para avaliação de expressões aritiméticas 36

Figura 3.1 - Modelos em camadas do sistema traduzido e do sistema legado – Adaptado de (SOMMERVILLE,2001)..... 50

Figura 3.2 – Diagrama de Classes do compilador Clipper2Java..... 52

Figura 3.3 – Diagrama de caso de uso do compilador Clipper2Java 53

Figura 3.4 – Diagrama de Colaboração do compilador Clipper2Java..... 54

Figura 3.5 – Diagrama de classes da máquina JClipper 56

Figura 3.8. A Arquitetura do ambiente JClipper 57

Figura 4.1 – Exemplos de regras de produção da gramática elaborada para o compilador 62

Figura 4.2 – Exemplo de definições de *tokens* na classe ***Lexico*** 63

Figura 4.3 – Início do método ***buscaProximoToken*** 64

Figura 4.4 – Método que implementa a análise sintática para a regra ***expressao*** 66

Figura 4.5 - Parte do método ***acaoSemantica*** que invoca métodos correspondentes a cada ação semântica. 67

Figura 4.6 – Método que implementa o conjunto de regras semânticas do comando ***if*** 68

Figura 5.1- Exemplo de código Clipper alterado para a importação 75

Figura 5.2 – interface do Clipper2Java 76

Figura 5.3 – Mensagem de erro no *console* da plataforma Eclipse 77

Figura 5.4 – impressão do código Java na tela do compilador 78

Figura 6.1 – Exemplo de utilização da classe ***Variavel***, com e sem correspondência de tipo de dados das variáveis na tabela de símbolos 80

Figura 6.2 – Exemplo de código traduzido com o compilador 82

Figura 6.3 – Interface de um programa da aplicação Java 85

LISTA DE ABREVIATURAS

BNF *Backus-Naür-Form ou Backus-Normal-Form*

EJB *Enterprise JavaBeans*

LALR *Lookahead LR*

LL *Left-to-Right Scanning, Leftmost Derivation (leitura da esquerda para à direita, com derivação mais à esquerda)*

LNK *Extensão de arquivo a serem linkeditados (link)*

LR *Left-to-Right Scanning, Rightmost Derivation (leitura da esquerda para a direita, com derivação mais à direita)*

RMK *Extensão de arquivo com ações a serem realizadas (rmake)*

SLR *Simple LR*

SGBD *Sistema Gerenciador de Banco de Dados*

TI *Tecnologia da Informação*

SUMÁRIO

INTRODUÇÃO 14

1 SISTEMAS LEGADOS 18

1.1 Manutenção de Software 20

1.2 A Modernização do Software 23

1.3 O Dilema dos Sistemas Legados 25

1.3.1 O Desafio da Modernização 27

1.4 A Tradução de Códigos Legados 28

1.4.1 Reengenharia 29

1.4.2 Tradutores Automáticos de Código 30

2 COMPILADORES 32

2.1 Gramáticas 34

2.2 Análise Léxica 38

2.3 Analise Sintática 39

2.3.1 Análise Sintática Descendente 40

2.3.2 Análise Sintática Ascendente 41

2.4 Análise Semântica 42

2.5 Tratamento de Erros 42

2.6 Aplicações de Compiladores 43

3 O PROJETO DO COMPILADOR PARA TRADUÇÃO DE CÓDIGO LEGADO 45

3.1 O Sistema Escrito em Clipper e a Motivação para a Modernização 45

3.2 A Escolha das Novas Tecnologias 47

3.2.1 A Linguagem Java 47

3.2.2 PostgreSQL 48

3.3 O Compilador Clipper2Java 49

3.4 A Máquina JClipper 54

3.4.1 Interpretador de Comandos 58

3.4.2 Simulador de Comandos de Tela 58

3.4.3 Simulador de Comandos de Arquivos 59

3.5 Construção da Ferramenta 60

4 A IMPLEMENTAÇÃO DO COMPILADOR 61

4.1 A Gramática 61

4.2 A Classe Lexico 63

4.3 A Classe Sintatico 65

4.4 A Classe Semantico 67

4.4.1 Métodos da Análise Semântica 69

4.5 Adição de Novos Métodos 69

5 A UTILIZAÇÃO DO COMPILADOR CLIPPER2JAVA 70

5.1 Alterações no Código Clipper 70

5.1.1 Identificador de Pacote 71

5.1.2 Identificador de Classe 72

5.1.3 Identificador do Método main 73

5.1.4 Identificador de Final de Função 74

5.2 Código a Ser Traduzido 74

5.3 Importação do Código Clipper 76

6 TESTES E RESULTADOS COM CÓDIGOS JAVA 79

6.1 Testes e Alterações em Códigos e Aplicações Java 80

6.1.1 Testes em Aplicações 83

6.1.2 Modificações na Camada de Persistência 83

6.2 Resultados das Aplicações 84

CONCLUSÕES 87

REFERÊNCIAS BIBLIOGRÁFICAS 89

ANEXO 1 – GRAMÁTICA 93

INTRODUÇÃO

As tecnologias utilizadas nas linguagens de programação permanecem em constante evolução. São várias as linguagens hoje disponíveis para o desenvolvimento de software.

Neste trabalho, é avaliado um sistema que atende a uma rede de cooperativa médica. A concepção deste sistema data de mais de dez anos e sua construção foi sobre a linguagem Clipper. Segundo Sommerville (2001), Sistemas antigos, mas que ainda fornecem serviços essenciais, são denominados sistemas legados.

> *O tempo de duração de sistemas de software é muito variável, e muitos sistemas de grande porte permanecem em uso por mais de 10 anos. Algumas organizações ainda dependem de sistemas de software que tem mais de 20 anos de existência. Muitos desses antigos sistemas ainda são fundamentais para as empresas, isto é, as empresas dependem dos serviços fornecidos pelo software, e qualquer falha desses serviços teria um sério efeito em seu dia-a-dia. A esses antigos sistemas foi dado o nome de sistemas legados. (SOMMERVILLE, 2001, p.498)*

Em geral, os softwares apresentam um ciclo de vida que inicia-se em sua idealização e termina em sua manutenção. Para Pressman (1995) o ciclo de vida de um software inicia-se na idealização do sistema e avança ao longo da análise, projeto, codificação, teste e manutenção.

O sistema aqui avaliado é amplo e flexível, atendendo de forma satisfatória, dentro de sua abrangência, as necessidades operacionais de seus clientes. Contudo, foi desenvolvido há muito tempo. Sua lógica é aplicada a partir de métodos procedurais inerentes à linguagem Clipper, e sua arquitetura de dados é baseada em sistemas de arquivos.

Este sistema tem boa aceitação por parte de seus usuários, no entanto, em função de sua tecnologia obsoleta e de suas constantes manutenções, seu ciclo de vida já chegou ao final. Com a finalidade de que sua utilização permaneça ativa, é recomendado que, a este sistema, seja aplicado um processo de reengenharia.

Busca-se, então, uma solução prática para realizar este processo. O objetivo desta solução é traduzir para uma nova tecnologia o que já estaria obsoleto.

Substituindo a linguagem procedural Clipper, aplica-se o paradigma de orientação a objetos a partir da linguagem Java. De acordo com Paula Filho (2003, p. 120) "Nas metodologias de modelagem orientadas a objetos, as entidades do domínio do problema são representadas por objetos. Objetos podem ser vistos como estruturas de dados encapsuladas por procedimentos".

Além da orientação a objetos, tem-se como objetivo substituir, neste novo sistema, a utilização de sistemas de arquivos por sistemas gerenciadores de banco de dados e, ainda, estruturar este novo software para uma arquitetura em três camadas.

Em uma arquitetura de três camadas, a camada de apresentação depende de outra (a camada de domínio ou negócio) para tomar decisões concernentes a informação, realizar processamento, e preocupar-se para onde os dados vão após a submissão pelo usuário. A camada de domínio, ou camada de negócio, por sua vez, depende da camada de persistência, que usualmente é um sistema gerenciador de banco de dados para armazenar de forma permanente os objetos. (JEPSON et al 2000)

A conversão de sistemas legados vem sendo pesquisada pela engenharia de software, onde se realiza um estudo completo do código fonte original e projeta-se uma melhor performance a ser aplicada ao novo sistema. Várias técnicas de desenvolvimento e aperfeiçoamento de software são examinadas de acordo com o que guia a engenharia reversa. (SOMMERVILLE, 2001). A utilização de uma ferramenta rápida para tradução do sistema legado é proposta por alguns autores como Sommerville (2001) e Seacord et al (2003), entretanto, não é tratado sobre a construção de tal ferramenta.

Objetivando apresentar uma solução de otimização de código, a partir de uma aplicação que utiliza sistemas de arquivos, para um sistema que utilize banco de dados e, aliado à arquitetura em camadas baseada na orientação a objetos, utiliza-se uma estrutura que, suportando um código legado, implementa a camada de apresentação e a camada de negócios. A utilização de um compilador é uma forma de estabelecer esta interface entre o código legado e tal arquitetura.

Conforme Grune et al (2001, p.1) "Em sua forma mais geral, um compilador é um programa que aceita como entrada um texto de programa em uma certa linguagem e produz como saída um texto de programa em outra linguagem, enquanto preserva o significado desse texto".

É, então, desenvolvido um compilador para atender especificamente os objetivos deste projeto. Sendo assim, este compilador tem a finalidade de otimizar o processo de tradução do código-fonte em Clipper para um código em Java. Este preserva a lógica de negócios e mantém a interface original. Seu grande obstáculo é atribuir uma aplicação para banco de dados orientada a objetos, utilizando-se de uma abordagem em três camadas.

O objetivo principal deste trabalho consiste em estudar a reengenharia de sistemas legados para uma arquitetura em três camadas, e desenvolver um compilador que otimize o processo de tradução de um sistema desenvolvido na linguagem Clipper, para a linguagem Java

Este trabalho divide-se da seguinte forma: os dois primeiros capítulos apresentam conceitos teóricos sobre sistemas legados e compiladores. O primeiro capítulo apresenta conceitos de sistemas legados, onde aborda-se sua problemática apontando a necessidade de modernização do mesmo e sugere a utilização de uma ferramenta rápida de tradução de códigos legados, como meio desta modernização. O segundo capítulo apresenta um conceito sobre a estrutura de compiladores e suas partes relevantes.

A partir do capítulo três é abordada a construção do compilador proposto. No terceiro capítulo é apresentado um modelo deste compilador; a construção da frramenta é abordada no capítulo quatro; o quinto capítulo elucida sobre a utilização do compilador; e por fim o último capítulo apresenta os resultados dos códigos gerados pelo mesmo.

1 SISTEMAS LEGADOS

Sistemas antigos cujos serviços são essenciais para a empresa, são denominados sistemas legados (SOMMERVILLE, 2001).

Este capítulo aborda a definição de sistemas legados, ilustrando sua estrutura e a importância que o mesmo representa para as organizações; é apontada a problemática quanto a permanência no mercado e sua respectiva manutenção e evolução, indicando os processos de tradução destes sistemas.

É comum encontrar empresas que mantém ativos sistemas construídos sobre uma tecnologia obsoleta. Entende-se que no período inicial da concepção do sistema, a tecnologia empregada era aceita. Parte destes softwares, no entanto, foram inseridos no contexto operacional da empresa e marcaram grande importância para a sua atuação, tornando-se indispensáveis.

De acordo com Peters; Pedrycz (2001, p. 559) "um sistema legado é aquele que tenha sido desenvolvido de forma a satisfazer às necessidades antigas do cliente e que tenha sido implementado utilizando-se uma tecnologia antiga".

Estes sistemas foram desenvolvidos no passado, em um momento em que a realidade operacional da empresa era diferente, suas necessidades eram outras e, foi sobre essa necessidade original que estes sistemas foram desenvolvidos (PFLEEGER, 2004).

Sistemas de gestão organizacionais costumam ser de expressiva importância por conter dados fundamentais sobre a empresa e por adquirir notáveis processos para a rotina

diária dessas organizações. Diante desta característica, muitos destes softwares conservam-se ativos na empresa por muitos anos, e sua funcionalidade, muitas vezes, permanece corrente mesmo que este software encontre-se obsoleto.

Conforme Sommerville (2001), sistemas legados são um conjunto de sistemas denominados sistemas sociotécnicos que incluem software, hardware, dados e processos corporativos. Estas partes são interligadas, formando um todo e qualquer modificação em alguma destas partes, afetam a esse todo. Este conjunto de sistemas pode ser representado em camadas, conforme mostra a figura 1.1

Figura 1.1 – Modelo em camadas de um sistema legado. (SOMMERVILLE, 2001, p.501)

Com a evolução da tecnologia, é comum que as camadas de hardware e software de apoio sejam atualizadas com maior freqüência do que as camadas de software de aplicação e de processos de negócios. No entanto, a cada modificação destes componentes, as camadas relativas ao software legado sofrem impactos até que, em um dado momento precisam, também, ser atualizadas.

Dessa forma, sistemas legados são considerados sistemas que precisam de algum reparo para ajudá-los a evoluir (PLEEGER, 2004)

1.1 Manutenção de Software

O desenvolvimento de um software está completo quando este software está sendo utilizado em um ambiente real, ou seja, quando este software está operacional. Qualquer trabalho que seja realizado neste software após a sua inserção operacional na empresa, é denominado manutenção (PFLEEGER, 2004).

Em geral, os sistemas apresentam um ciclo de vida. Em sua grande maioria este ciclo inicia-se em seu projeto e termina em sua manutenção. Pressman (1995, p.32) descreve o ciclo de vida clássico: "O paradigma do ciclo de vida requer uma abordagem sistemática, seqüencial ao desenvolvimento do software, que se inicia no nível do sistema e avança ao longo da análise, projeto, codificação, teste e manutenção". A figura 1.2 ilustra este ciclo.

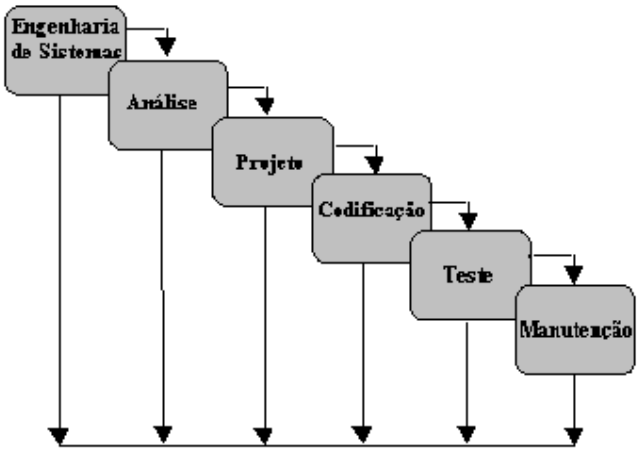

Figura 1.2 – Modelo clássico do ciclo de vida. (PRESSMAN, 1995, p.33).

A manutenção costuma representar um certo custo para as organizações. Na década de 70, a maior parte do orçamento de software era destinada ao desenvolvimento. Já nos anos 80 essa proporção foi revertida, e a manutenção passou a representar entre 40% e 60% do custo total no ciclo de vida do software. As estimativas atuais sugerem que no ano 2000 os custos de manutenção aumentaram em até 80% do custo referente ao tempo de vida de um sistema (PFLEEGER, 2004).

Segundo Seacord et al (2003), pode haver maior dificuldade de manutenção no software, quando há, por muitos anos, mudanças cumulativas no mesmo.

As modificações em sistemas são, em sua grande maioria, inevitáveis. As alterações em leis, modificações na estrutura organizacional de empresas, novas necessidades de informações, entre outras, fazem com que ocorram manutenções nos sistemas. Estas manutenções, no entanto, podem, ao longo do tempo, tornar-se prejudiciais. Na opinião de Seacord, et al (2003), a manutenção do software legado é vista de forma evolutiva enquanto ela apresenta benefícios para o mesmo. Entretanto, quando a tecnologia empregada torna-se obsoleta, as manutenções podem não alcançar as propriedades desejadas.

Para Sommerville (2001), as manutenções podem ser de natureza corretiva, adaptativa, ou perfectiva. As modificações corretivas reparam defeitos no software. A manutenção adaptativa surge quando o ambiente externo (seja ele hardware, sistemas operacionais, ou software de apoio), sofre mudanças, sendo necessário adaptar o sistema a esta nova realidade. Já as manutenções perfectivas destinam-se a fazer acréscimos à funcionalidade do software ou modifica-la, seja em função de requisitos dos sistemas ou mudanças organizacionais de negócio.

Para cada tipo de manutenção é apresentado um certo nível de esforço. Sommerville (2001) apresenta essa divisão conforme figura 1.3.

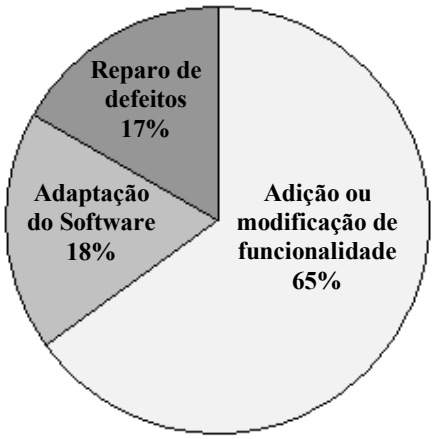

Figura 1.3 – Distribuição do esforço da manutenção. (SOMMERVILLE , 2001, p.518)

"Corrigir erros detectados faz com que uma aplicação original seja modificada e expandida. Os produtos de software também são modificados de forma a acomodar os novos objetivos (mudanças nos requisitos). Como efeito disso, os produtos de software evoluem" (PETERS; PEDRYCZ, 2001, p.568). Entretanto, observa-se que o maior nível de esforço da manutenção concentra-se na adição ou modificação de funcionalidades.

É neste estágio da manutenção, onde adições ou modificações de funcionalidades tornam-se freqüentes e trabalhosas, que deve ser feita uma análise do software para decidir-se sobre o destino do sistema legado.

1.2 A Modernização do Software

Paula Filho (2003) afirma que a engenharia de software se preocupa com o software enquanto produto. Descarta de seu escopo, softwares pequenos que não atendem a um objetivo organizacional.

Partindo-se da ótica na qual os softwares avaliados são vistos como um produto, é interessante que este produto esteja atualizado para garantir sua permanência no mercado. Entretanto, sabe-se que a tecnologia evolui constantemente e de forma rápida. Assim, hardwares e softwares têm uma forte tendência a tornar-se, repentinamente, obsoletos.

Na análise do destino de um sistema legado, é preciso considerar a situação atual do software em relação à empresa e, a partir de algumas estratégias, decidir qual delas é a mais apropriada à condição atual deste software. É preciso considerar, ainda, que tal decisão afeta não só o software legado, mas todo o contexto sociotécnico ao qual esse software está inserido.

Sommerville (2001) apresenta quatro opções quanto ao software legado:

- descartar completamente o sistema,
- continuar a manter o sistema;
- transformar o sistema; ou
- substituir o sistema.

Entretanto, de acordo com Peters; Pedrycz (2001), a partir do princípio de que sistemas legados são sistemas de essencial atuação para a empresa, embora precisem ser atualizados, descarta a possibilidade de exclusão total do sistema. Classificam três escolhas básicas quando se trata da avaliação de sistemas legados:

- atualização;
- re-desenvolvimento; ou
- reengenharia.

A modernização envolve mudanças mais extensivas do que a manutenção propriamente dita, mas conserva uma parcela significativa do sistema existente. Estas mudanças incluem, freqüentemente: reestruturar o sistema, aperfeiçoar a sua funcionalidade ou modificar atributos do software. A modernização do software é requerida quando um sistema legado necessita de mudanças mais drásticas do que aquelas possíveis durante o estágio de manutenção, entretanto tem a finalidade de preservar as regras de negócio. Em virtude de a modernização do software causar impacto nos sistemas de informática que compõe seu sistema sociotécnico, essa constante atualização tecnológica pode representar um desafio (SEACORD, et al 2003).

Os sistemas legados, desde a sua concepção até a sua modernização, apresentam um ciclo de vida. Seacord, et al (2003) apresentam as seguintes etapas: concepção do sistema, manutenção do sistema, modernização do sistema e substituição do sistema. A figura 1.4 ilustra essas etapas dentro de uma linha progressiva de tempo.

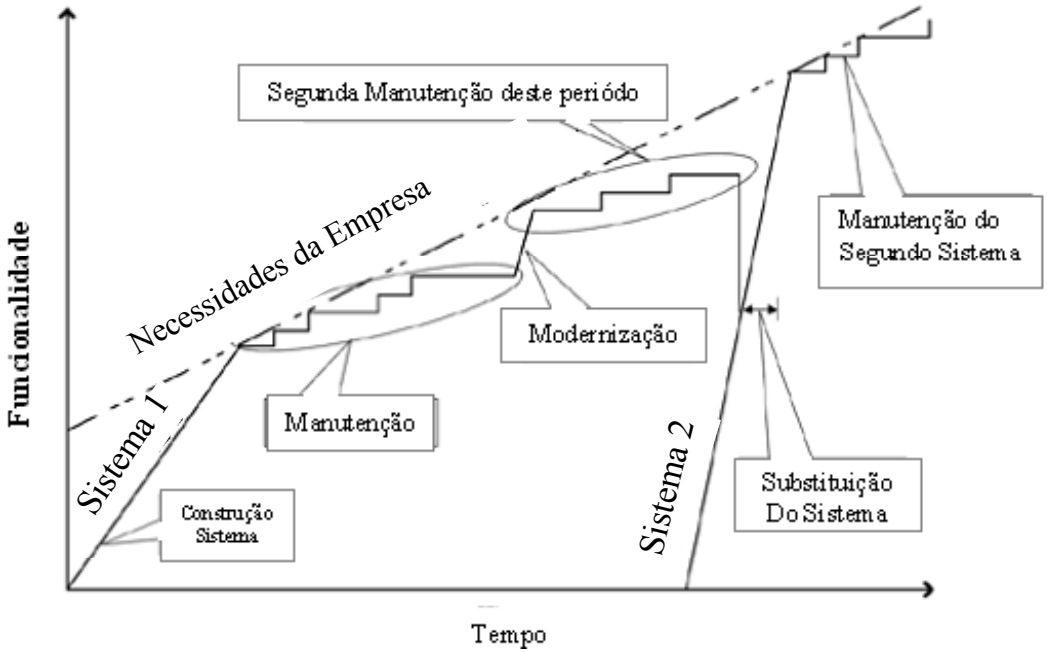

Figura 1.4 – O ciclo de vida da modernização do software (SEACORD, et al, tradução nossa)

1.3 O Dilema dos Sistemas Legados

Em termos de desenvolvimento de software, a utilização de novas tecnologias pode representar um fator de grande importância para o seu sucesso. A tecnologia empregada pode conferir maior credibilidade ao software junto ao mercado, bem como trazer uma maior confiança em termos de desempenho e acessibilidade.

Para a maioria das organizações, a utilização de softwares é um fator de sucesso para a realização de suas atividades. É interessante também que, não apenas os softwares, mas também as ferramentas que compõe o sistema de informática, estejam presentes de forma eficaz (PAULA FILHO, 2003). Tratando-se de informática, a tecnologia evolui com grande velocidade e muitas empresas buscam estar inseridas no contexto tecnológico corrente. Essa atualização de tecnologia refere-se tanto a configurações físicas de seus computadores, como sistemas operacionais e softwares de apoio.

Com a evolução da tecnologia, várias configurações de máquina tornam-se necessárias. As empresas acabam por adquirir computadores com maior capacidade de processamento, maior capacidade de armazenamento de dados, com sistemas operacionais mais modernos, etc. Essa modernização é essencial para atender aos requisitos de softwares e hardwares que vêm surgindo com o passar do tempo. Sem embargo, executar um sistema antigo em uma máquina com uma configuração avançada, pode acarretar em problemas como perda de desempenho, problemas de compatibilidade, entre outros.

Sommerville (2001) comenta que alguns sistemas legados foram construídos para hardwares antigos, onde suas respectivas atualizações, muitas vezes. não estão mais disponíveis ou possuem uma custosa manutenção e esses podem não fazer parte das políticas organizacionais de compras na área de TI. Aos softwares de apoio ou aos sistemas operacionais também pode-se identificar essa característica obsoleta. A estes pode ser difícil encontrar assistência técnica de seus fornecedores originais.

Com essas características de falta de apoio às tecnologias utilizadas, muitas empresas tendem a supervalorizar o desenvolvimento de um novo sistema em detrimento da opção de manutenção.

Maffeo (1992) identifica este processo citado como a crise do software. Ele menciona que esta expressão alude a um conjunto de problemas recorrentemente enfrentados no processo de construção, implantação e manutenção de software. Complementa, ainda, que a crise do software se refere aos problemas relacionados a como os sistemas operacionais são construídos; como sistemas operacionais são implantados; como é provida a manutenção do software; e como atender à demanda crescente de construção de software.

1.3.1 O Desafio da Modernização

No desenvolvimento de software algumas medidas são tomadas para que seus projetos sejam concluídos com êxito, dentro das perspectivas. É recomendado determinar a possibilidade de eventos indesejáveis e fazer planos para evitar esses eventos. Caso estes sejam inevitáveis, é preciso minimizar as conseqüências negativas (PLEEGER, 2004).

Muitos sistemas legados ainda apresentam um alto grau de complexidade. Um dos mais significantes motivos para esta complexidade é o seu tamanho, em número de linhas de código. "Quando um programa em evolução é continuamente modificado, sua estrutura deteriora. Conseqüentemente, a complexidade aumenta, a menos que seja feito um trabalho para mantê-la estacionada ou reduzi-la" (LEHMAN, 1980, apud PFLEEGER, 2004, p. 385). Seacord, et al (2003) caracterizam o tamanho do software como um dos principais fatores prejudiciais à sua evolução. Ainda complementam que esse tamanho em linhas de código tendem a aumentar pelo fato de estas linhas não utilizadas serem pouco tratadas ou eliminadas.

Para Seacord, et al (2003) modernizar um sistema de informática pode ser desafiante para a empresa. A partir de alguns empecilhos, esta modernização pode sofrer alguma resistência até que se efetue. Muitas organizações não conseguem gerenciar os riscos que a modernização pode ocasionar, seja por falta de tempo, pela incompreensão dos mesmos ou, até mesmo, pela falta de planejamento, podendo tornar tardia a atualização de seus sistemas legados.

A modernização de sistemas legados, entretanto, pode ser crucial para o bom desempenho das organizações para com o mercado. Sommerville (2001) sugere que a modernização de sistemas legados é essencial para que as operações corporativas das organizações continuem ativas.

1.4 A Tradução de Códigos Legados

Conforme Martin (1991) sistemas antigos são reconstruídos ao longo dos anos. Geralmente é mais barato reconstruir um sistema do que manter o seu código original. Quase todos os sistemas antigos de uma empresa podem ser reconstruídos, afim de que tenha-se uma empresa com sistemas de informática satisfatórios, capaz de modificar os seus procedimentos com rapidez, objetivando promover a organização perante o mercado.

Na opinião de Feliciano Neto, et al (1988), as otimizações em sistemas podem ser provenientes das próprias manutenções, das atualizações tecnológicas, ou, ainda, em função de novas técnicas de linguagens que surgem no mercado.

Ao tratar-se de atualizações tecnológicas ou novas técnicas de linguagem, a modernização do software pode ser feita através de uma migração de código, onde a linguagem fonte original pode ser substituída por uma linguagem moderna. Nesta migração de código, obter-se-á um novo software, o qual fará parte de uma nova plataforma tecnológica.

1.4.1 Reengenharia

Para atualizar um sistema legado, pode-se aplicar uma reengenharia. A reengenharia é a transformação sistemática de um sistema existente em um nova perspectiva, para que se obtenha melhorias na qualidade, na potencialidade, na funcionalidade, no desempenho e uma economia de custos, de programação ou de riscos ao cliente (TILLEY, 1995 apud Seacord, et al, 2003).

Para Seacord, et al (2003) a reengenharia pode ser mais cara e menos atrativa do ponto de vista técnico, do que a substituição do sistema por um sistema novo. Contudo, embora tenha esses fatores negativos, ela torna-se, na grande maioria das vezes, a alternativa mais pragmática. Ressaltam ainda, que, com a reengenharia pode-se obter uma ampla recolocação do sistema e, dessa forma, tornar-se mais atrativa para as organizações. A reengenharia apresenta custo-benefício satisfatório na perspectiva de aumentar a vida útil do sistema legado

Sommerville (2001) complementa que esta estratégia se difere pelo fato de nenhuma nova funcionalidade ser adicionada ao sistema. O sistema é modificado afim de tornar mais fácil a sua compreensão e alteração. A reengenharia de sistemas pode envolver poucas alterações estruturais, mas geralmente não envolve grandes mudanças em sua arquitetura. Tal

reestruturação efetuada em sistemas legados, segundo Peters; Pedtrycz (2001) traz diversos benefícios como:

- aumento da manutenibilidade do software;
- identificação dos candidatos ao reuso[1];
- valorização do sistema que sofre o processo de reengenharia; e
- aprimoramento da gestão de riscos do projeto.

Turban, et al (2004) salientam que a reengenharia em sistemas legados pode apresentar um custo-benefício satisfatório, quando esta se propuser a converter códigos de programas antigos para um formato mais eficiente. Complementam, ainda, que esta reengenharia torna-se mais eficaz quando a reestruturação dos sistemas legados forem feitas em conjunto com a reestruturação do processo do negócio da empresa.

1.4.2 Tradutores Automáticos de Código

Feliciano Neto, et al (1988) enfatizam a importância de geradores automáticos de aplicação no auxilio da construção ou modernização do sistema. Conforme Martin (1991), é importante a utilização de uma ferramenta automática de geração de programas. Essas ferramentas tendem a trazer maior produtividade e rendimento na tradução de sistemas.

Seacord, et al (2003) comentam que a tradução de um código legado tende a representar riscos, por esta técnica ser valorizada em termos de benefícios, no entanto, seu

---

[1] O reuso do software é o processo de criar sistemas de software baseado em sistemas já existentes, antes que estes apresentem algum tipo de risco (KRUEGER, 1992)

custo por tempo de tradução pode ser alto. Desta forma, sugerem a utilização de uma ferramenta rápida para a tradução de código.

Quando a intenção é atualizar o sistema em termos de tecnologia, esta reengenharia pode apresentar maior eficácia, reduzindo o tempo do processo de reestruturação do sistema, através de um tradutor de código. Este tradutor de código pode ser implementado na forma de um compilador.

O papel de um compilador nesse estágio da evolução do software pode contribuir de forma representativa para a tradução de códigos. Um tradutor de código ocupa-se em aceitar como entrada códigos fontes em determinada linguagem e produzir, como saída, um resultado equivalente a este código, em outra linguagem (GRUNE et al, 2001).

Na visão de Sommerville (2001), "O meio mais simples de proceder a reengenharia de software é a tradução do programa, quando o código-fonte em uma linguagem de programação é automaticamente traduzido para o código-fonte em uma outra linguagem".

Seacord, et al (2003) enfatizam que códigos traduzidos a partir de um tradutor automático costumam apresentar uma padronização, facilitando o seu entendimento. Sommerville (2001) complementa que sua estrutura, em termos de regras de negócio, permanecem inalteradas.

2 COMPILADORES

Compiladores são ferramentas que surgiram no final dos anos 50 para tentar solucionar o problema da complexidade de se escrever programas utilizando a codificação de instruções em códigos binários de uma determinada máquina. (MCCRACKEN, 1984, apud CRESPO, 1998)

Este capítulo versa sobre compiladores e suas características; cita sobre sua estrutura e as fases de sua construção.

Conforme Aho et al (1995) ilustram na figura 2.1, um compilador é um programa que lê um programa escrito em uma linguagem, denominada fonte, e o traduz para um programa equivalente em uma outra linguagem, a denominada alvo.

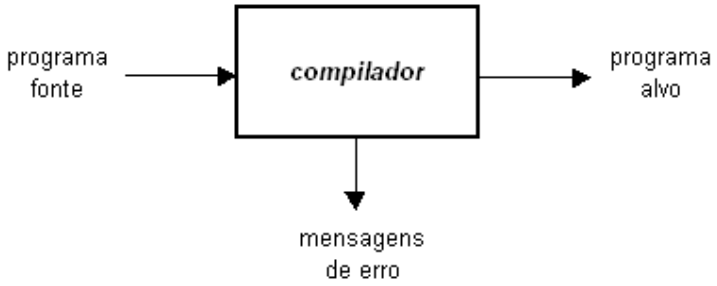

Figura 2.1 – Um compilador. (AHO et al, 1995, p.1)

Os compiladores, geralmente, aceitam um texto escrito em uma linguagem de alto nível e o traduzem para a linguagem de máquina. Mas existem compiladores que aceitam como texto uma linguagem de alto nível e traduzem para outra linguagem de alto nível, para a qual já existem compiladores para código de máquina (CRESPO, 1998).

De forma geral, um compilador lê um texto escrito na linguagem fonte, transformando os caracteres de entrada em unidades reconhecíveis, faz uma análise da estrutura sintática, verifica os requisitos semânticos previstos na linguagem e gera o código alvo. Caso encontre erros de sintaxe ou de semântica, reporta os erros ao usuário procurando apresentar o máximo de informações relativas ao erro (JOSÉ NETO, 1987).

Um compilador divide-se em duas partes: a análise e a síntese. A parte da análise representa as definições do programa fonte, e cria uma representação intermediária do mesmo. A partir desta representação, na fase de síntese, se constrói o programa alvo desejado. Um compilador opera em fases, onde cada uma delas transforma o programa fonte em uma representação para outra. Essa descrição é ilustrada conforme a figura 2.2 (AHO, et al, 1995)

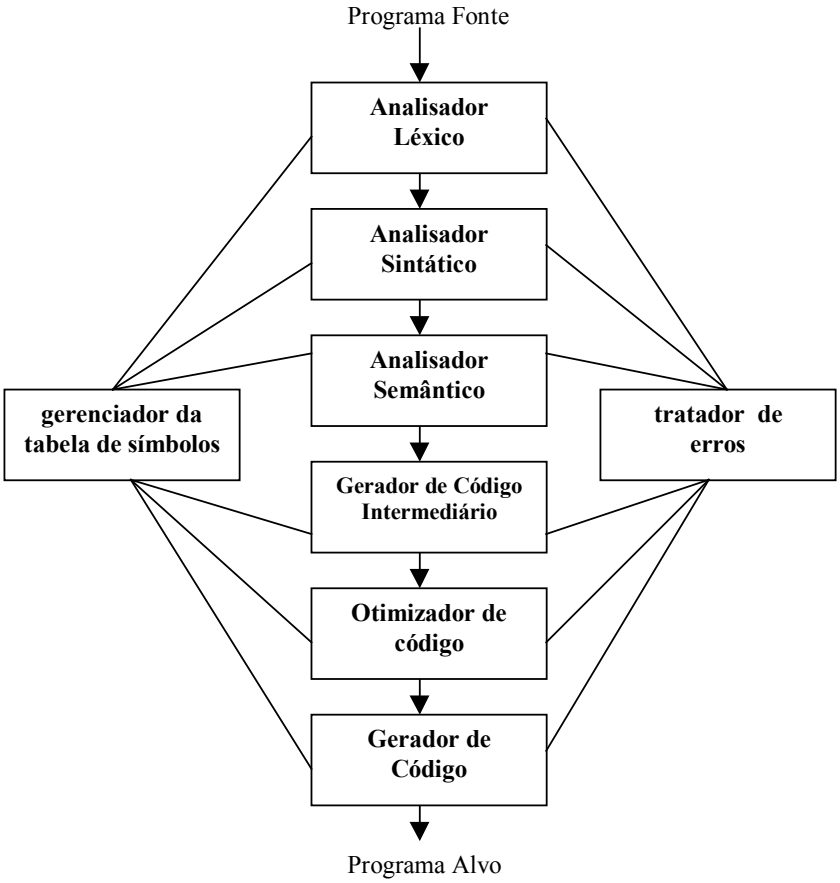

Figura 2.2 – Estrutura de um compilador (AHO, et al, 1995, p.5)

2.1 Gramáticas

À compreensão de compiladores, relaciona-se também o entendimento de linguagens formais. Gramáticas e autômatos são conceitos inseridos neste contexto. Conforme proposto por John Backus e Naom Chomsky, as linguagens são dividia em:

- linguagens irrestritas;
- linguagens sensíveis ao contexto;
- linguagens livres de contexto; e
- linguagens regulares.

Para Crespo (1998), uma gramática BNF[1] é formada por um conjunto finito de regras que definem uma linguagem formal, como no caso de uma linguagem de programação. Além de representar os elementos que constam no alfabeto previsto para a linguagem, a BNF usa de abstrações voltadas à representação das estruturas sintáticas da linguagem.

Uma gramática é um mecanismo para gerar sentenças ou palavras de uma linguagem. Estas gramáticas, segundo Aho, et al (1995), possuem quatro componentes:

- um conjunto de *tokens*[2], conhecidos como símbolos terminais;
- um conjunto de não terminais;
- um conjunto de produções, onde uma produção consiste em um não terminal, chamado de lado esquerdo da produção, uma seta e uma seqüência de *tokens* e/ou não terminais, chamados de lado direito da produção; e
- uma designação a um dos não-terminais como símbolo de partida.

Para reconhecer se uma dada entrada de dados pertence ou não à gramática, pode-se implementar autômatos finitos. Um autômato finito é um grafo direcionado contendo um conjunto de estados intermediários e um conjunto de estados finais representados como vértices deste grafo. Entre alguns dos estados, arestas direcionadas são estabelecidas a fim de representar as transições de estado previstas de acordo com a entrada lida.

Regras de produção definem o mecanismo de geração das sentenças da linguagem. A aplicação sucessiva de regras de produção, a partir do símbolo inicial da gramática, permite derivar as sentenças válidas da linguagem representadas pela gramática. Estes *tokens* derivados constituem a linguagem definida pela gramática. (PRICE; TOSCANI, 2001).

---

[1] BNF - Backus-Naür-Form ou Backus-Normal-Form, implementada para linguagens livre de contexto (GRUNE, et al, 2001)

[2] "Os *tokens* constituem classes de símbolos tais como palavras reservadas, delimitadores, identificadores, etc." (PRICE; TOSCANI, 2001. p.7)

Aho, et al (1995) ilustram uma gramática simples, para avaliação de expressões aritméticas, conforme figura 2.3.

```
expr := expr op expr
expr := ( expr )
expr := - expr
expr := id
op   := +
```

Figura 2.3 – Gramática simples para avaliação de expressões aritiméticas

Outra forma de derivação de uma gramática é através de árvores gramaticais, onde a raiz representa o símbolo inicial, as folhas representam seus símbolos terminais e seus nós representam um conjunto de símbolos não terminais.

Quando sentenças geradas a partir de uma gramática podem ser representadas por mais de uma árvore gramatical, esta gramática é dita ambígua. A ambigüidade é uma propriedade indesejável em linguagens de programação, pois faz com que o analisador sintático tenha mais de um caminho possível, ocasionando *backtracking*[3] (KOWALTOWSKI, 1983)

[3] *Backtracking* – é um procedimento que retorna na árvore, após uma tentativa sem sucesso e desce por outro ramo na busca da solução em uma nova tentativa (RUSSEL; NORVIG, 1995)

Segundo Aho, et al (1995), a árvore de derivação representa gramáticas livre de contexto. Dois tipos de derivação são comumente aplicados: as derivações do tipo LL[4] e as derivações do tipo LR[5].

Uma gramática do tipo LL(k) denota uma leitura do texto-fonte da esquerda para a direita e que, a cada passo de derivação da árvore sintática, o não-terminal mais à esquerda é substituído por um terminal. Desta forma, uma análise sintática LL(k) percorre as etapas de uma derivação mais à esquerda. Uma derivação mais à esquerda se obtém quando deriva-se, a cada passo, sempre o não-terminal que se encontra mais à esquerda. O k entre parêntesis representa o número de símbolos *lookahead*[6] que serão necessários pelo analisador sintático.

Uma gramática LR(k) denota, da mesma forma que na gramática LL(k), que a leitura do texto-fonte acontece da esquerda para à direita. Entretanto, a cada passo de derivação, o que se procura é encontrar é qual a regra da gramática para a qual o lado direito da regra se encontra na entrada lida.

Desta forma, uma análise sintática LR(k) percorre as etapas de uma derivação mais à direita lida de trás para frente, ou seja, em reverso. Uma derivação mais à direita se obtém quando derivamos, a cada passo, sempre o não-terminal que se encontra mais à direita. Da mesma forma, o símbolo k representa o número de *lookaheads* necessários ao analisador sintático.

---

[44] LL – Left-to-Right Scanning, Leftmost Derivation (leitura da esquerda para a direita, com derivação mais à esquerda)
[5] LR – Left-to-Right Scanning, Rightmost Derivation (leitura da esquerda para a direita, com derivação mais à direita)
[6] *lookahead*: símbolos que precisam ser consultados a frente pelo analisador léxico para que se possa tomar uma decisão em termos de análise sintática (AHO; SETHI; ULLMAN, 1995)

2.2 Análise Léxica

Conceitualmente, a análise léxica é a primeira fase de um compilador. Nessa fase é distinguido, no programa fonte, símbolos que precisam ser consultados a frente pelo analisador sintático, para que se possa tomar uma decisão em termos de análise sintática cada símbolo que tenha algum significado. O analisador léxico tem o papel de percorrer o código fonte, caracter a caracter, até formar uma seqüência denominada *lexema*[7], que formam nomes, palavras-chave ou marcadores. Estes são denominados identificadores, ou *tokens*. Durante essa fase da análise são eliminados espaços em branco, caracteres de controle e comentários (CRESPO, 1998).

Em termos de linguagens formais, a análise léxica pretende reconhecer, a partir do código fonte, se os caracteres encontrados formam cadeias presentes em um determinado alfabeto. Este alfabeto é definido, em geral, a partir de uma gramática definida na forma de uma BNF (CRESPO, 1998).

O analisador léxico, conhecido pelo nome de *scanner* é freqüentemente implementado como um autômato finito. Ao mesmo tempo em que elabora uma varredura no código de entrada, caracter a caracter, as transições vão se realizando até que algum estado final seja atingido. Neste ponto, a menos que tenha identificado algum erro léxico, um *token* é reconhecido e enviado ao analisador sintático (AHO, et al, 1995)

Os analisadores léxicos podem ser implementados manualmente, segundo as características desejadas pelo desenvolvedor, ou a partir de ferramentas geradoras de analisadores léxicos como o Lex e o Flex (LEVINE, et al, 1990).

---

[7] lexema: "A seqüência dos caracteres que formam um *token* é chamada o *lexema* para aquele *token*" (AHO; SETHI; ULLMAN, 1995, p.6)

2.3 Analise Sintática

O analisador sintático, também conhecido como *parser*, representa a segunda fase do compilador. De acordo com Aho, et al (1995), o analisador sintático obtém uma cadeia de *tokens* provenientes do analisador léxico, e verifica se a mesma pode ser gerada pela gramática da linguagem fonte. Segundo Kowaltowski (1983), o analisador sintático utiliza-se de métodos para decidir se uma dada cadeia pertence ou não à gramática definida pela linguagem.

A função do analisador sintático é verificar se as construções usadas no programa estão gramaticalmente corretas. Com a gramática correta, o analisador sintático busca descobrir formas de derivação desta gramática. (JOSÉ NETO, 1987)

Enquanto o analisador léxico apresenta a função de identificar *tokens* e os reconhece de forma a serem vistos como uma seqüência de palavras de uma linguagem regular, o analisador sintático consome estes *tokens* e os identifica como parte de uma gramática livre de contexto, ou seja, como uma sentença que satisfaça as regras da gramática (PRICE; TOSCANI, 2001).

As abordagens possíveis de serem efetuadas pela análise sintática são duas: a análise descendente, também conhecida como *top-down* e a análise ascendente, também conhecida como *bottom-up*.

2.3.1 Análise Sintática Descendente

A análise sintática descendente parte da raiz, procurando identificar as regras sintáticas restantes, comparando os símbolos encontrados na entrada com os lados esquerdos das regras da gramática. Cada vez que encontra uma regra, o analisador reconhece os *tokens* invocando outras regras até que todo o texto tenha sido analisado ou alguma entrada não tenha uma regra correspondente ocasionando um erro sintático (AHO, et al, 1995)

Conforme Pyster (1988), a análise sintática recursiva descendente é um método assim chamado porque é implementado por um conjunto de rotinas recursivas, cada uma delas corresponde à cada não-terminal da gramática

Aho, et al(1995) sintetizam que a análise sintática descendente é um tipo de análise que utiliza-se da notação LL. Este tipo de analisador tem como objetivo percorrer a árvore gramatical a partir de sua raiz, criando os nós desta árvore em pré-ordem.

Conforme Watt; Brown (2000) neste tipo de análise, o analisador sintático busca consumir *tokens* do lado esquerdo de cada regra de produção. O Analisador sintático examina os símbolos terminais do texto de entrada, consumindo os mesmos se estiverem conforme a regra analisada. A partir daí, métodos relativos aos não terminais (do lado esquerdo) encontrados são invocados.

Algumas transformações, certas vezes, são necessárias na gramática para que possa ser submetida a um analisador recursivo descendente. A recursão à esquerda não é permitida, logo é necessário que as regras recursivas à esquerda sejam transformadas (JOSÉ NETO, 1987). Almejando-se construir um analisador sintático LL(1), algumas regras também precisam ser fatoradas, de modo a se obter a condição na qual o analisador sintático consiga determinar que regra chamar, olhando apenas o primeiro *token* da entrada.

2.3.2 Análise Sintática Ascendente

Conforme José Neto (1987) e Aho, et al (1995) a análise sintática ascendente parte dos símbolos encontrados na entrada, procurando identificar uma regra que tenha como lado esquerdo a entrada. Enquanto não encontra uma regra cujo lado direito se conforma com a entrada, o analisador vai empilhando a entrada numa operação conhecida como *shift*. Quando a entrada se conforma com o lado direito de uma regra, o analisador substitui o lado direito pelo não terminal que está no lado esquerdo da regra numa operação conhecida como redução (*reduce*). Caso não encontre erros, o analisador termina a compilação ao reduzir a regra inicial.

Conforme Aho, et al (1995) esta análise é conhecida como a análise de empilhar e reduzir. As duas formas mais comuns de implementação da análise sintática são: precedência de operadores e LR. Esta notação constrói uma árvore gramatical para determinada cadeia, tendo seu início pelas folhas em direção à raiz, produzindo derivações mais à direita.

De acordo com Crespo (1998), a análise de precedência de operadores é implementada para gramáticas que possuem a propriedade de não ter nenhuma produção vazia do lado direito da produção, ou que não tenham dois não terminais seguidos.

A análise LR reconstrói uma derivação mais à direita, ao contrário. Possui eficiência em reconhecimento de construções gramaticais de linguagens livres de contexto. Entretanto, sua implementação manual pode ser trabalhosa podendo, neste caso, ser utilizada ferramentas de análise LR, tais como o Yacc ou o Bison (CRESPO, 1998).

Para construção destas tabelas sintáticas, existem três técnicas: LR canônico, SLR, LALR. A técnica SLR (LR Simples) é a mais fácil de implementar. Contudo, é a menos poderosa das três. A técnica LALR tem poder e custo de implementação medianos. O Yacc

utiliza esta técnica. Por fim, a mais poderosa e cara em termos de implementação é a técnica conhecida como LR canônico (AHO, et al, 1995),

2.4 Análise Semântica

A análise semântica ocupa-se em efetuar a tradução propriamente dita, do programa origem para o programa alvo. Esta também é conhecida como classe de ações semânticas. A semântica é a interpretação que pode-se atribuir a todas as sentenças da linguagem (JOSÉ NETO, 1987). As ações semânticas são associadas às regras de produção da gramática, tal que as regras são processadas tão logo tenham sido examinadas pelo analisador sintático.

Segundo Aho, et al (1995) a análise semântica é facilitada pela tradução dirigida pela sintaxe, onde os atributos da gramática dividem-se em dois tipos: atributos sintetizados e atributos herdados. Atributos sintetizados dependem deles mesmos ou de atributos filhos, enquanto atributos herdados dependem de atributos pais ou irmãos.

2.5 Tratamento de Erros

Os compiladores precisam ser capazes de identificar erros durante cada passagem. Erros de programação que não afetam a validade do programa não são identificados (HUNTER, 1987).

Os erros que podem ser identificados no processo de compilação, de acordo com Watt; Brown (2000) são:

- erros léxicos, como os símbolos inesperados ou construções de palavras não previstas;
- erros sintáticos, que são erros que ocorrem quando uma estrutura sintática apropriada não pode ser encontrada para uma entrada analisada; e
- erros semânticos, que são erros encontrados durante a fase da analise semântica, como erros de tipos de variáveis, por exemplo.

Alguns compiladores páram o processo de compilação logo após identificar um erro. Outros, entretanto, continuam seus trabalhos, mesmo após a detecção de um erro, objetivando reportar o maior número possível de erros a cada compilação. Desta forma, reduz-se o tempo dedicado à depuração por parte do usuário.

Para os usuários do compilador, é satisfatório que o mesmo reporte, em sua descrição de erro, o local exato onde esse erro foi detectado, tornar sua correção mais efetiva.

Com o texto fonte não apresentando mais erros de compilação, o compilador gera o código na linguagem alvo.

2.6 Aplicações de Compiladores

Compiladores costumam ser aplicados à conversão de linguagem algorítmica, ou linguagem de alto nível, em linguagem de máquina, afim de que se obtenha a execução do programa a que a linguagem fonte se destina.

A linguagem de máquina não deixa de ser uma linguagem equivalente à algorítmica. Desta forma, a tradução conduzida pelo compilador pode ser feita entre linguagens fontes de alto nível.

Traduções desse tipo se destinam a obter códigos fontes equivalentes de uma linguagem fonte para outra, afim de que a linguagem destino possa ser manipulada pelo seu compilador ou interpretador correspondente.

3 O PROJETO DO COMPILADOR PARA TRADUÇÃO DE CÓDIGO LEGADO

Objetivando modernizar seus sistemas legados escritos em linguagem Clipper, a Quatro Informática, uma *software house* que atende a cooperativas de gestão de saúde, buscou reescrever estes códigos em linguagem Java. Para atender à necessidade de modernização, a proposta adotada foi o desenvolvimento de um compilador para efetivar esta migração de código.

Este capítulo descreve a motivação para a construção e as etapas do projeto deste compilador.

3.1 O Sistema Escrito em Clipper e a Motivação para a Modernização

O sistema Operacional DOS é um software básico que disponibiliza diversos serviços aos aplicativos elaborados para executar nesta plataforma. O Clipper é um compilador compatível com o DOS. Utiliza as funções de baixo nível deste sistema operacional para gerenciar os seus arquivos, em um formato padrão denominado Xbase. Este sistema de arquivos contém as informações manipuladas elaboradas nesta ferramenta (COMPUTER ASSOCIATES, 1992).

Os sistemas legados da Quatro Informática apresentam-se escritos em linguagem Clipper e encontram-se em produção por mais de dez anos. Durante o tempo de sua atuação no mercado este software passou por constantes processos de manutenção. De acordo com Sommerville (2001) e Pfleeger (2004) o maior esforço de manutenção concentra-se na manutenção perfectiva, ou seja, naquela onde são adicionadas funcionalidades.

Proporcionalmente a adições de funcionalidades, expande-se o número de dados manipulados pelo sistema. Em programas escritos na linguagem Clipper, o armazenamento de dados é feito a partir de arquivos DBF. Estes são manipulados através de sistemas de arquivos, e não de sistemas gerenciadores de banco de dados (COMPUTER ASSOCIATES, 1992).

Winck et al (2005) relatam que, dentre as características que demandam uma evolução da plataforma Clipper para outras de tecnologia mais recente, destacam-se:

- a necessidade de gerenciamento dos índices por parte dos aplicativos;
- a falta de controle de restrições de integridade;
- a não existência de mecanismos de controle de transações;
- uma qualidade de apresentação voltada a caracter, ou seja, não orientada ao padrão Windows com janelas e interfaces gráficas; e
- novos sistemas operacionais que não possuem interface DOS.

Este sistema atende, dentro de sua abrangência, as necessidades operacionais de seus clientes. Entretanto, por apresentar as características acima, tal software legado caracteriza-se como um software obsoleto, necessitando de modernização. A proposta de modernizar o sistema busca traduzir este sistema para uma linguagem atual. Para tanto, é preciso analisar a estrutura do sistema escrito em Clipper, analisar novas tecnologias disponíveis e traçar uma estratégia de modernização.

3.2 A Escolha das Novas Tecnologias

Hoje existem várias linguagens e sistemas gerenciadores de banco de dados que suportariam a migração deste código, atuando de forma satisfatória na execução de um novo sistema. Entretanto, a linguagem Java e o SGBD PostgreSQL foram escolhidos para sustentar este novo sistema.

3.2.1 A Linguagem Java

Java é uma linguagem com sintaxe similar à linguagem C. Foi desenvolvida pela SUN em 1995. (DEITEL; DEITEL, 2001). Algumas características como portabilidade, código livre e orientação a objetos foram chaves para a escolha desta tecnologia. A SUN define esta linguagem como "uma linguagem de propósito geral, simples, orientada a objeto, distribuída, interpretada, robusta, segura, de arquitetura neutra, portável, de alta performance, multithread, dinâmica e de acordo com os termos da moda (buzzword=compliant)" (SUN, 1995, apud APPLEBY; VANDELOPPLE, 1997).

A linguagem Java é uma linguagem independente de plataforma, ou seja, seu desenvolvimento permite que seus programas sejam executados em diferentes máquinas, mesmo com diferentes configurações de hardware ou sistemas operacionais (DEITEL; DEITEL 2001). A partir desta portabilidade, a escolha desta ferramenta possibilita maior alcance de mercado. O custo com o desenvolvimento do software e instalação em clientes tornam-se menores por esta linguagem fazer parte da estratégia de código livre. Por ser uma linguagem totalmente orientada a objetos, a estrutura de seus programas possibilita uma melhor manutenibilidade e organização dos mesmos.

Conforme Deitel; Deitel (2001) a melhor forma de construir um programa é desenvolvê-lo em pequenas partes ou em módulos. Os módulos do Java são organizados em métodos e classes, podendo estas serem desenvolvidas ou já fazerem parte de um pacote disponível na *Java API*[1].

A estrutura dos programas em Java é dividida em classes e métodos. Uma classe é contemplada por atributos e por métodos. Os métodos contêm os códigos que manipulam os objetos ou que constroem determinada tarefa. As classes podem ser agrupadas em pacotes. Estes pacotes possibilitam a organização dos programas, modularizando os mesmos. (DEITEL; DEITEL, 2003)

3.2.2 PostgreSQL

De acordo com Momjian (2001), o SGBD PostgreSQL foi desenvolvido na universidade da Califórnia, em Berkeley. Assim como o Java, o PostGreSQL também faz parte da estratégia de código livre, tornando menor os custos com a sua utilização.

---

[1] Conjunto de classes e métodos fornecidos pelo J2SDK (Java 2 Developer´s Kit).

Além desta característica, esta ferramenta oferece, segundo Niederauer (2001), mecanismos eficientes de segurança e integridade de dados, além de suportar a maioria das instruções SQL. É um banco de dados relacional e orientado a objetos. Seus recursos podem ser comparados a bons bancos de dados comerciais.

3.3 O Compilador Clipper2Java

Para atender a aplicação destas novas tecnologias, com maior eficiência e economia de tempo de tradução de código, a proposta adotada é o desenvolvimento de um compilador para auxiliar esse processo. Este compilador recebe um código em linguagem Clipper e produz outro, equivalente, em linguagem Java. Tal compilador, denominado Clipper2Java, é construído utilizando-se da ferramenta Eclipse[2]. É desenvolvido na linguagem Java, em um conjunto de classes voltadas para cada uma das fases de compilação: os analisadores léxico, sintático e semântico.

A conversão do código preserva as regras de negócio originárias, entretanto, a partir de comandos inerentes ao ambiente de execução JClipper[3], sua estrutura procedural é alterada, de forma a acrescentar chamadas de métodos elaborados segundo o paradigma da orientação a objetos, suportada pelo Java.

---

[2] Eclipse é uma plataforma de desenvolvimento, baseada no paradigma de código aberto e construída com a linguagem Java, a qual suporta várias linguagens de programação, como a própria linguagem Java. (ECLIPSE, 2005)

[3] JClipper é uma máquina virtual desenvolvida para simular o comportamento da linguagem Clipper em ambiente Java. Essa ferramenta é descrita na sessão 3.4.

A manipulação de dados, que também tem como base a máquina JClipper, passa a atuar com banco de dados relacionais, servindo-se de uma proposta em três camadas, as quais são:

- a camada de persistência responsável por transações com o banco de dados;
- a camada de negócio, que contém classes responsáveis pelo processamento de consultas, processos e visões; e
- a camada de interface, responsável pela interface com o usuário.

Esta nova estrutura tende contemplar a deficiência de estruturas em camadas suportadas por código legados. A figura 3.1 ilustra um comparativo entre as camadas do software legado e as camadas do novo sistema.

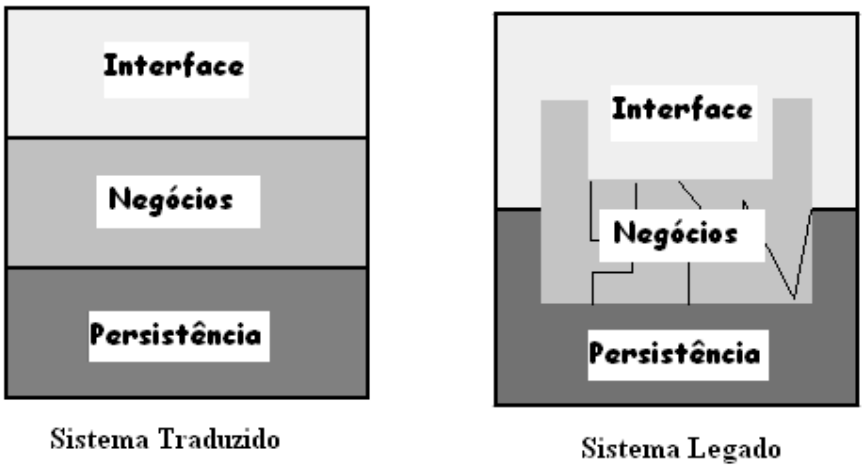

Figura 3.1 - Modelos em camadas do sistema traduzido e do sistema legado – Adaptado de (SOMMERVILLE,2001)

A estrutura do compilador é elaborada em classes Java. A classe principal é denominada ***Clipper2Java***. Esta classe, que implementa um compilador para uma máquina de pilha, é responsável por receber como entrada um código escrito em linguagem Clipper. A partir das linhas de códigos obtidas, são invocadas as demais classes que complementam o compilador. Após o término com sucesso de todas suas etapas é produzido, como saída, um código fonte equivalente em linguagem Java. A figura 3.2 ilustra o diagrama de classes[4] do compilador Clipper2Java

[4] Diagramas de classes mostram um conjunto de classes, interfaces e relacionamentos de um sistema. São utilizados para fazer a modelagem da visão estática do projeto de um sistema (BOOCH, 2000)

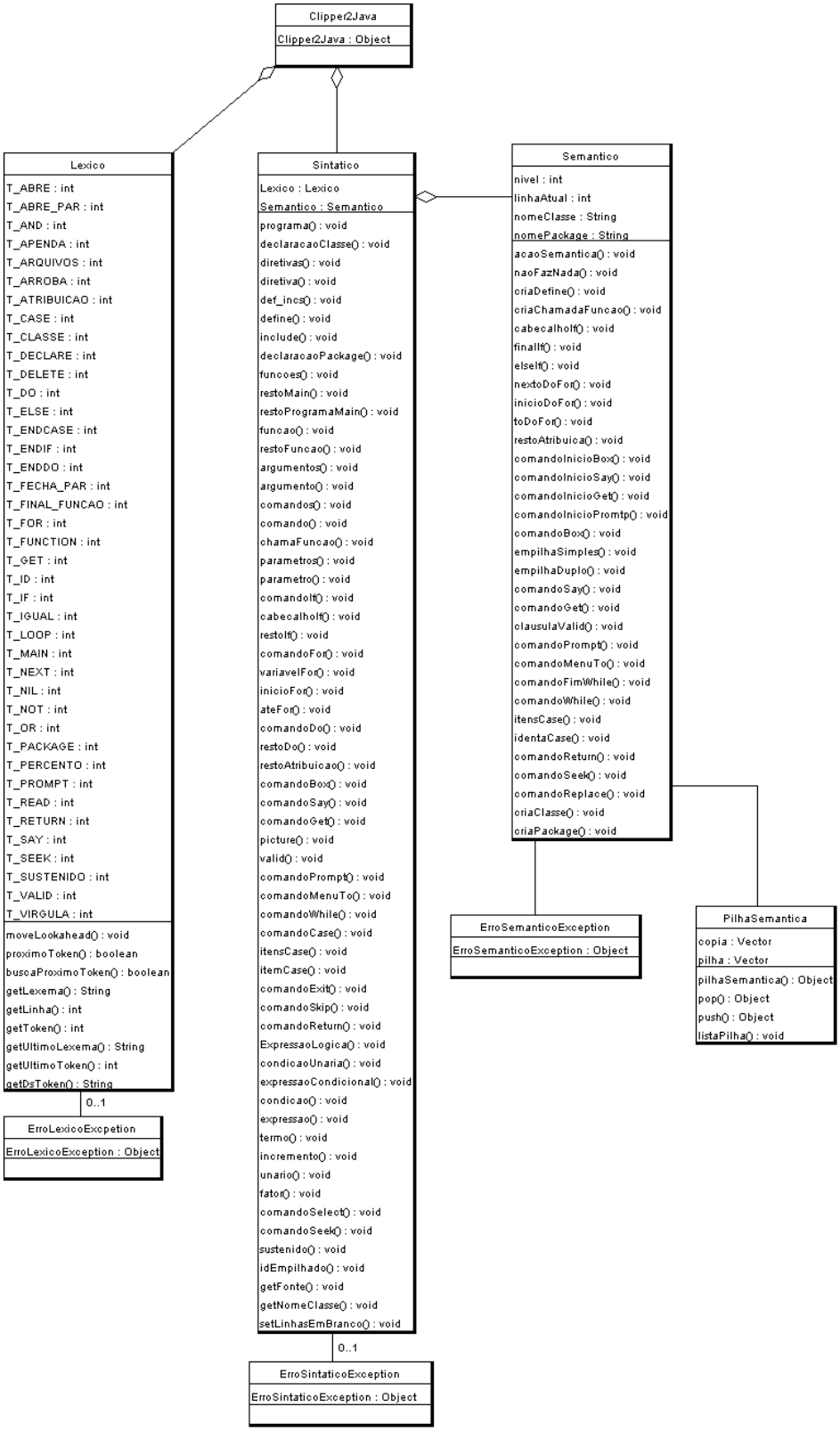

Figura 3.2 – Diagrama de Classes do compilador Clipper2Java

A classe ***Clipper2Java*** é acionada quando o usuário do compilador seleciona um arquivo Clipper para que o mesmo seja traduzido para Java. A figura 3.3 representa este processo através do seu diagrama de caso de uso[5].

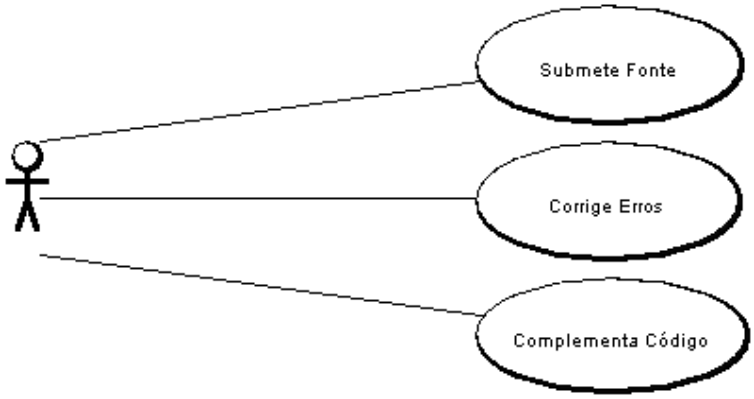

Figura 3.3 – Diagrama de caso de uso do compilador Clipper2Java

A classe ***Clipper2Java*** instancia as classes ***Lexico*** e ***Sintatico***. A última, por sua vez, instancia a classe ***Semantico***. Cada parte que representa o conjunto de classes do compilador é dotada de sua classe principal e uma classe de tratamento de erros correspondente. Estas classes de tratamento de erros são responsáveis por informar a ocorrência de alguma imperfeição encontrada no código, a qual não permite que o processo de varredura seja concluído. Para cada ocorrência de erro é informada, em uma tela de saída, a linha correspondente ao erro no código Clipper, o qual descreve sua possível causa. A figura 3.4 mostra a seqüência do compilador através do seu diagrama de colaboração[6].

---

[5] Diagramas de caso de uso modelam o comportamento de um sistema (BOOCH, 2000)
[6] Diagramas de colaboração são empregados para especificar a realização de casos de uso e operações, e para fazer a modelagem de mecanismos significativos da arquitetura do sistema

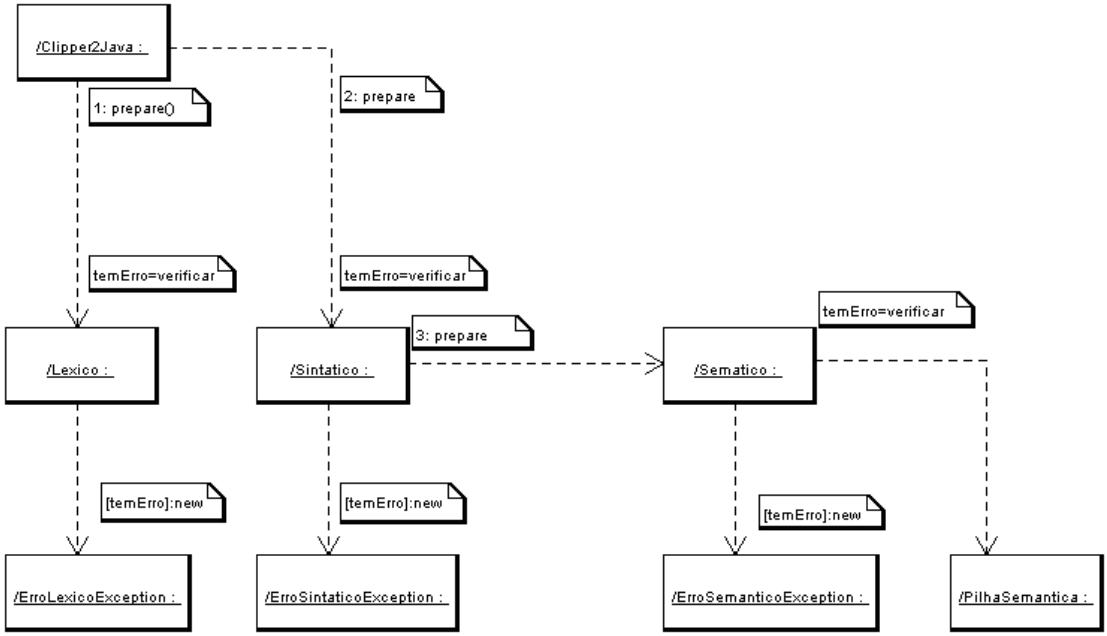

Figura 3.4 – Diagrama de Colaboração do compilador Clipper2Java

O Clipper2Java é um compilador de simples passagem. Dessa forma, a cada informação de erro, o processo é abortado. A partir desta situação, cabe ao operador do compilador analisar o código Clipper, corrigí-lo de acordo com sua necessidade e dar um novo início ao processo de compilação deste código.

3.4 A Máquina JClipper

A fim de aproveitar ao máximo a estrutura que a linguagem Java oferece, usufruindo de sua orientação a objetos e de sua aplicação voltada a uma estrutura em camadas, é desenvolvida uma máquina virtual que simula o comportamento da linguagem Clipper em um ambiente orientado a objetos, contemplado pela linguagem Java.

Para facilitar a execução dos programas escritos originalmente em Clipper, no ambiente Java, é desenvolvido um interpretador de comandos Clipper, denominado JClipper.

Trata-se de um ambiente operacional, dotado de classes capazes de executar métodos Java equivalentes às funções e comandos da linguagem Clipper. A figura 3.5 descreve seu diagrama de classes.

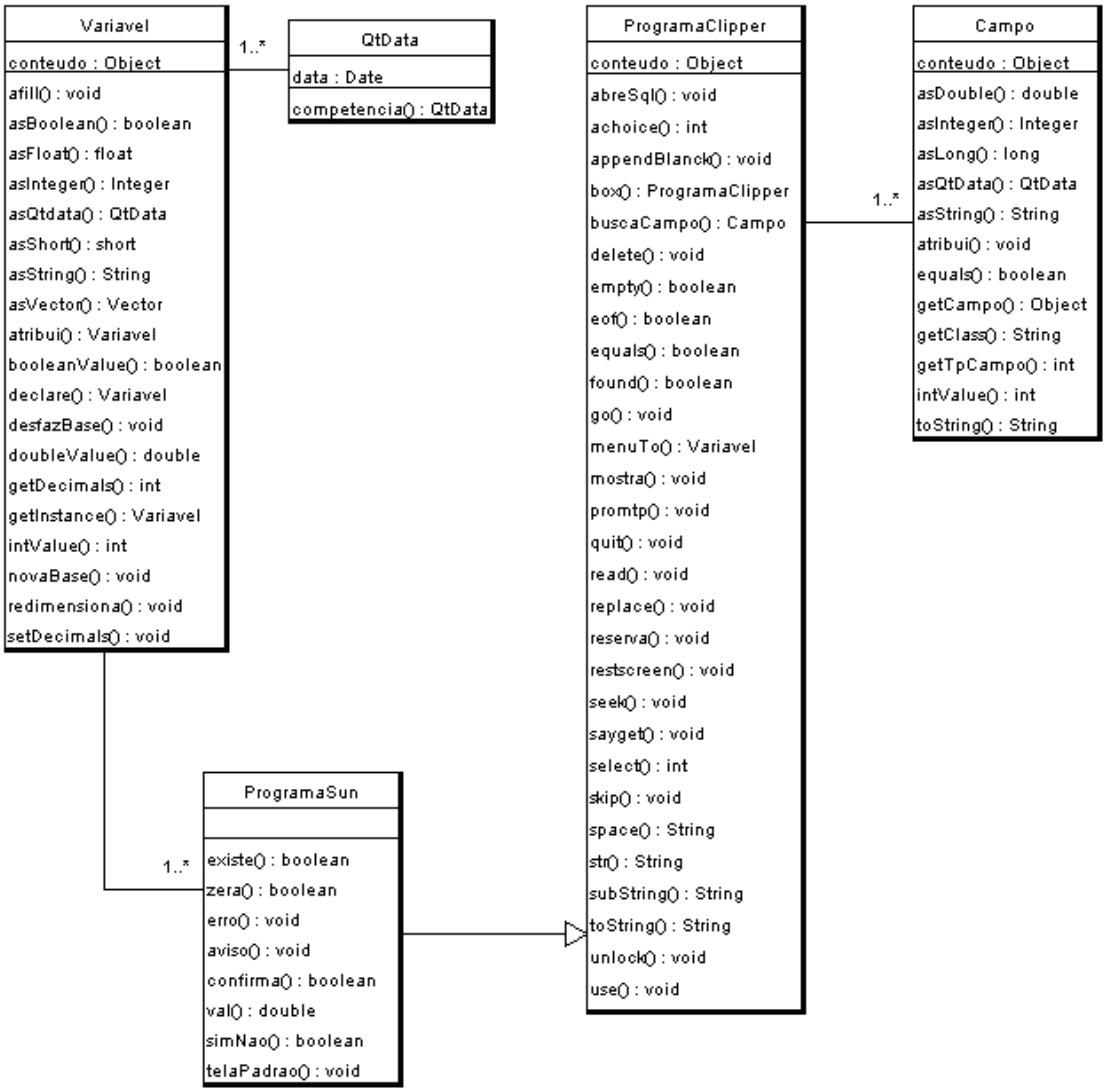

Figura 3.5 – Diagrama de classes da máquina JClipper

O desenvolvimento deste ambiente operacional se deu para suprir a carência de funções Java equivalente a funções provenientes do Clipper. Esta aplicação é responsável, também, por suportar a tipificação de dados, ausente na linguagem Clipper.

No entanto, para obter-se efetividade no ambiente, é necessário que os programas Clipper sejam, primeiramente, convertidos para programas escritos em linguagem Java que, por sua vez, utiliza-se desse ambiente operacional. A arquitetura deste ambiente operacional pode ser vista conforme mostra a figura 3.8.

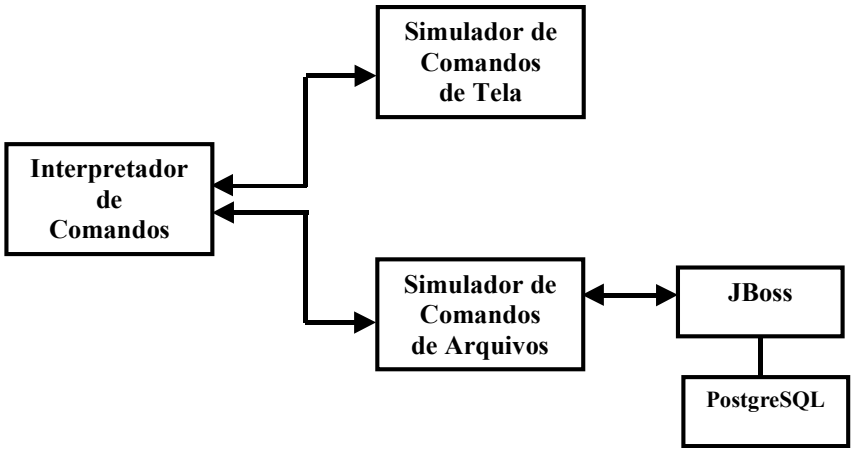

Figura 3.8. A Arquitetura do ambiente JClipper

As principais partes que compõe o ambiente operacional JClipper são:

- interpretador de comandos,
- simulador de comandos de tela; e
- simulador de comandos de arquivos.

3.4.1 Interpretador de Comandos

O interpretador de comandos é o módulo executor, composto das chamadas de métodos correspondentes aos comandos Clipper.

Quando as funções que representam comandos são de tratamento de data ou tratamento de *strings*, os métodos serão resolvidos neste mesmo módulo. Quando os comandos são relacionados a ações de tela, estes são redirecionados ao simulador de comandos de tela. Se os comandos tiverem relação com arquivos e base de dados, estes serão, então, redirecionados ao simulador de comandos de arquivos.

3.4.2 Simulador de Comandos de Tela

O simulador de comandos de tela é um módulo dotado de classes herdadas da *package* Swing, e que procura imitar uma tela característica da linguagem Clipper.

A *package* Swing oferece uma variedade muito grande de objetos de interface que são utilizados, como *ComboBox*, *ListBox*, entre outros, (DEITEL; DEITEL, 2003) imitando uma interface caracter. O aparente retrocesso à uma interface do tipo caractere pode ser justificada pelo fato que este ambiente deverá servir somente durante a transição, para automatizar a conversão de um grande número de programas, com um menor custo possível.

Uma vez feita a transição, os programas poderão, paulatinamente, ter sua interface convertida para o uso de todos os recursos que a API Swing[7] oferece.

3.4.3 Simulador de Comandos de Arquivos

O módulo simulador de comandos de arquivos recebe comandos de manipulação de arquivos típicos da linguagem Clipper, e os converte para chamadas de classes escritas em EJB Container[8] JBoss. Estas classes são responsáveis pela conversão dos comandos para comandos SQL que simulam o comportamento de funções Clipper.

Este módulo é o de mais difícil conversão, não pelas dificuldades propriamente ditas, mas por questões de performance. Quedas muito significativas de performance podem ser tratadas caso a caso.

---

[7] API Swing – API que permite configurar layout de aplicação (DEITEL; DEITEL, 2003)
[8] EJB Container – Enterprise JavaBeans Container – objetos Java instanciados remotamente, de forma transparente ao cliente (AMBLER, JEWELL, 2002)

3.5 Construção da Ferramenta

Os modelos que descrevem o compilador Clipper2Java, assim como o ambiente JClipper, são um protótipo para o desenvolvimento da ferramenta que permitirá traduzir códigos para Java, a partir da linguagem Clipper.

A ferramenta desenvolvida basear-se-á nos métodos propostos no modelo, integrando os métodos do JClipper, para obter sucesso na conversão do código, bem como nas aplicações geradas por estes códigos.

4 A IMPLEMENTAÇÃO DO COMPILADOR

A partir do modelo projetado para o compilador, utilizando-se seus métodos, bem como os métodos que compõe a ferramenta JClipper, dá-se início à implementação de código do Clipper2Java.

É elaborada uma gramática, a qual serve de base para a implementação de regras das classes que compõe o compilador, obedecendo a lógica de construção de compiladores, denominada de tradução dirigida pela sintaxe (AHO et al, 1995)

Este capítulo descreve as etapas de desenvolvimento do compilador Clipper2Java, assim como as classes que o compõe.

4.1 A Gramática

Para a implementação do compilador, foi projetada uma gramática do tipo LL(1), contendo regras voltadas ao ambiente de execução JClipper. Esta gramática passa pelos processos de fatoração e remoção de transições vazias.

A gramática contempla as regras da linguagem Clipper utilizada pelos sistemas legados e suas regras da biblioteca de funções. Algumas regras desta gramática são apresentadas conforme ilustra a figura 4.1.

```
[405] <expressao>  ::=  <expressao> '+' <termo>
[410]              |    <expressao> '-' <termo>
[412]              |    <expressao> '$' <termo>
[415]              |    <termo>
[420] <termo>      ::=  <termo> '*' <incremento>
[421]              |    <termo> '/' <incremento>
[422]              |    <incremento>
[423]              |    <termo> '%' <incremento>
[425] <incremento> ::=  '++' <unario>
[426]              |    '--' <unario>
[427]              |    <unario> '++'
[428]              |    <unario> '--'
[429]              |    <unario>
```

Figura 4.1 – Exemplos de regras de produção da gramática elaborada para o compilador

A gramática apresenta um número único para cada regra de produção. Este número é utilizado para endereçar com maior facilidade as ações semânticas do analisador semântico. A cada regra semântica que é reconhecida, o analisador semântico é invocado, passando-se como parâmetro o número desta regra.

Os símbolos não-terminais desta gramática encontram-se entre os sinais '<' e '>'. Quaisquer símbolos que não estejam entre tais sinais, representam os símbolos terminais e devem ser assumidos como *tokens*.

A gramática construída para o compilador é uma base para as funções que são utilizadas pelo ambiente de execução JClipper. Esta gramática dá respaldo para a implementação dos analisadores léxico, sintático e semântico.

4.2 A Classe Lexico

Optou-se por desenvolver manualmente o analisador léxico, na forma de uma classe Java denominado ***Lexico***. A classe ***Lexico*** define uma lista de *tokens* e métodos voltados para percorrer o código fonte, identificando *lexemas* e classificando os mesmos em *tokens*. Para os possíveis erros reconhecidos durante a fase de análise léxica, é implementada uma classe ***ErroLexicoException***.

Na classe ***Lexico***, é definida uma constante do tipo inteiro para cada *token* pertencente à gramática. Algumas definições são visualizadas conforme figura 4.2.

```
public static final int T_FUNCTION          = 1;
public static final int T_ABRE_PAR          = 2;
public static final int T_FECHA_PAR         = 3;
public static final int T_RETURN            = 4;
public static final int T_IF                = 5;
public static final int T_ENDIF             = 6;
public static final int T_ELSE              = 7;
public static final int T_FOR               = 8;
public static final int T_STEP              = 9;
public static final int T_TO                = 10;
public static final int T_DO                = 11;
public static final int T_IGUAL             = 12;
public static final int T_ATRIBUICAO        = 13;
public static final int T_MAIS_IGUAL        = 14;
public static final int T_MENOS_IGUAL       = 15;
public static final int T_VEZES_IGUAL       = 16;
public static final int T_SOBRE_IGUAL       = 17;
public static final int T_PERCENTO          = 19;
public static final int T_ARROBA            = 20;
public static final int T_VIRGULA           = 21;
```

Figura 4.2 – Exemplo de definições de *tokens* na classe ***Lexico***

É implementado um construtor que recebe o código fonte em linguagem Clipper, eliminando caracteres em branco e marcadores, percorrendo caractere a caractere, até que um *lexema* seja formado. A partir deste *lexema*, busca-se identificar um *token* reconhecível. Caso o *lexema* não seja identificado como um *token*, a classe ***ErroLexicoException*** é invocada e o processo é abortado. Estas regras são implementadas a partir do método ***buscaProximoToken***. A figura 4.3 apresenta o início da implementação desta regra.

```
private boolean buscaProximoToken() {
    ultimoToken  = token;
    ultimoLexema = lexema;
    lexema = "";
    StringBuffer esteToken = new StringBuffer();

    while( brancos.toString().indexOf( lookahead ) >= 0 ) {
        moveLookahead();
    }
    if( isAlpha( lookahead ) ) {

        while( isAlpha( lookahead ) ||
               isDigit( lookahead ) ||
               isPonteiro( lookahead ) ) {
                esteToken.append( lookahead );
                moveLookahead();
        }
        lexema = esteToken.toString().toUpperCase();

        if( lexema.equals( "FUNCTION" ) ) {
            token = T_FUNCTION;
        } else if( lexema.equals( "RETURN" ) ) {
            token = T_RETURN;
        } else if( lexema.equals( "IF" ) ) {
            token = T_IF;
        } else if( lexema.equals( "ENDIF" ) ) {
            token = T_ENDIF;
        } else if( lexema.equals( "ELSE" ) ) {
            token = T_ELSE;
        } else if( lexema.equals( "FOR" ) ) {
```

Figura 4.3 – Início do método ***buscaProximoToken***

4.3 A Classe Sintatico

O método de análise sintática escolhido foi o da Análise Sintática Recursiva Descendente, por se tratar de um método simples, cujo fonte é compreensível, de modo a facilitar alterações localizadas. A classe ***ErroSintaticoException*** acompanha a classe ***Sintatico***, implementada para o tal analisador.

Para cada *token* reconhecido no analisador léxico, o mesmo é identificado pelo analisador sintático, dentro de uma seqüência de *tokens* coerentes com a gramática. São atribuídos métodos para consumir estes *tokens*. A cada *token* consumido, busca-se um próximo. Caso esse próximo *token* não seja correspondente aos *tokens* definidos na gramática para a regra analisada, é disparado um erro através da classe ***ErroSintaticoException***, e o processo é abortado.

Na ausência de erro, a cada *token* consumido, uma regra semântica é chamada. É nesse momento em que observa-se a importância de cada regra de produção da gramática ser identificada por um número distinto. Desta forma, a regra semântica invocada é correspondente ao número da regra de produção da gramática. A figura 4.4 ilustra a implementação da análise sintática para o método ***expressao***, inserida na gramática.

```
private void expressao() {


    termo();
    semantico.acaoSemantica( 415 );

    while ( lexico.getToken() == Lexico.T_MAIS ||
            lexico.getToken() == Lexico.T_MENOS ||
            lexico.getToken() == Lexico.T_DOLAR ) {

        if ( lexico.getToken() == Lexico.T_MAIS ) {
            lexico.proximoToken();
            termo();
            semantico.acaoSemantica( 415 );
            semantico.acaoSemantica( 405 );
        } else {
            if ( lexico.getToken() == Lexico.T_MENOS ) {
                lexico.proximoToken();
                termo();
                semantico.acaoSemantica( 415 );
                semantico.acaoSemantica( 410 );
            } else {
                lexico.proximoToken();
                termo();
                semantico.acaoSemantica( 415 );
                semantico.acaoSemantica( 412 );
            }
        }
    }
}
```

Figura 4.4 – Método que implementa a análise sintática para a regra ***expressao***

4.4 A Classe Semantico

A análise semântica serve-se de uma gramática de atributos para estabelecimento das regras semânticas. Estas regras são invocadas a partir do reconhecimento, pelo analisador sintático, das regras sintáticas. Para o armazenamento dos atributos necessários à geração do código, é construída uma pilha que armazena regras semânticas até que toda a ação semântica esteja completa.

Nesse momento, então, é gerado em arquivo texto, dentro de uma formatação adequada de programação Java, um conjunto de linhas de código Java correspondentes a esta regra semântica. O método ***fazIdentacao,*** acompanhado da variável ***nível***, identa o código de acordo com os valores informados em cada método de regras semânticas.

A classe ***Semantico*** constitui de um método denominado ***acaoSemantica***, o qual recebe como parâmetro o número de sua regra. Este método chama o método correspondente a cada regra, conforme mostra a figura 4.5.

```
public void acaoSemantica( int regra ) {

    if ( regra != 1000 ) {
        linhaAtual = lexico.getLinha();
    }

    switch ( regra ) {
        case 2      : criaClasse(); break;
        case 10     : pilhaSemantica.push( lexico.getUltimoLexema(), regra ); break;
        case 11     : criaDefine(); break;
        case 12     : naoFazNada(); break;
        case 14     : criaPackage(); break;
        case 15     : criaDefineString(); break;
        case 17     : criaMetodosChamadores(); break;
        case 20     : naoFazNada(); break;
        case 21     : naoFazNada(); break;
        case 22     : criaFinzinhoFuncao( 22 ); break;
        case 23     : criaMain(); break;
        case 24     : naoFazNada(); break;
        case 25     : pilhaSemantica.push( lexico.getUltimoLexema(), regra );
```

Figura 4.5 - Parte do método ***acaoSemantica*** que invoca métodos correspondentes a cada ação semântica.

A figura 4.6 ilustra os métodos ***cabecalhoIf***, ***finalIf*** e ***elseIf***, que implementam as ações semânticas da regras que compõe as instruções do comando ***if***.

```
private void cabecalhoIf() {
    String c1, ct1;
    c1 = pilhaSemantica.pop().toString();

    ct1 = c1;

    fazIdentacao();
    fonte.append( "if ( " );
    fonte.append( ct1 );
    fonte.append( " ) {\n" );
    nivel++;
}

private void finalIf() {
    nivel--;
    fazIdentacao();
    fonte.append( "}\n" );
}

private void elseIf() {
    nivel--;
    fazIdentacao();
    fonte.append( "} else {\n" );
    nivel++;
}
```

Figura 4.6 – Método que implementa o conjunto de regras semânticas do comando ***if***

4.4.1 Métodos da Análise Semântica

A análise semântica é implementada com métodos Java correspondentes a funções da linguagem Clipper. Devido à diferenças entre paradigmas destas duas linguagens, muitas funções Clipper não possuem correspondência direta ao Java. Assim, a implementação destas correspondências tornou-se necessária.

Evitando códigos muito extensos para determinados métodos, e buscando a equivalência entre os códigos, utiliza-se o ambiente operacional JClipper para implementar tais métodos.

4.5 Adição de Novos Métodos

A elaboração da gramática, bem como o desenvolvimento do compilador teve início com um mapeamento das principais funções utilizadas pelo Clipper, e a elaboração de sua gramática.

Entretanto, é a partir da utilização do compilador que sua implementação torna-se mais refinada. A importação de códigos Clipper complexos, que utilizam-se de uma vasta biblioteca de funções, reportam alguns erros onde observa-se a necessidade de aperfeiçoamento do Clipper2Java e da máquina JClipper.

5 A UTILIZAÇÃO DO COMPILADOR CLIPPER2JAVA

Para a implementação do compilador buscou-se obter praticidade para a tradução dos códigos. Em virtude disto, evitou-se a implementação de métodos complexos que acarretariam em um grande consumo de tempo. No que tange esta característica, o compilador não estaria apto a receber um código legado da maneira como ele é utilizado pelo Clipper, e traduzi-lo completamente para Java.

Desta forma, optou-se pela inserção manual de alguns identificadores no código Clipper e, posteriormente à sua tradução, pela correção de possíveis falhas de sintaxe no código Java. Este capítulo aponta tais modificações, para a correta utilização do Clipper2Java.

5.1 Alterações no Código Clipper

Ao Compilador cabe identificar corretamente as funções do código Clipper para que possa gerar um código Java com as mesmas ações semânticas. Contudo, algumas incompatibilidades são encontradas entre essas duas linguagens. Isto se deve a questões de sintaxe, semântica e paradigmas distintos.

A fim de obter-se um código Java completo, em termos de sua estrutura, essas divergências deveriam ser tratadas e reconhecidas pelo compilador. No entanto, as implementações destas regras tornar-se-iam complexas. Buscando a construção de um código simples, optou-se por inserir, de forma manual, identificadores no código Clipper, os quais seriam reconhecidos como *tokens* pelo compilador, e seriam tratados por suas respectivas regras sintáticas e semânticas.

5.1.1 Identificador de Pacote

Os sistemas legados aqui descritos são organizados em módulos, onde cada módulo contém diversos programas que se interagem ou que tem relação entre si. Em Clipper, os módulos são definidos agrupando-se programas que fazem parte destes módulos em arquivos com extensão *rmk*[1] e *lnk*[2], respectivamente arquivos de lote para compilação e linkedição do programa.

Esta organização em módulos, com a conversão do código, passará a ser disposta em pacotes. Conforme Deitel; Deitel (2003), pacotes são um mecanismo de agrupar classes com finalidades relacionadas. Os pacotes facilitam a organização conceitual das classes e permite localizar cada classe necessária durante a execução da aplicação. Além desta perspectiva, pode-se ter classes com os mesmos nomes, porém em pacotes diferentes. Cada programa Java recebe o registro a que pacote pertence. Assim, obter-se-á um respectivo pacote para cada módulo do sistema Clipper.

---

[1] Ações a serem realizadas (COMPUTER ASSOCIATES, 1992)
[2] Arquivos a serem linkeditados (COMPUTER ASSOCIATES, 1992)

Buscando uma melhor performance de tradução, optou-se por inserir um identificador de pacote ao código a ser importado. Assim, cabe ao programador consultar o módulo a que determinado programa pertence, e inserir tal identificador, na primeira linha do código, com os seguintes itens:

- sustenido (#);
- a palavra ***package;***
- espaço em branco; e
- o nome do pacote

5.1.2 Identificador de Classe

Aplicações Java são dispostas em classes, onde classes são uma especificação que engloba variáveis e métodos comuns (DEITEL; DEITEL, 2003). As classes Java devem ter o nome de forma idêntica ao da classe principal.

Desta forma, de modo a seguir uma padronização e obter-se uma fácil identificação de classes, o nome do programa original é preservado. Portanto, optou-se pela inserção de um identificador de nome de classe, onde o programador o informa na segunda linha do código, contendo o nome do programa correspondente a este código. Tal identificador é composto da seguinte forma:

- sustenido (#);
- a palavra ***classe***;
- espaço em branco; e
- o nome da classe.

5.1.3 Identificador do Método main

Os programas em Clipper são organizados em funções onde, em sua grande maioria, a primeira função do código representa a sua função principal. Não restrito a isso, existem códigos que são apenas funções independentes que, por sua vez, são chamadas por outras funções contidas no mesmo ou em outro programa.

Em Java, classes contendo um método *main* são conhecidas como aplicações nestas classes. O método *main* determina o ponto de início de execução desta aplicação (DEITEL; DEITEL, 2003).

Em virtude desta característica, é preciso focar no código Clipper qual função é, e se esta é, a função principal do código fonte. Neste caso, o programador avalia o código como um todo e identifica no mesmo a sua função principal. Desta forma, quando um método *main* se faz necessário no programa a ser traduzido, inclui-se um identificador deste método, junto à função principal do Clipper, da seguinte forma:

- a palavra reservada ***function*** (já disposta no código Clipper);
- espaço em branco;
- a palavra reservada ***main*** (a ser inserida pelo programador);
- espaço em branco; e
- o nome da função (já disposta no código Clipper).

5.1.4 Identificador de Final de Função

Funções Clipper são finalizadas com a palavra reservada ***return***, seguida do valor que a função está retornando ou da palavra ***nil***, que faz com que a função não retorne nada. Entretanto, as funções podem, antes de terminar, retornar algum valor, onde este retorno também é identificado com a palavra ***return***.

Com essa característica, seria difícil implementar uma regra ao compilador que interpretasse se o *token* ***return*** identificado em determinada função representa, ou não, o final desta função. Buscando-se tornar a identificação de final de função mais fácil, introduz-se no código Clipper, a cada final de função, uma linha com os seguintes dados:

- percentual (%); e
- a palavra ***fim***

5.2 Código a Ser Traduzido

Os identificadores descritos anteriormente representam as premissas básicas para que o compilador interprete o código Clipper como um código válido. A figura 5.1 ilustra um exemplo de código Clipper que sofreu tais alterações.

```
#package sigedu
#classe SigEdu

Function main SigEducativo()

    TelaPadrao( "Modulo : Comite Educativo" )

    poe_sombra( 2, 1, 20, 38, 8 )
    @  2, 1, 19, 36 BOX SIMPLES

    janela := savescreen( 0, 0, 24, 79 )

    do while .t.
         @  3, 2 prompt "Gera Estatistica Com. Ed. do mes  "
        @  4, 2 prompt "Emite Estatistica Com. Ed. do mes"
        menu to Opcao

        do case
        case Opcao == 0
            exit

        case Opcao == 1
            if VerifPerm( "SIG", "A" )
                GeraEstEdu()
            else
                loop
            endif

        case Opcao == 2
            if VerifPerm( "SIG", "B" )
                EmiEstEdu()
            else
                loop
            endif

        endcase

        restscreen( 0, 0, 24, 79, janela )

    enddo

    P_TelaFim()
    close all
    quit


Return NIL
%fim
```

Figura 5.1- Exemplo de código Clipper alterado para a importação

Com estas adições de identificadores inseridas no programa, tem-se um código apto a ser reconhecido e importado pelo Clipper2Java.

5.3 Importação do Código Clipper

Ao percorrer o código Clipper alterado com seus devidos identificadores, o compilador aciona cada uma das fases de compilação, gerando sua respectiva semântica em Java. O Clipper2Java possui uma interface simples, onde é possível selecionar o arquivo que deseja-se traduzir, conforme ilustra a figura 5.2.

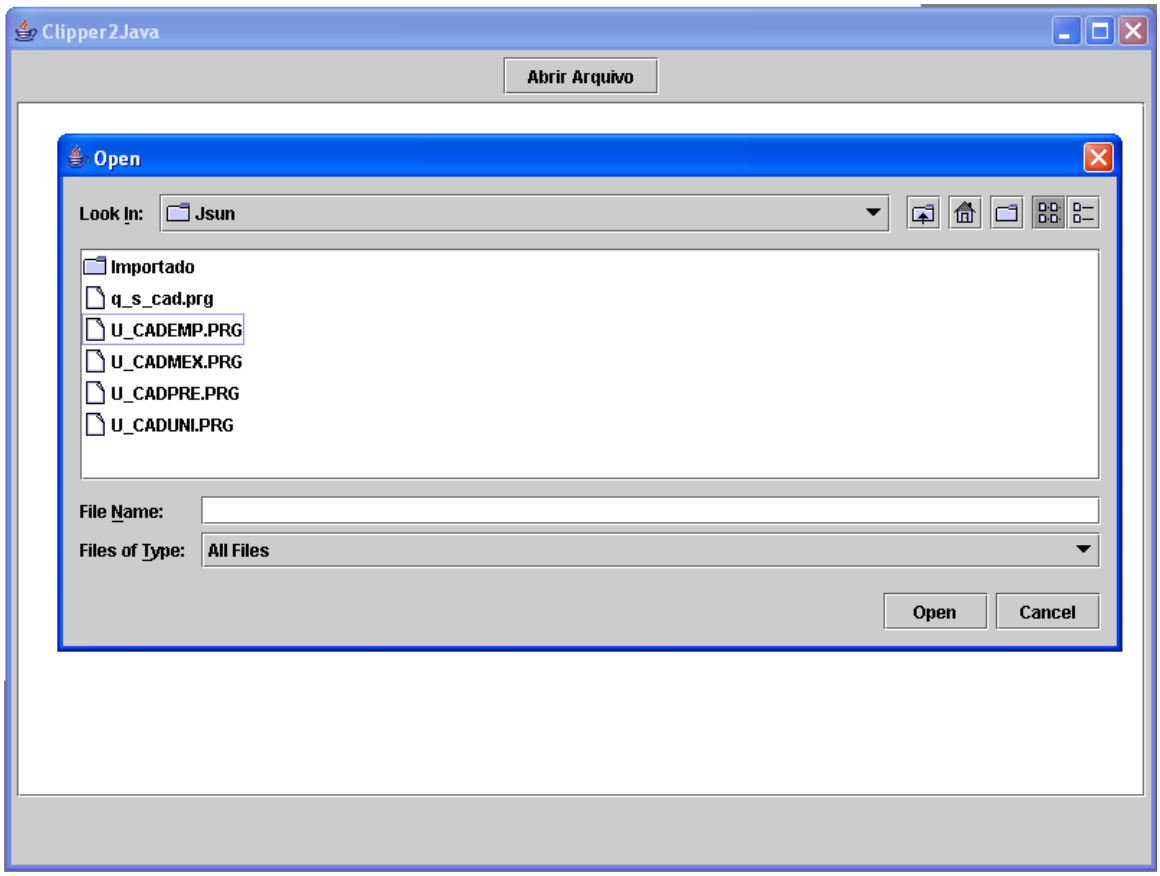

Figura 5.2 – interface do Clipper2Java

Durante a importação do código, o compilador está apto a identificar e exibir possíveis erros de sintaxe encontrados no código para que o programador possa corrigí-los. Estes erros são exibidos no *console*[3] da plataforma Eclipse, conforme figura 5.3.

```
Problems   Console ✕   Search
Clipper2Java [Java Application] C:\Arquivos de programas\Java\j2re1.4.2_05\bin\javaw.exe (26/10/2005 09:54:37)
------------------------
| 6 | - Variavel.asString( "empres" ) pela regra:  810
| 4 | - #chamada_funcao# pela regra:  184
| 2 | - existe pela regra:  800
------------------------
ErroSintaticoException: Eu esperava encontrar aqui ')', mas encontrei:
 na linha: 205
        at Sintatico.chamaFuncao(Sintatico.java:477)
        at Sintatico.fator(Sintatico.java:1446)
        at Sintatico.unario(Sintatico.java:1402)
        at Sintatico.incremento(Sintatico.java:1376)
        at Sintatico.termo(Sintatico.java:1330)
        at Sintatico.expressao(Sintatico.java:1298)
        at Sintatico.condicao(Sintatico.java:1264)
        at Sintatico.expressaoCondicional(Sintatico.java:1255)
        at Sintatico.condicaoUnaria(Sintatico.java:1239)
```

Figura 5.3 – Mensagem de erro no *console* da plataforma Eclipse

Após o término da *verredura* do código Clipper, pelo compilador, estando este código isento de imperfeições sintáticas, o compilador imprime em sua tela o respectivo código Java, conforme ilustra a figura 5.4.

[3] Console – saída de dados da plataforma

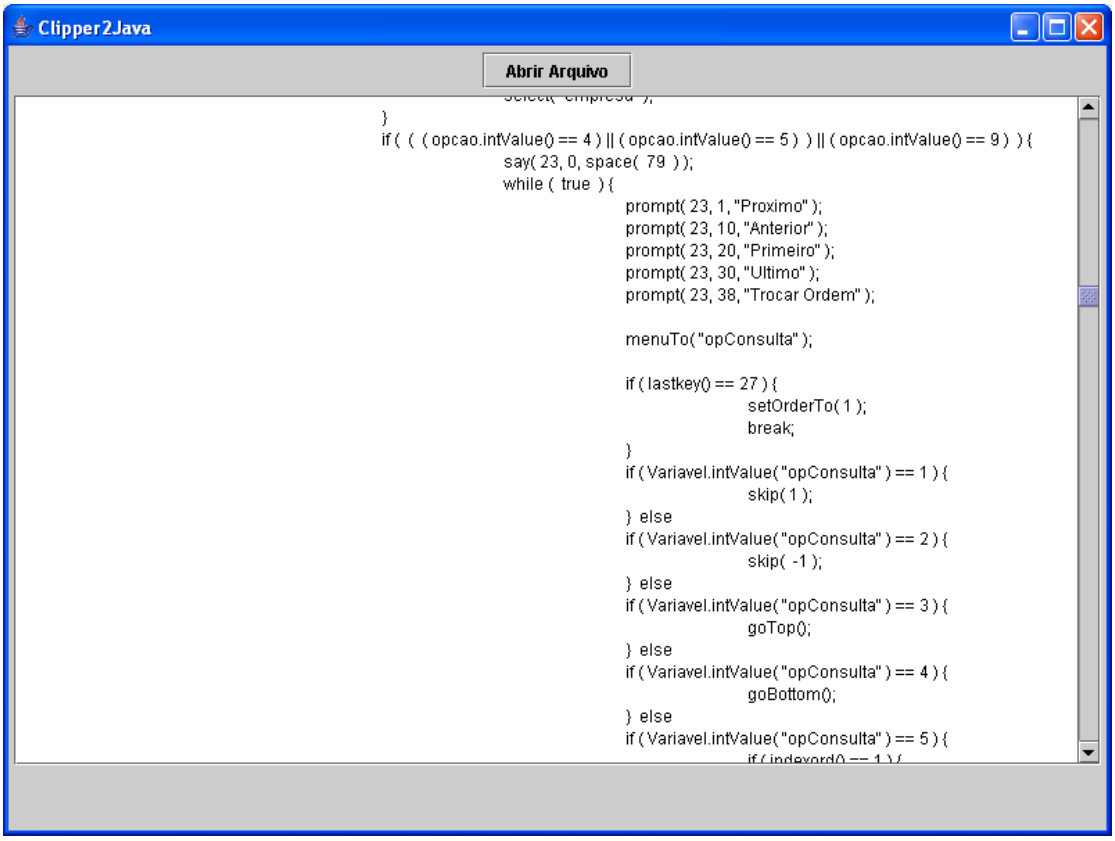

Figura 5.4 – impressão do código Java na tela do compilador

Por se tratar de uma tela com dimensões reduzidas, o código Java não se apresenta de forma legível. Entretanto, ao ser inserido na plataforma de desenvolvimento, passa a ter características e formatações adequadas à programação Java. Este código, após ser inserido em seu ambiente de programação, pode passar por algumas modificações até que esteja pronto para ser utilizado.

6 TESTES E RESULTADOS COM CÓDIGOS JAVA

Após a tradução do código, é preciso inserir tal código em seu ambiente de desenvolvimento, respeitando seus respectivos pacotes e classes. Os pacotes serão criados, dentro da ferramenta Eclipse, cumprindo-se a estrutura modularizada do Clipper.

Para obter o código projetado na tela do compilador copia-se o mesmo e, posteriormente, o insere em sua respectiva classe Java.

Erros no código Java são apontados pelo Eclipse. Estes erros, na maioria das vezes, ocorrem de forma inevitável, entretanto, são de fácil identificação pelo programador, e devem ser tratados individualmente, até que obtenha-se um código habilitado para fornecer uma aplicação Java.

O capítulo que segue apresenta resultados obtidos através das aplicações Java geradas a partir dos códigos convertidos.

6.1 Testes e Alterações em Códigos e Aplicações Java

Os códigos gerados apresentam-se bem estruturados, obedecendo uma formatação coerente com a linguagem Java, contendo os tratamentos de classes e métodos, importações de pacotes, entre outros. Por se tratar de um código com estrutura semelhante ao código original, a identificação e entendimento do mesmo torna-se fácil por parte dos programadores que necessitam utilizá-lo.

Entretanto, alguns erros podem ser reportados. Estes, na maioria das vezes, estão relacionados à classe ***Variavel***. Esta classe foi implementada a fim de tipificar os dados que, no código origem, não possuem declaração de tipos.

Ao compilador foi implementada uma tabela de símbolos que permite identificar o tipo de dado de determinadas variáveis, e projetar a correta estrutura desta classe. Entretanto, muitos dados não possuem prévia identificação, e necessitam alterações manuais pelo programador. Caso não haja correspondência de dados de determinada variável na tabela de símbolos, o tipo da variável é completado com "sublinhados", para que estes sejam substituídos manualmente com a sua tipificação correta. A figura 6.1 ilustra estes dois tipos de abordagens.

```
replace( "EMP_INSS", Variavel.as_________( "inss" ) );
replace( "EMP_EMIRPP", Variavel.asString( "emiRPP" ) );
replace( "EMP_MESREA", Variavel.asString( "mesRea" ) );
replace( "EMP_TABCAR", Variavel.asString( "tabCar" ) );
replace( "EMP_MODTAD", Variavel.asString( "modTad" ) );
replace( "EMP_RESMOV", Variavel.asString( "resMov" ) );
```

Figura 6.1 – Exemplo de utilização da classe ***Variavel***, com e sem correspondência de tipo de dados das variáveis na tabela de símbolos

Ainda neste contexto, outros erros de relevância mais baixa podem aparecer no código. Contudo, esta parte de código que não pôde ser traduzida por completo representa, em média, 20% da totalidade de códigos traduzidos.

Considera-se esta uma média satisfatória nos resultados, pois inicialmente projetava-se uma tradução de aproximadamente 60% do código, todavia com o aperfeiçoamento da ferramenta a cada importação realizada, e a praticidade que a mesma oferece em termos de sua utilização, conseguiu-se atingir um bom resultado no que tange a conversão de código.

Com o código fonte corrigido e revisado, inicia-se os testes com as aplicações destes códigos. A figura 6.1 mostra um código traduzido a partir do compilador Clipper2Java. Tal código é a conversão correspondente ao código Clipper apresentado na figura 5.1 do capítulo anterior.

```
package unimed.sigedu;
import jSun.lib.baseDados.BaseDeDadosException;

public class SigEdu extends ProgramaSun {

        public static void main( String [] args ) {
                new SigEdu();
        }

        public void executa() throws MenuException, TipoInvalidoException,
                                NomeInvalidoException, BaseDeDadosException,
                                ProcedureInvalidaException, FimProgramaException,
                                VariavelNaoEncontradaException, JSunException {

                telaPadrao( "Modulo : Comite Educativo" );
                box( 2, 1, 19, 36, BOX_SIMPLES );
                int[] janela = savescreen( 0, 0, 23, 79 );

                Variavel opcao = Variavel.atribui( "opcao", 1 );

                while (  true  ) {
                   if ( Variavel.asString( "numCli" ).equals( "055" ) ) {
                       prompt( 3, 2, "Gera Est. Com. Edu. - Dia 20 -   " );
                   } else {
                       prompt( 3, 2, "Gera Estatistica Com. Ed. do mes  " );
                   }
                       prompt( 4, 2, "Emite Estatistica Com. Ed. do mes" );

                   menuTo( "opcao" );

                   if ( opcao.intValue() == 0 ) {
                       break;
                   }  else
                   if ( opcao.intValue() == 1 ) {
                       if ( Seguranca.verifPerm(  "SIG", "A"  ) ) {
                               geraEstEdu();
                       } else {
                               continue;
                       }
                   }  else
                       if ( opcao.intValue() == 2 ) {
                               if ( Seguranca.verifPerm(  "SIG", "B"  ) ) {
                                       emiEstEdu();
                               } else {
                                       continue;
                               }
                       }
                       restscreen( 0, 0, 23, 79, janela );
                   }
                   BibliotecaP.p_TelaFim();
                   dbCloseAll();
                  return;
        }
}
```

Figura 6.2 – Exemplo de código traduzido com o compilador

6.1.1 Testes em Aplicações

Aplicações Java provenientes dos códigos traduzidos devem ser testadas com o objetivo de analisar sua correta atuação, bem como traçar um paralelo entre sua performance em ambiente DOS e a performance em ambiente Java.

Contudo, não se pôde obter um amplo resultado de testes quanto a performance de todos os módulos, pois a conversão até o momento da conclusão deste trabalho se deu em programas de manutenções, ou seja, onde existe o cadastro básico para determinadas tabelas. Nestes casos, o desempenho da aplicação mostrou-se satisfatório em termos das operações básicas: inclusão, alteração, exclusão e consulta.

Algumas quedas de performance, no entanto, são identificadas em aplicações que utilizam-se de processos mais complexos. Neste caso, deve-se avaliar e tratar de forma individual cada uma destas aplicações.

6.1.2 Modificações na Camada de Persistência

O acesso aos dados nos programas legados é feito de forma seqüencial, percorrendo registros de arquivos através de um índice. Esta forma não pôde ser convertida para *query* sql durante o processo de tradução.

Embora a máquina JClipper possua métodos de manipulação de dados através de *query* sql, tratando superficialmente estes casos, em algumas ocasiões observa-se uma queda muito significativa de performance.

Nestes casos, se faz necessário alterar a camada de persistência da aplicação, inserindo comandos sql no próprio código-fonte, substituindo a forma antiga de acesso aos dados. Estas alterações devem ser analisadas caso a caso, para que implemente-na da forma mais adequada à aplicação que se destina.

6.2 Resultados das Aplicações

Embora seja uma aplicação cuja aparência assemelha-se com a interface DOS, e aceita entrada de dados apenas através do teclado, sua atuação apresentou-se satisfatória, proporcionando melhor usabilidade em relação ao Clipper. A figura 6.3 mostra a interface de um programa da aplicação Java.

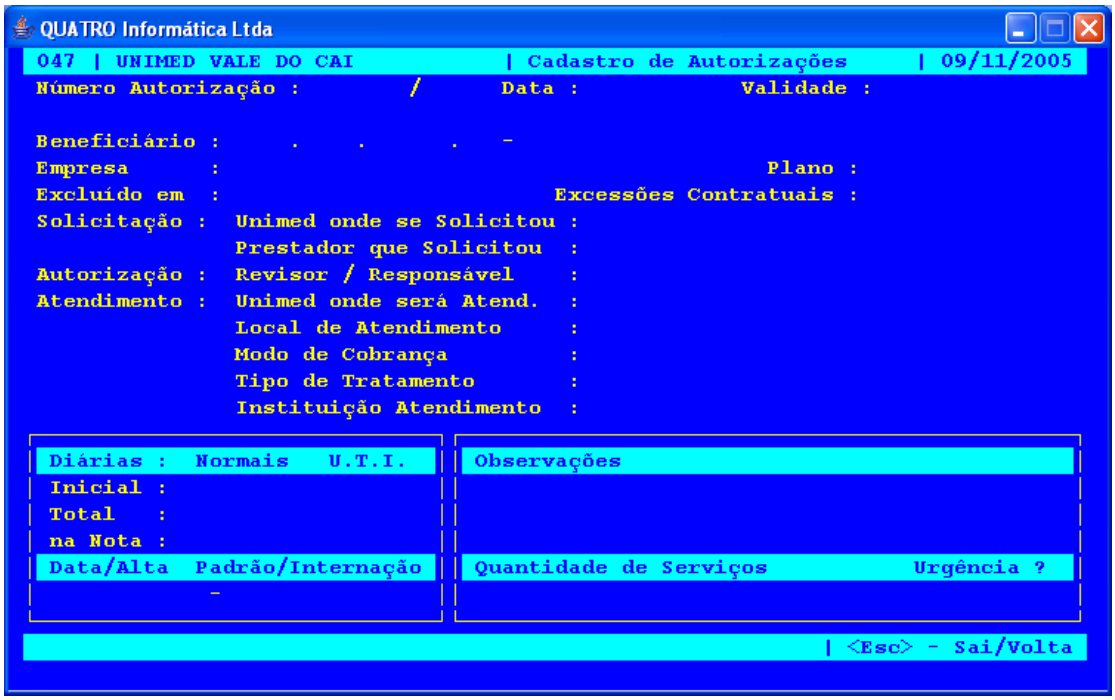

Figura 6.3 – Interface de um programa da aplicação Java

Ainda no âmbito de sua interface, esta torna-se amigável e familiar aos usuários do sistema, evitando-se custos e tempo de treinamento.

Por se tratar de uma aplicação Java, a utilização desta ferramenta pode-se dar em qualquer ambiente operacional. Esta característica propicia aos usuários, bem como à equipe de desenvolvimento, uma maior flexibilidade de escolha de configuração de máquinas, descartando-se a exigência de possuir configuração Windows com ambiente operacional DOS.

A possibilidade de execução em ambiente *web* proporciona facilidades no que tange a manutenção deste software, pois a possibilidade ter, à distância, a realidade operacional do cliente, torna-se ágil a identificação de suas necessidades, assim como o tempo de resposta destas manutenções.

As funcionalidades a que cada sistema se destina, apresentaram-se igualmente confiáveis às do sistema legado, com o diferencial de este novo sistema estar inserido em um recente contexto tecnológico.

Por fim, as características apresentadas quanto aos resultados da conversão de códigos através do Clipper2Java, refletem o sucesso do desenvolvimento desta ferramenta. O desenvolvimento, bem como a atuação desta, obedeceu aos critérios previstos no início de seu projeto. Foi possível alcançar significativa parcela no que diz respeito a número de linhas de código migrados, economizando-se tempo nas fases de tradução. O resultado da aplicação Java representou uma atualização tecnológica, preservando a estrutura do sistema, em termos de suas regras de negócio. Desta forma, atingiu-se os objetivos quanto ao desenvolvimento do compilador, e à migração de um sistema legado.

CONCLUSÕES

O objetivo deste trabalho concentrou-se em descrever a problemática de sistemas legados e apresentar uma proposta de modernização do mesmo para uma linguagem atual, através de um compilador que atua como tradutor deste código. O estudo de sistemas legados mostrou a necessidade de modernização destes sistemas e apontou a importância da utilização de uma ferramenta que automatizasse este processo.

O estudo de compiladores enfatiza que, embora o objetivo de um compilador seja assumir como entrada um código em linguagem de alto nível e produzir como saída um código em linguagem de máquina, observou-se que o mesmo pode atuar como tradutor de códigos, transformando determinado código escrito em uma linguagem de alto nível, para outra equivalente e também de alto nível.

A partir da necessidade de modernização de um software legado escrito em linguagem Clipper, projetou-se a construção de um compilador que traduz este código para Java. O processo da construção do compilador ocupou-se em projetar todas as etapas que compõe um compilador, elaborando-se, assim, uma gramática de atributos para o sistema desenvolvido em linguagem Clipper, construindo analisadores léxico, sintático e semântico, aliados à ferramenta JClipper, para identificar o código e projetá-lo em Java.

O processo de desenvolvimento do compilador foi acompanhado de testes com o mesmo proporcionando, assim, sua constante otimização. A proposta de economia em termos de tradução de código foi atingida com sucesso, diminuindo o tempo de tradução e verificação de regras de negócios durante as fases de migração de código.

Conseguiu-se alcançar o objetivo proposto, permitindo migrar para uma nova tecnologia os sistemas legados da Quatro Informática. No entanto, as regras desta ferramenta não se restringem às já desenvolvidas, podendo estas ser refinadas para melhor desempenho.

Em termos de trabalhos futuros, é possível agregar ao compilador um pré-processador que facilite as entradas dos identificadores no código Clipper, para a sua posterior importação, bem como otimização de regras que permitam maior eficiência e desempenho quanto à camada de persistência da aplicação gerada.

A partir deste trabalho, pôde-se alcançar propósitos que vão além de seu escopo. A elaboração de um artigo baseado em sua funcionalidade, publicado nos Anais do Primeiro Congresso Sul Catarinense de Computação destaca-se como reconhecimento deste trabalho. Além disto, a estrutura do código fonte do compilador serviu de base para a implementação do compilador *Verto* – um projeto de parceria com o Centro Universitário Feevale.

Em conclusão, os resultados atingidos a partir deste trabalho foram satisfatórios e recompensadores.

REFERÊNCIAS BIBLIOGRÁFICAS

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

APPLEBY, Doris.; VANDELOPPLE, Julius J. **Programming languages: paradigm and pratice**. McGrawHill: New York, 1997.

AMBLER, Scott W; JEWELL, Tyler. **Mastering enterprise JavaBeans.** Roman: Canada, 2002.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: guia do usuário.** Campus: Rio de Janeiro, 2000.

COMPUTER ASSOCIATES. **CA Clipper:** Quick Reference for DOS. Computer Associates: 1992.

CRESPO, Rui Gustavo. **Processadores de linguagem**: da concepção à implementação. Lisboa: IST Press, 1998.

DEITEL, Harvey M; DEITEL, Paul J. **Java** : como programar. Porto Alegre: Bookman, 2003.

ECLIPSE – **Eclipse.org**: 2005. Disponível em: <http://www.eclipse.org>. Acesso em <01 jun. 2005>.

FELICIANO NETO, Acácio; et al **Engenharia da informação**: metodologia, técnicas e ferramentas. Sao Paulo: McGraw-Hill, 1988.

GRUNE, Dick; BAL, Henri; JACOBS, Ceriel J. H.; LANGENDOEN, Koen G. **Projeto moderno de compiladores:** implementação e aplicações. Rio de Janeiro: Campus, 2001.

HUNTER, Robin. **Compiladores**: sua concepção em programaçãoem pascal. Lisboa: Presença, 1987.

JEPSON, Brian; PECKHAM, Joan; SADASIV, Ram. **Database application programing with Linux**. New York: Wiley, 2000.

JOSÉ NETO, João. **Introdução à compilação**. Rio de Janeiro: LTC, 1987.

KOWALTOWSKI, Tomasz. **Implementação de linguagens de programação**. Rio de Janeiro: Guanabara, 1983.

KRUEGER, Charles W. **Software reuse**. In: Acm Computer Surveys vol.4 No. 2. 1992.

LEVINE, John R.; MASON, Tony; BROWN, Doug. **Lex & Yacc**. Beijing: O'Reilly, 1990.

MARTIN, James. **Engenharia da informação**: introdução. Rio de Janeiro: Campus, 1991.

MAFFEO, Bruno. **Engenharia de software e especificação de sistemas**. Rio de Janeiro: Campus, 1992.

MOMJIAN, Bruce. **PostgreSQL:** introduction and concepts. Indianapolis: Addison-Wesley, 2001.

NIEDERAUER, Juliano. **PostgreSQL:** guia de consulta rápida. São Paulo: Novatec, 2001.

PAULA FILHO, Wilson de Pádua. **Engenharia de software**: fundamentos, métodos e padrões. Rio de Janeiro: LTC, 2003.

PETERS, James F.; PEDRYCZ, Witold. **Engenharia de software**: teoria e prática. Rio de Janeiro: Campus, 2001.

PFLEEGER, Shari Lawrence. **Engenharia de software**: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. **Implementação de linguagens de programação:** Compiladores. Porto Alegre: Sagra Luzato, 2001.

PYSTER, Arthur B. **Compiler design and construction:** tools and techniques with C and Pascal. New **York**: Van Nostrand Reinhold Company, 1988.

RUSSEL, Stuart; NORVIG, Peter. **Artificial inteligence:** a modern aproach. Prentice-Hall: New Jersey, 1995.

SEACORD, Robert C; PLAKOSH, Daniel; LEWIS, Grace A. **Modernizing legacy systems**: software technologies, engineering processes, and business practices. USA: Addison Wesley, 2003.

SOMMERVILLE, Ian. **Engenharia de software**. 6. ed. São Paulo: Addison-Wesley, 2001.

TURBAN, Efraim; MCLEAN, Ephraim; WETHERBE, James C. **Tecnologia da informação para gestão:** transformando os negócios na economia digital. Porto Alegre: Bookman, 2004.

WATT, David A.; BROWN, Deryck F. **Programming language processors in Java:** compilers and interpreters. Harlow: Prentice-Hall, 2000.

WINCK, Ana T; SCHNEIDER, Carlos S. OLIVEIRA, Ricadro F. **O auxilio de um compilador na tradução de um sistema legado.** In: Anais do I Congresso Sul Catarinense de Computação, 2005.

ANEXO 1 – GRAMÁTICA