UNIVERSIDADE DO OESTE PAULISTA
UNOESTE
PRESIDENTE PRUDENTE - SP

FACULDADE DE INFORMÁTICA DE PRESIDENTE PRUDENTE

CORAL: PROTÓTIPO DE UMA LINGUAGEM ORIENTADA A OBJETOS USANDO GERADORES DE COMPILADORES JAVA

DIOGO BRANQUINHO RAMOS

Presidente Prudente – SP
2005

FACULDADE DE INFORMÁTICA DE PRESIDENTE PRUDENTE

CORAL: PROTÓTIPO DE UMA LINGUAGEM ORIENTADA A OBJETOS USANDO GERADORES DE COMPILADORES JAVA

DIOGO BRANQUINHO RAMOS

Trabalho monográfico apresentado no curso de graduação, Bacharelado em Ciência da Computação, como requisito parcial para sua conclusão.

Área de concentração: Linguagens Formais e Autômatos.

Orientadora: Profª. MSc. Ana Paula Domeneghetti Parizoto Fabrin
Co-orientador: Profº MSc. Francisco Assis da Silva

Presidente Prudente – SP
2005

004 RAMOS, Diogo Branquinho.
Coral: Protótipo de uma Linguagem Orientada a Objetos usando Geradores de Compiladores Java / Diogo Branquinho Ramos. – Presidente Prudente : UNOESTE – Universidade do Oeste Paulista, 2005.
68p. : il

Monografia (Graduação) – Universidade do Oeste Paulista – UNOESTE: Presidente Prudente – SP, 2005.
Orientador: MSc. Ana Paula Domeneghetti Parizoto Fabrin.
Co-orientador: MSc. Francisco Assis da Silva

1. Compiladores, 2. Orientação a Objetos, 3. Java. I. Autor. II. Título.

DEDICATÓRIA

Dedico este trabalho aos meus pais, Wilson e Mara, que nunca mediram esforços para que os meus objetivos fossem conquistados. A minha irmã Kadija, pelo carinho e afeto, e a minha namorada Carolina que sempre me apoio e entedeu as minhas ausências.

AGRADECIMENTOS

A minha família que, em todos os momentos de realização desta pesquisa, esteve presente.

A minha orientadora, MSc. Ana Paula Domeneghetti Parizoto Fabrin que, na rigidez de seus ensinamentos, fez aprimorar meus conhecimentos.

Aos amigos, Amandia, Eli, Julierme, Profº Leandro e Profº Chico pelo companheirismo e os muitos momentos de alegria compartilhados.

Agradecimentos também aos amigos da república, Carlos, Guilherme, Rodolfo e Rafael, que compartilharam todos os momentos deste trabalho.

EPÍGRAFE

“A decolagem é opcional, mas o pouso é obrigatório...”

Autor Desconhecido

RAMOS, Diogo Branquinho. **Coral: Protótipo de uma Linguagem Orientada a Objetos Usando Geradores de Compiladores Java**. Presidente Prudente: UNOESTE, 2005. Monografia de Graduação.

Orientadora: MSc. Ana Paula Domeneghetti Parizoto Fabrin
Co-Orientador: MSc. Francisco Assis da Silva

RESUMO

Este trabalho refere-se ao desenvolvimento de um Compilador para uma Linguagem do Paradigma Orientado a Objetos com o auxílio de Geradores de Compiladores Java. A função do compilador é de analisar o código em alto nível, escrito na linguagem Coral – linguagem desenvolvida para este trabalho – e transformá-lo em linguagem de montagem, para a Máquina Virtual Java. Para obter o código em linguagem de montagem, o compilador divide-se em duas fases: a fase de análise e a fase de síntese, implementadas pelos Geradores de Compiladores. Na fase de análise são identificados os *tokens* para a construção da tabela de símbolos, a manipulação de erros, a sintaxe e a semântica de cada palavra contida na linguagem. Após a conclusão com sucesso da fase de análise, o compilador realiza a fase de síntese. Nesta fase obtém a geração do código objeto, ou seja, código em linguagem de montagem (J*ava Assembly*). Com a elaboração e projeto de um compilador, desde a análise do código-fonte até a geração do código-objeto, é possível verificar o grau de complexidade que envolve um projeto de um compilador e aumentar os conhecimentos na área de linguagens formais e autômatos, principalmente quando se trata de uma linguagem do paradigma Orientado a Objetos.

RAMOS, Diogo Branquinho. **Coral: Prototype of Language Objects Oriented Paradigm using Java Compilers Generators**. Presidente Prudente: UNOESTE, 2005. Graduation Monograph.

Adviser: MSc. Ana Paula Domeneghetti Parizoto Fabrin
Co-Adviser: MSc. Francisco Assis da Silva

ABSTRACT

This work refers the Compiler development for a Language of Objects Oriented Paradigm with assists of the Java Compilers Generators. The compiler function is to analyze the code in high level written in the Coral language – language developed for this work – and to transform it into assembler language, for the Java Virtual Machine. To get the assembler language code the compiler divides itself in two phases: the phase of analysis and the phase of synthesis implemented for the Compilers Generators. The tokens are identified in the analysis phase to the construction of the symbols table, the errors manipulation, the syntax and the semantics of each word contained in the language. After the conclusion successfully of the analysis phase the compiler to make the synthesis phase. In this phase is making the object code generation that is code in assembler language (Java Assembly). With the elaboration and project of a compiler since the analysis of the source code until the object code generation it is possible to verify the complexity degree that involves a compiler project and to increase the knowledge in the area of formal languages and automatons, mainly when to refer a language of Objects Oriented Paradigm.

LISTA DE FIGURAS

Figura 1 - Execução de um programa fonte 15
Figura 2 - Estrutura de um Compilador 16
Figura 3 - Analisador Léxico 17
Figura 4 - Classe em Coral 38
Figura 5 - Declaração de Atributos 39
Figura 6 - Criação de Métodos 39
Figura 7 - Declaração de Variáveis 39
Figura 8 - Entrada de Dados 40
Figura 9 - Saída de Dados 40
Figura 10 - Atribuição de Dados 40
Figura 11 - Seleção 41
Figura 12 - Repetição 41
Figura 13 - Parâmetros 42
Figura 14 - Retorno 42
Figura 15 – Comentários 43
Figura 16 - Construtor de classe 43
Figura 17 - Início do Arquivo de Configuração do JavaCC 44
Figura 18 - Definindo os caracteres que serão ignorados pelo Compilador 45
Figura 19 - *Tokens* da linguagem 46
Figura 20 - Identificadores da linguagem 46
Figura 21 - Símbolos Especiais e Operadores 47
Figura 22 - Construção do não-terminal **`programa`** 48
Figura 23 - Construção do não-terminal **`declclasses`** 48
Figura 24 - Construção do não-terminal **`corpoclasse`** 48
Figura 25 - Construção do não-terminal **`corpoclasse`** com **`LOOKAHEAD`** 49

Figura 26 - Não-terminal **`programa`** com tratamento de erros 51

Figura 27 - Diagrama de classes do pacote **`recuperacao`** 52

Figura 28 - Não-terminal **`listaclasses`** com recuperação de erros 53

Figura 29 - Exemplo de árvore sintática.....54

Figura 30 - Classe **`GeralNo`**.....55

Figura 31 - Construção do não-terminal **`declclasses`** para árvore sintática.....55

Figura 32 - Classe **`DeclClasseNo`** 56

Figura 33 - Diagrama de classes do pacote **`tabelaSimbolos`**.....58

Figura 34 - Diagrama de classes do pacote **`analiseSemantica`**.....61

Figura 35 - Esquema do Compilador para a linguagem Coral 62

Figura 36 - IDE Coral 65

LISTA DE TABELAS

Tabela 1 - Tabela de *Tokens*, Padrões e Lexemas 17

Tabela 2 - A Evolução das Linguagens de Programação.....24

Tabela 3 - Exemplo de programa interativo em Pseudolinguagem.....25

Tabela 4 - Código-fonte do Lex.....34

Tabela 5 - Código-fonte do Lex em conjunto com o Yacc 34

Tabela 6 - Código-fonte do JavaCC para análise léxica.....35

Tabela 7 - Análise Sintática para uma calculador de expressões Matemáticas 36

Tabela 8 - Representação dos tipos em Coral para a JVM.....62

SUMÁRIO

1 INTRODUÇÃO 12

2 COMPILADORES 14

2.1 Implementação de um Compilador e suas Fases 15

2.2 Análise Léxica 16

2.2.1 Tokens, Padrões e Lexemas 17

2.2.2 Erros Léxicos 18

2.3 Analise Sintática 18

2.3.1 Top-Down (Análise Descendente) 19

2.3.1.1 Análise Recursiva com Retrocesso 20

2.3.1.2 Análise Recursiva Preditiva 20

2.3.1.3 Análise Preditiva Tabular 20

2.3.2 Botton-Up (Análise Redutiva) 22

2.4 Analise Semântica 22

3 PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO 24

3.1 Linguagens Imperativas 24

3.2 Linguagens Orientada a Objetos 25

3.2.1 Definições 26

3.2.2 Conceitos Básicos 28

3.2.2.1 Abstração 28

3.2.2.2 Encapsulamento 29

3.2.2.3 Compartilhamento 29

3.2.3 Objetos e Classes 30

3.2.3.1 Atributos 30

3.2.3.2 Operações e Métodos 31

3.2.3.3 Generalização e Herança 32

4 GERADORES DE COMPILADORES 33
4.1 Lex e Yacc 33
4.2 Flex e Bison 35
4.3 JavaCC e Jasmin 35
5 CORAL 38
5.1 Especificação da Linguagem 38
5.2 Construção Léxica 43
5.3 Construção Sintática 47
5.3.1 Erros Sintáticos 49
5.3.2 Árvore Sintática 53
5.3.3 Tabela de Símbolos 56
5.4 Construção Semântica 58
5.5 Geração de Código 61
5.6 IDE 64
6 CONCLUSÃO 66
6.1 Trabalhos Futuros 67
REFERÊNCIAS BIBLIOGRÁFICAS 68

1 INTRODUÇÃO

Na programação dos computadores sempre foi necessária a existência de um meio de comunicação entre máquina e homem para que através dele fosse possível se fazer à ligação do pensamento humano e a precisão requerida para o processamento pela máquina.

Segundo Price (2000), uma linguagem de programação serve como meio de comunicação entre o indivíduo que deseja resolver um determinado problema e o computador escolhido para ajudá-lo na solução. Esse elo de ligação é consolidado pelo compilador que tem por função transformar a linguagem fonte em uma linguagem interpretável pela máquina. Para que a tradução se faça de forma completa é necessário reconhecer as unidades lexicamente significativas do código gerado pelo individuo programador, além disso, o código gerado deve respeitar uma adequação da estrutura sintática das unidades léxicas com as regras de formação da linguagem e por fim garantir que a semântica do código-fonte seja preservada no código destino.

O desenvolvimento de problemas por meio das linguagens de programação se torna fácil na medida em que a linguagem escolhida se torna cada vez mais natural à linguagem humana, em contrapartida isso se torna um problema para o compilador que deve traduzir essa linguagem com alto nível de abstração. No decorrer dos anos as linguagens de programação foram evoluindo, principalmente com o paradigma da Orientação a Objetos que visa tratar a natureza como se fossem conjuntos. Para exemplificar, Platão (427-347 a.C) afirma que, ao considerar um conjunto de cavalos, apesar de eles não serem exatamente iguais, existe algo que é comum a todos os cavalos, algo que garante que jamais existirá problema para reconhecer um cavalo. Paralelo a evolução das linguagens, ferramentas de auxílio ao desenvolvimento de compiladores foram construídas, chamadas de geradores de compiladores, automatizando fases complexas do mesmo e maximizando a capacidade de se desenvolver novos conceitos para aproximar cada vez mais a linguagem de programação à linguagem natural, humana.

O objetivo deste trabalho é desenvolver um Compilador para uma Linguagem Orientada a Objetos, que possa ser executa em uma máquina virtual, para isso será definida uma linguagem chamada Coral – é uma linguagem orientada

a objetos semelhante a Java, criado por Sun Microsystems, porém mais simples, utilizando as ferramentas de auxílio ao desenvolvimento, os geradores de compiladores JavaCC (Java Compiler Compiler), criado por Sun Microsystems, que trata das fases de analise e o Jasmin (Java Assembler Interface), criado por Jonathan Meyer, que trata da fase de síntese de um compilador.

2 COMPILADORES

Neste capítulo será tratado a funcionalidade de um compilador no ambiente computacional, definir as suas principais etapas e como elas trabalham entre si, além de fazer um paralelo entre a evolução das linguagens de programação e dos computadores.

As linguagens de programação evoluíram para tornar a programação de computadores muito mais fácil para o ser humano, na hora de resolver um problema computacional, mudaram radicalmente e de forma bastante eficiente, já os computadores continuam inalterados e só conseguem entender instruções simples que estejam previamente descritas em suas coleções de instruções. Dessa forma programas escritos em linguagens de alto nível necessitam ser convertidos para as instruções de máquinas para serem executados.

Todo programa que converta ou transforme uma linguagem de programação específica a partir da linguagem fonte, em uma linguagem interpretável pela máquina é chamada de tradutor. Segundo Prince (2000), os tradutores são classificados em:

- Montadores assemblers – mapeiam uma instrução de linguagem simbólica (*assembly*) para uma instrução de linguagem de máquina;
- Macro-assemblers – mapeiam uma instrução macro de linguagem simbólica para uma seqüência de comandos simbólicos antes de tradução para a linguagem de máquina;
- Pré-processadores ou Filtros – são tradutores que efetuam conversões entre duas linguagens de alto nível;
- Interpretadores – aceitam um código intermediário de um programa anteriormente traduzido e produzem a execução do algoritmo sem mapeá-lo para linguagem de máquina;
- Compiladores – mapeiam programas escritos em linguagens de alto nível para programas equivalentes em linguagem de máquina ou simbólica.

Dessa forma um compilador é simplesmente um tradutor que recebe um programa fonte, descrito conforme a linguagem de programação, faz a tradução do mesmo e produz uma outra linguagem, a linguagem objeto, que será interpretada pela máquina. Os dois processos acontecem em momentos distintos, em um primeiro tempo de compilação e em um segundo tempo de execução, o processo é ilustrado na Figura 1.

Figura 1 - Execução de um programa fonte

Fonte: Implementação de Linguagens de Programação: Compiladores (Price, 2000).

2.1 Implementação de um Compilador e suas Fases

Na maioria das vezes os tradutores de linguagem de programação são programas bastante complexos. Com o passar dos anos surgiu a necessidade de trabalhar melhor a questão e muitos estudos foram desenvolvidos, criando assim diversas teorias com intuito de estruturar um tradutor em fases. Nessa estrutura estão todas as funções básicas do tradutor, divídidas na fase de análise e outra de síntese, conforme ilustrado na Figura 2.

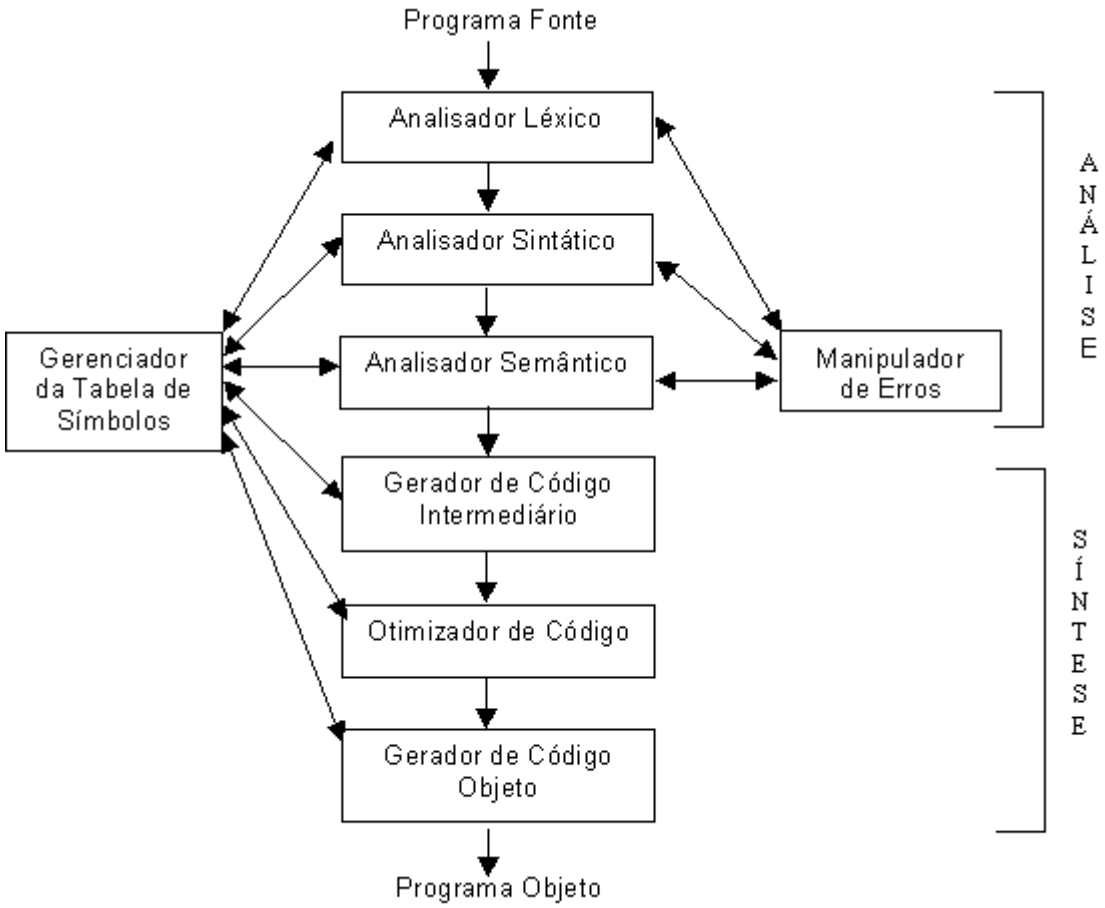

Figura 2 - Estrutura de um Compilador

Fonte: Compiladores: princípios, técnicas e ferramentas (Aho, 1986).

2.2 Análise Léxica

O analisador léxico é a primeira fase de um tradutor, no nosso caso o compilador, segundo Delamaro (2004), o analisador léxico encarrega-se de separar no programa fonte cada símbolo que tenha algum significado para a linguagem ou de avisar quando um símbolo que não faz parte da linguagem é encontrado, os símbolos são chamados de *tokens*, os *tokens* constituem classes de símbolos tais como palavras reservadas, delimitadores, identificadores entre outros, já outras classes de símbolos são descartadas durante o processo léxico, como espaços em brancos, tabulações, caracteres de avanço de linha e comentários, todos os *tokens* encontrados são inseridos em uma tabela interna, tanto para indicar erros léxicos como para servir de entrada para o analisador sintático. Observe na Figura 3 o funcionamento do analisador léxico.

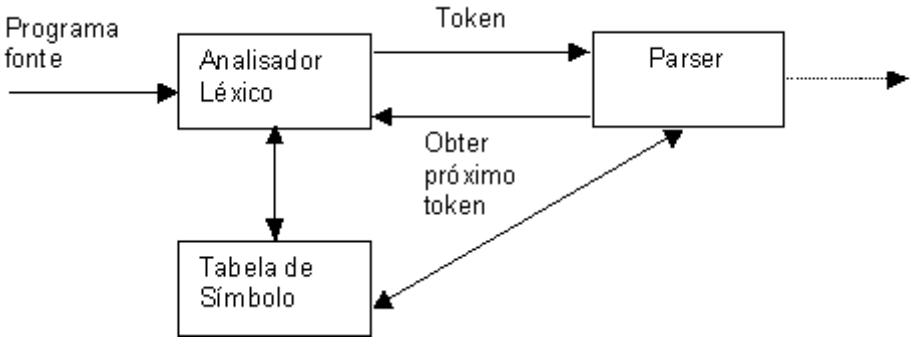

Figura 3 - Analisador Léxico

Fonte: Compiladores: princípios, técnicas e ferramentas (Aho, 1986).

As linhas do programa fonte devem ser controladas e armazenadas durante a varredura da análise léxica, pois os erros de análise sintática e/ou análise léxica podem estar associados ao programa fonte. O desempenho do compilador pode ser melhorado por meio de técnicas de buferização especializadas para a leitura de caracteres e processamento de *tokens*.

2.2.1 *Tokens*, Padrões e Lexemas

Na analise léxica são utilizados os termos *tokens*, padrões e lexemas, onde cada termo possui um significado específico, de forma que, os *tokens* são símbolos terminais na gramática para a linguagem fonte, como, palavras-chave, operadores, identificadores, constantes, literais, cadeias e símbolos de pontuação, como parênteses, vírgulas e ponto e vírgulas. Padrão é uma regra que descreve o conjunto de lexemas que podem representar um *token* particular no programa fonte, como na linguagem Coral, o padrão para o *token se* é simplesmente *se*, pois é uma palavra-chave, para o *token* relação o padrão são todos os operadores relacionais. Lexema é o conjunto de caracteres no programa fonte que é reconhecido pelo padrão de algum *token*, por exemplo, pi =3.1415 a subcadeia pi é um lexema para o *token* identificador. Vejamos alguns exemplos na Tabela 1.

Tabela 1 - Tabela de *Tokens*, Padrões e Lexemas

Token	Lexemas	Descrição informal do Padrão
Se	Se	Se
Classe	Classe	Classe
Id	pi, contador, i, x	Letra seguida por letras e/ou dígitos.

Num	3.1324, 0, 6.0432E2	Qualquer constante numérica.
Literal	“mensagem”	Quaisquer caracteres entre aspas duplas, exceto as aspas.

A formalização de um padrão de um determinado *token* é descrita por meio de expressões regulares. O reconhecimento de *tokens* é realizado a partir de regras de produção do tipo BNF (Backus Naur Form) ou até mesmo por meio de um autômato finito determinístico (AFD).

2.2.2 Erros Léxicos

Poucos erros são distingüíveis somente no nível léxico, considerando que o analisador léxico possui uma visão muito local do programa fonte, um erro comum é alguma cadeia que aparece na entrada não pode ser reconhecida como um *token* válido. Por exemplo a cadeia: @#@$@#@#@#@.

2.3 Analise Sintática

A segunda fase da estrutura básica de tradutor é a sintática, muitos autores a chamam de “coração” de um compilador, por meio da tabela de símbolos e/ou árvore sintática, ele vai analisar o programa fonte e verificar se este pertence ou não a linguagem desejada, se a cadeia de *tokens* pode ser gerada pela gramática da linguagem fonte e relatar quaisquer erros de sintaxe de forma inteligível. As vantagens da utilização das gramáticas na análise sintática é que ela oferece uma especificação sintática precisa e fácil de entender, o processo de construção do analisador pode revelar ambigüidades sintáticas, bem como outras dificuldades difíceis de serem detectadas que auxilia na detecção de erros.

Segundo Price (2000), o analisador sintático identifica seqüências de símbolos que constituem estruturas sintáticas (por exemplo, expressões, comandos), por meio de uma varredura ou *parsing* da representação interna (cadeia de *tokens*) do programa fonte. O analisador sintático produz uma estrutura de árvore, chamado árvore de derivação, que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem. A árvore de sintaxe visa eliminar redundâncias e elementos supérfluos. Esta estrutura objetiva facilitar a geração do código que é a fase seguinte.

Os analisadores sintáticos devem ser projetados de forma que os mesmos possam trabalhar de forma robusta, que consigam prosseguir na analise, até o fim do programa, mesmo que encontre erros sintáticos no código-fonte. Existem para isso duas estratégias bastante utilizadas para a analise sintática:

- *Top-Down* ou descendente: analisa a gramática desde seu início até o seu fim, construindo sua árvore de derivação a partir do símbolo inicial da gramática, fazendo com que ela cresça ate atingir suas folhas;
- *Bottom-Up* ou redutiva: analisa a gramática do fim para o seu começo, a partir dos *tokens* de derivação do código-fonte, constrói a árvore até o símbolo inicial da gramática.

2.3.1 Top-Down (Análise Descendente)

A análise *Top-Down* pode ser vista como uma tentativa de se encontrar uma derivação mais a esquerda para uma cadeia de entradas, segundo AHO (1986), a análise *Top-Down* pode ainda ser vista numa tentativa de se construir uma árvore gramatical, para a cadeia de entrada, a partir da raiz, criando nós da árvore gramatical em pré-ordem. Existem três tipos de analisadores sintáticos descendentes:

- recursivo com retrocesso *(backtracking);*
- recursivo preditivo;
- tabular preditivo.

Os dois primeiros tipos, trabalham de forma onde cada símbolo não-terminal é implementado por um procedimento que realiza o reconhecimento do(s) lado(s) direito(s) das produções que definem o símbolo em questão. Já o terceiro tipo, é implementado utilizando se um autômato de pilha controlado por uma tabela de análise, indicando a regra de produção a ser aplicada relativa ao símbolo não-terminal que se encontra no topo da pilha.

2.3.1.1 Análise Recursiva com Retrocesso

A análise recursiva com retrocesso faz a expansão da árvore de derivação a partir da raiz, crescendo sempre pelo não-terminal mais à esquerda, Prince (2000) diz mais, que quando existe uma regra de produção para o não-terminal a ser expandido, a opção escolhida é função do símbolo corrente na fita de entrada (*token* sob o cabeçote de leitura). Se o *token* de entrada não define univocamente a produção a ser usada, então todas as alternativas vão ser tentadas até que se obtenha sucesso (ou até que se falhe irremediavelmente).

Segundo Takehana (2004), em muitos casos, escrevendo-se cuidadosamente uma gramática eliminando-se a recursão à esquerda e fatorando-se à esquerda a gramática resultante, pode-se obter uma nova gramática processável por um analisador sintático de descendência recursiva que não necessite de retrocesso, ou seja, um analisador sintático preditivo. Assim, a alternativa adequada precisa ser detectável examinando apenas para o primeiro símbolo da cadeia que a mesma deriva. Por exemplo: **`cmd → if...| while…| begin…end|…`** As palavras chaves **`if, while e begin`** informam qual é a única alternativa que possivelmente teria sucesso no momento de encontrar um comando.

2.3.1.2 Análise Recursiva Preditiva

Os analisadores Recursivos Preditivos não possuem retrocessos, são recursivos sem retrocesso, para eles o símbolo sobre o cabeçote de leitura determina exatamente qual produção dever ser aplicada na expansão de cada não-terminal, para que esse eles possam trabalhar dessa forma a gramática a ser analisada não possua recursividade à esquerda, a gramática esteja fatorada à esquerda e que não exista para os não-terminais mais de uma regra de produção.

2.3.1.3 Análise Preditiva Tabular

Na análise Preditiva Tabular é possível construir analisadores preditivos não recursivos que utilizam uma pilha explícita ao invés de chamadas recursivas (pilha implícita). Esse tipo de analisador implementa um autômato de pilha controlado por uma tabela de análise. O princípio do reconhecimento preditivo é a determinação da produção a ser aplicada, cujo lado direito irá substituir o símbolo não-terminal que se encontra no topo da pilha. O analisador busca a

produção a ser aplicada na tabela na tabela de análise, levando em conta o não-terminal no topo da pilha e o *token* sob o cabeçote de leitura. O analisador preditivo compreende uma fita de entrada, uma pilha e uma tabela de análise. A fita de entrada contém a sentença a ser analisada seguida de $, símbolo que marca o fim da sentença. A tabela de análise é uma matriz M com n linhas e t+1 colunas, onde n é o número de símbolos não-terminais e t, o número de terminais – a coluna extra corresponde ao $.

Segundo Price (2000), a implementação de um analisador preditivo tabular tem como maior dificuldade a construção da tabela de análise. Para construir essa tabela, é necessário computar duas funções associadas à gramática: as funções *FIRST* e *FOLLOW*. Para computar *FIRST(x)* para um símbolo x da gramática, aplicam-se regras até que não se possa adicionar mais terminais ou ε ao conjunto em questão. As regras são:

1. Se a é terminal, então *FIRST(a)* = {a};
2. Se $x \rightarrow \varepsilon$ é uma produção, então adicione ε a *FIRST(x)*;
3. Se $x \rightarrow y1y2...yk$ e, para algum i, todos y1y2...yi-1 derivam ε, então *FIRST(yi)* está em *FIRST(x)*. Se todo yj {j=1,2,...,k} deriva ε, então ε está em *FIRST(x)*.

Para computar *FOLLOW(x)*, aplicam-se regras até que não se possa adicionar mais símbolos ao conjunto em questão. As regras são:

1. Se S é símbolo inicial da gramática e $ é o marcador de fim da sentença, então $ está em *FOLLOW(S)*;
2. Se existe uma produção do tipo $A \rightarrow \alpha X \beta$, então todos os terminais de *FIRST(β)* faz parte de *FOLLOW(X)*;
3. Se existe produção do tipo $A \rightarrow \alpha X$ ou $A \rightarrow \alpha X \beta$ sendo que $\beta \rightarrow^* \varepsilon$, então todos os terminais que estiverem em *FOLLOW(A)* fazem parte de *FOLLOW(x)*.

2.3.2 Botton-Up (Análise Redutiva)

Para Price (2000), a análise redutiva de uma sentença (ou programa) pode ser vista como a tentativa de construir uma árvore de derivação a partir das folhas, produzindo uma derivação mais à direita ao reverso. A denominação redutiva refere-se ao processo que sofre a sentença de entrada, a qual é reduzida até ser atingido o símbolo inicial da gramática (raiz da árvore de derivação). Dá-se o nome de redução à operação de substituição do lado direito de uma produção pelo não-terminal correspondente (lado esquerdo).

Price (2000), afirma ainda que os analisadores redutivos, também chamados empilha-reduz, são normalmente implementados por autômatos de pilha, com controle dirigido por uma tabela e análise. Na configuração inicial do analisador, a fita de entrada contém sentença de entrada seguida de um $ (marcador de fim), e a pilha contém apenas o marcador de base $. O processo de reconhecimento consiste em transferir símbolos da fita de entrada para a pilha até que se tenha na pilha um lado direito de produção. Quando isso ocorre, esse lado direito é substituído (reduzido) pelo símbolo do lado esquerdo da produção. O processo segue adiante com esses movimentos (empilhamento e redução) até que a sentença de entrada seja completamente lida, e a pilha fique reduzida ao símbolo inicial da gramática.

2.4 Analise Semântica

A análise semântica é utilizada para verificar aspectos do programa relacionados ao significado de cada comando, os erros semânticos no programa fonte e capturar as informações de tipo para a fase subseqüente de geração de código. Um programa pode estar escrito de forma correta de acordo com as regras de sintaxe definidas pela gramática, mais apresentar alguns problemas com a sua semântica.

Delamaro (2004), afirma que a análise semântica é realizada em diversas fases, cada uma delas objetivando a análise de aspectos específicos da semântica da linguagem. Um ponto a ser destacado é que cada uma dessas fases é feita por meio de um passeio pela árvore sintática, dessa forma a casa fase a árvore

sintática é percorrida e dados são levantados para serem utilizados na fase seguinte.

Outra função do analisador semântico é a verificação de tipos e a coerção de operandos, por exemplo, quando um operando aritmético binário é aplicado a um inteiro e um real, o compilador pode converter o inteiro para real. Isto porque dentro da máquina, um padrão de bits representando um inteiro é geralmente diferente do padrão de bits para um real, mesmo com um número de mesmo valor.

A conversão, também chamada de coerção, de um tipo para outro é dita implícita se for realizada automaticamente pelo compilador. Estão limitadas em muitas linguagens, de forma que nenhuma informação seja perdida, em princípio, por exemplo, de inteiro para real, mas não de real para inteiro, o que acarretaria perca de algumas casas decimais do número real. A conversão é dita explícita se o programador precisa escrever algum comando para provocar a conversão, como em Pascal **chr** (inteiro para caractere).

3 PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

As linguagens de programação percorreram um longo caminho desde a introdução do computador. Conforme mostra a Tabela 2, primeiramente a programação passou pela Era do Caos, quando os programas eram construídos usando-se instruções de desvio como "*ADD AX,5*" e "*JMP Error*". Desnecessário dizer que a Era do Caos não durou muito e a maioria dos programadores recebeu de braços abertos à evolução que começou nos anos 60 - a Era da Estruturação. Essa nova era introduziu linguagens de programação, como o Pascal, C e Ada e outras ferramentas que ajudaram a colocar ordem na caótica atividade de programação. E por fim a era dos Objetos, onde surgiram as primeiras linguagens Orientadas a Objetos, como Smalltalk, C++ entre outras (Sebesta,1999).

Tabela 2 - A Evolução das Linguagens de Programação

1950-1960 **Era do Caos**	**1970-1980** **Era da Estruturação**	**1990** **Era dos Objetos**
Saltos, goto's, "variáveis não estruturadas ao longo dos programas".	If-then-else blocos registros laços while.	Objetos, mensagens, métodos, herança.

3.1 Linguagens Imperativas

As linguagens Imperativas possuem como principal característica a execução linear de expressões, atribuições de valores, acesso variáveis, desvios condicionais, entre outros comandos, ou seja, um programa imperativo possue início e fim sendo que neste intervalo acontecem a execução dos comandos já citados. Muitos autores comparam os programas escritos em linguagens imperativas, como receitas culinárias, onde o autor da receita, indica qual o próximo passo a ser executado, incluindo ainda, condições, desvios entre outros.

No auxílio ao desenvolvimento imperativo, surgiram os procedimentos e funções, que executam desvios lógicos nos programas, com o objetivo de executarem uma rotina específica e logo após ao seu término, retornarem ao ponto em que foram chamadas. Veja na Tabela 3.

Tabela 3 - Exemplo de programa interativo em Pseudolinguagem

Pascal	Detalhes
Programa Teste;	
Variaveis a,b,c : inteiro;	
Inicio	Inicio do programa
Exibir(“Teste de um somador”);	
Exibir(“Valor 1:”);	
Readln(a);	Linearidade de comandos
Exibir(“Valor 2:”);	
Readln(b);	
c:=a+b	
Exibir(“Resultado da soma”,c);	
Fim.	Fim do programa

3.2 Linguagens Orientada a Objetos

> [...] Antes de Platão (427-347 a.C.), Empédocles (494-434 a.C.) e Demócrito (460-370 a.C.) haviam observado que apesar de os fenômenos da natureza "fluírem", havia "algo" que nunca se modificava (as *quatro raízes* ou os *átomos*). Para Platão tudo o que podemos tocar e sentir na natureza "flui". Não existe, portanto, um elemento básico que não se desintegre. Absolutamente tudo o que pertence ao *mundo dos sentidos* é feito de um material sujeito à corrosão do tempo. Ao mesmo tempo, tudo é formado a partir de uma *forma* eterna e imutável.
>
> Para exemplificar a visão de Platão, considere um conjunto de cavalos. Apesar de eles não serem exatamente iguais, existe algo que é comum a todos os cavalos; algo que garante que nós jamais teremos problemas para reconhecer um cavalo. Naturalmente, um *exemplar* isolado do cavalo, este sim "flui", "passa". Ele envelhece e fica manco, depois adoece e morre. Mas a verdadeira *forma* do cavalo é eterna e imutável. Numa outra situação, considere que você passe em frente a uma vitrine de uma padaria (sua primeira padaria) e vê sobre um tabuleiro cinqüenta broas exatamente iguais, todas em forma de anõezinhos. Apesar de você perceber que um anãozinho está sem o braço, o outro perdeu a cabeça e um terceiro tem uma barriga maior que a dos outros, você chega à conclusão que todas as broas têm um denominador comum. Embora nenhum dos anõezinhos seja absolutamente perfeito, você suspeita que eles devem ter uma origem comum. E chega à conclusão de que todos foram assados na mesma fôrma.
>
> Platão ficou admirado com a semelhança entre todos os fenômenos da natureza e chegou, portanto, à conclusão de que "por cima" ou "por trás" de tudo o que vemos à nossa volta há um número limitado de *formas*. A estas *formas* Platão deu o nome de *idéias*. Por trás de todos os cavalos, porcos e homens existe a "idéia cavalo", a "idéia porco" e a "idéia homem".

> (E é por causa disto que a citada padaria pode fazer broas em forma de porquinhos ou de cavalos, além de anõezinhos. Pois uma padaria que se preze geralmente tem mais do que uma fôrma. Só que uma única fôrma é suficiente para todo um tipo de broa.)
>
> Platão acreditava numa realidade autônoma por trás do *mundo dos sentidos*. A esta realidade ele deu o nome de *mundo das idéias*. Nele estão as "imagens padrão", as imagens primordiais, eternas e imutáveis, que encontramos na natureza. Esta concepção é chamada por nós de a *Teoria das Idéias de Platão*.
>
> Em resumo, para Platão a realidade se dividia em duas partes. A primeira parte é *o mundo dos sentidos*, do qual não podemos ter senão um conhecimento aproximado ou imperfeito, já que para tanto fazemos uso de nossos cinco (aproximados e imperfeitos) sentidos. Neste mundo dos sentidos, tudo "flui" e, consequentemente, nada é perene. Nada é no *mundo dos sentidos*; nele, as coisas simplesmente surgem e desaparecem. A outra parte é o *mundo das idéias*, do qual podemos chegar a ter um conhecimento seguro, se para tanto fizermos uso de nossa razão. Este mundo das idéias não pode, portanto, ser conhecido através dos sentidos. Em compensação, as *idéias* (ou *formas*) são eternas e imutáveis.
>
> Assim como os filósofos que o antecederam, Platão também queria encontrar algo de eterno e de imutável em meio a todas as mudanças. Foi assim que ele chegou às idéias perfeitas, que estão acima do mundo sensorial. Além disto, Platão considerava essas idéias mais reais do que os próprios fenômenos da natureza. Primeiro vinha a *idéia* cavalo e depois todos os cavalos do *mundo dos sentidos*. A *idéia* galinha vinha, portanto, antes da galinha e do ovo. (GAARDER, 1991, p.98).

Na citação acima é possível observar que Platão já observava as coisas como um objeto e que o mesmo pertencia a uma classe, conjunto, algo que era comum entre todos, algo que garante que jamais existiria problemas para reconhecer um cavalo, seguindo este mesmo princípio as linguagens orientada a objetos foram concebidas.

O termo orientação a objetos pressupõe uma organização de um programa em termos de coleção de objetos discretos incorporando estrutura e comportamento próprios. Esta abordagem de organização é essencialmente diferente do desenvolvimento tradicional de programas, onde estruturas de dados e rotinas são desenvolvidas de forma apenas fracamente acopladas.

3.2.1 Definições

Um objeto é uma entidade do mundo real que tem uma identidade. Objetos podem representar entidades concretas (um arquivo no meu computador, uma bicicleta) ou entidades conceituais (uma estratégia de jogo, uma política de

escalonamento em um sistema operacional). Cada objeto ter sua identidade significa que dois objetos são distintos mesmo que eles apresentem exatamente as mesmas caraterísticas. Embora objetos tenham existência própria no mundo real, em termos de linguagem de programação um objeto necessita um mecanismo de identificação. Esta identificação de objeto deve ser única, uniforme e independente do conteúdo do objeto. Este é um dos mecanismos que permite a criação de coleções de objetos, as quais são também objetos em si.

A estrutura de um objeto é representada em termos de atributos. O comportamento de um objeto é representado pelo conjunto de operações que podem ser executadas sobre o objeto. Objetos com a mesma estrutura e o mesmo comportamento são agrupados em classes. Uma classe é uma abstração que descreve propriedades importantes para uma aplicação e simplesmente ignora o resto.

Cada classe descreve um conjunto (possivelmente infinito) de objetos individuais. Cada objeto é uma instância de uma classe. Assim, cada instância de uma classe tem seus próprios valores para cada atributo, mas dividem os nomes dos atributos e métodos com as outras instâncias da classe. Implicitamente, cada objeto contém uma referência para sua própria classe – em outras palavras, ele sabe o que ele é.

Polimorfismo significa que a mesma operação pode se comportar de forma diferente em classes diferentes. Por exemplo, a operação *move* quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um método é uma implementação específica de uma operação para uma certa classe.

Polimorfismo também implica que uma operação de uma mesma classe pode ser implementada por mais de um método. O usuário não precisa saber quantas implementações existem para uma operação, ou explicitar qual método deve ser utilizado: a linguagem de programação deve ser capaz de selecionar o método correto a partir do nome da operação, classe do objeto e argumentos para a operação. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos.

No mundo real, alguns objetos e classes podem ser descritos como casos especiais, ou especializações, de outros objetos e classes. Por exemplo, a classe de computadores pessoais com processador da linha 80x86 é uma especialização de computadores pessoais, que por sua vez é uma especialização de computadores. Não é desejável que tudo que já foi descrito para computadores tenha de ser repetido para computadores pessoais ou para computadores pessoais com processador da linha 80x86.

Herança é o mecanismo do paradigma de orientação a objetos que permite compartilhar atributos e operações entre classes baseada em um relacionamento hierárquico. Uma classe pode ser definida de forma genérica e depois refinada sucessivamente em termos de subclasses ou classes derivadas. Cada subclasse incorpora, ou herda, todas as propriedades de sua superclasse (ou classe base) e adiciona suas propriedades únicas e particulares. As propriedades da classe base não precisam ser repetidas em cada classe derivada. Esta capacidade de fatorar as propriedades comuns de diversas classes em uma superclasse pode reduzir dramaticamente a repetição de código em um projeto ou programa, sendo uma das principais vantagens da abordagem de orientação a objetos.

3.2.2 Conceitos Básicos

A abordagem de orientação a objetos favorece a aplicação de diversos conceitos considerados fundamentais para o desenvolvimento de bons programas, tais como abstração e encapsulação. Tais conceitos não são exclusivos desta abordagem, mas são suportados de forma melhor no desenvolvimento orientado a objetos do que em outras metodologias.

3.2.2.1 Abstração

Abstração consiste de focalizar nos aspectos essenciais inerentes a uma entidade e ignorar propriedades “acidentais". Em termos de desenvolvimento de sistemas, isto significa concentrar-se no que um objeto é e faz antes de se decidir como ele será implementado. O uso de abstração preserva a liberdade para tomar decisões de desenvolvimento ou de implementação apenas quando há um melhor entendimento do problema a ser resolvido.

Muitas linguagens de programação modernas suportam o conceito de abstração de dados; porém, o uso de abstração juntamente com polimorfismo e herança, como suportado em orientação a objetos, é um mecanismo muito mais poderoso.

O uso apropriado de abstração permite que um mesmo modelo conceitual (orientação a objetos) seja utilizado para todas as fases de desenvolvimento de um sistema, desde sua análise até sua documentação.

3.2.2.2 Encapsulamento

Encapsulamento, também referido como esconder informação, consiste em separar os aspectos externos de um objeto, os quais são acessíveis a outros objetos, dos detalhes internos de implementação do objeto, os quais permanecem escondidos dos outros objetos. O uso de encapsulamento evita que um programa torne-se tão interdependente que uma pequena mudança tenha grandes efeitos colaterais.

O uso de encapsulamento permite que a implementação de um objeto possa ser modificada sem afetar as aplicações que usam este objeto. Motivos para modificar a implementação de um objeto podem ser, por exemplo, melhoria de desempenho, correção de erros e mudança de plataforma de execução.

Assim como abstração, o conceito de encapsulamento não é exclusivo da abordagem de orientação a objetos. Entretanto, a habilidade de se combinar estruturas de dados e comportamento em uma única entidade torna a encapsulação mais elegante e mais poderosa do que em linguagens convencionais que separam estruturas de dados e comportamento.

3.2.2.3 Compartilhamento

Técnicas de orientação a objetos promovem compartilhamento em diversos níveis distintos. Herança de estruturas de dados e comportamento permite que estruturas comuns sejam compartilhadas entre diversas classes derivadas similares sem redundância. O compartilhamento de código usando herança é uma das grandes vantagens da orientação a objetos. Ainda mais importante que a economia de código é a clareza conceitual de reconhecer que operações diferentes são na verdade a mesma coisa, o que reduz o número de casos distintos que devem ser entendidos e analisados.

O desenvolvimento orientado a objetos não apenas permite que a informação dentro de um projeto seja compartilhada como também oferece a possibilidade de reaproveitar projetos e código em projetos futuros. As ferramentas para alcançar este compartilhamento, tais como abstração, encapsulamento e herança, estão presentes na metodologia; uma estratégia de reuso entre projetos é a definição de bibliotecas de elementos reusáveis. Entretanto, orientação a objetos não é uma fórmula mágica para alcançar reusabilidade; para tanto, é preciso planejamento e disciplina para pensar em termos genéricos, não voltados simplesmente para a aplicação corrente.

3.2.3 Objetos e Classes

Objeto é definido neste modelo como um conceito, abstração ou coisa com limites e significados bem definidos para a aplicação em questão. Objetos têm dois propósitos: promover o entendimento do mundo real e suportar uma base prática para uma implementação computacional. Não existe uma maneira “correta" de decompor um problema em objetos; esta decomposição depende do julgamento do projetista e da natureza do problema. Todos objetos têm identidade própria e são distingüíveis.

Uma classe de objetos descreve um grupo de objetos com propriedades (atributos) similares, comportamento (operações) similares, relacionamentos comuns com outros objetos e uma semântica comum. Por exemplo, Pessoa e Companhia são classes de objetos. Cada pessoa tem um nome e uma idade; estes seriam os atributos comuns da classe. Companhias também podem ter os mesmos atributos nome e idade definidos. Entretanto, devido à distinção semântica elas provavelmente estariam agrupadas em outra classe que não Pessoa. Como se pode observar, o agrupamento em classes não leva em conta apenas o compartilhamento de propriedades. Todo objeto sabe a que classe ele pertence, ou seja, a classe de um objeto é um atributo implícito do objeto.

3.2.3.1 Atributos

Um atributo é um valor de dado assumido pelos objetos de uma classe. Nome, idade e peso são exemplos de atributos de objetos Pessoa. Cor, peso e modelo são possíveis atributos de objetos Carro. Cada atributo tem um valor para

cada instância de objeto. Por exemplo, o atributo idade tem valor “29" no objeto Pedro Y. Em outras palavras, Pedro Y tem 29 anos de idade. Diferentes instâncias de objetos podem ter o mesmo valor para um dado atributo.

Cada nome de atributo é único para uma dada classe, mas não necessariamente único entre todas as classes. Por exemplo, ambos Pessoa e Companhia podem ter um atributo chamado endereço.

Não se deve confundir identificadores internos de objetos com atributos do mundo real. Identificadores de objetos são uma conveniência de implementação, e não têm nenhum significado para o domínio da aplicação. Por exemplo, CIC e RG não são identificadores de objetos, mas sim verdadeiros atributos do mundo real.

3.2.3.2 Operações e Métodos

Uma operação é uma função ou transformação que pode ser aplicada a objetos, ou apliacadas por objetos em uma classe. Por exemplo, abrir, salvar e imprimir são operações que podem ser aplicadas a objetos da classe Arquivo. Todos objetos em uma classe compartilham as mesmas operações.

Toda operação tem um objeto-alvo como um argumento implícito. O comportamento de uma operação depende da classe de seu alvo. Como um objeto “sabe" qual sua classe, é possível escolher a implementação correta da operação. Além disto, outros argumentos (parâmetros) podem ser necessários para uma operação.

Uma mesma operação pode se aplicar a diversas classes diferentes. Uma operação como esta é dita ser polimórfica, ou seja, ela pode assumir distintas formas em classes diferentes.

Um método é a implementação de uma operação para uma classe. Por exemplo, a operação imprimir pode ser implementada de forma distinta, dependendo se o arquivo a ser impresso contém apenas texto ASCII, é um arquivo de um processador de texto ou binário. Todos estes métodos executam a mesma operação - imprimir o arquivo; porém, cada método será implementado por um diferente código.

A assinatura de um método é dada pelo número e tipos de argumentos do método, assim como por seu valor de retorno. Uma estratégia de

desenvolvimento recomendável é manter assinaturas coerentes para métodos implementando uma dada operação, assim como um comportamento consistente entre as implementações.

3.2.3.3 Generalização e Herança

Generalização e herança são abstrações poderosas para compartilhar similaridades entre classes e ao mesmo tempo preservar suas diferenças. Generalização é o relacionamento entre uma classe e uma ou mais versões refinadas (especializadas) desta classe. A classe sendo refinada é chamada de superclasse ou classe base, enquanto que a versão refinada da classe é chamada uma subclasse ou classe derivada. Atributos e operações comuns a um grupo de classes derivadas são colocadas como atributos e operações da classe base, sendo compartilhados por cada classe derivada. Diz-se que cada classe derivada herda as características de sua classe base. Algumas vezes, generalização é chamada de relacionamento *is-a* (é-um), porque cada instância de uma classe derivada é também uma instância da classe base.

Generalização e herança são transitivas, isto é, podem ser recursivamente aplicadas a um número arbitrário de níveis. Cada classe derivada não apenas herda todas as características de todos seus ancestrais como também pode acrescentar seus atributos e operações específicos.

Uma classe derivada pode sobrepor uma característica de sua classe base definindo uma característica própria com o mesmo nome. A característica local (da classe derivada) irá refinar e substituir a característica da classe base. Uma característica pode ser sobreposta, por exemplo, por questões de refinamento de especificação ou por questões de desempenho.

Entre as características que podem ser sobrepostas estão valores *default* de atributos e métodos de operação. Uma boa estratégia de desenvolvimento não deve sobrepor uma característica de forma inconsistente com a semântica da classe base.

4 GERADORES DE COMPILADORES

Atualmente, a implementação de compiladores é reforçada por sistemas geradores de compiladores, basicamente servem para economizar tempo de desenvolvimento e o trabalho “braçal” que seria a programação das fases de um compilador. Esses programas podem ser divididos em três categorias de geradores, que correspondem às fases da estrutura de um compilador:

- geradores de analisadores léxicos: geram automaticamente reconhecedores para os símbolos léxicos (palavras-chave, identificadores, operadores, etc.) a partir de especificações de gramáticas ou expressões regulares;
- geradores de analisadores sintáticos: geram reconhecedores sintáticos a partir de gramáticas livres de contexto. Inicialmente, a implementação da análise sintática consumia grande esforço na construção de compiladores;
- geradores de geradores de códigos: recebem como entrada, regras que definem a tradução de cada operação da linguagem intermediária para a linguagem de máquina. As regras devem incluir detalhes suficientes para possibilitar a manipulação de diferentes métodos de acesso a dados (como, uma variável pode estar em registradores, em memória ou na pilha da máquina). Em geral, instruções intermediárias são mapeadas para esqueletos que representam seqüências de instruções de máquina.

Existem vários geradores de compiladores no mercado, a maioria de uso gratuito, o que facilita o uso dos mesmos, porém algumas destas ferramentas carecem de documentação para a sua utilização, citaremos neste trabalho algumas das ferramentas mais conhecidas e aquelas que serão utilizadas para desenvolver este trabalho.

4.1 Lex e Yacc

O Lex e Yacc são duas ferramentas geradores de compiladores nativas do sistema operacional GNU/Linux, esses programas permitem que se faça o

parsing de linguagens complexas com facilidade. Isso é muito bom quando se deseja ler um arquivo de configuração, ou quer escrever um compilador para alguma linguagem que você (ou outra pessoa) tenham inventado. Os dois programas servem para propósitos diferentes, quando usados juntos realizam grandes feitos.

O Lex é responsável pela análise léxica do compilador, a partir de um programa fonte, o Lex busca no programa fonte cada símbolo que tenha algum significado, realizando assim alguma ação. Observe a descrição de um programa fonte do Lex na Tabela 4.

Tabela 4 - Código-fonte do Lex

Código	**Detalhes**
%{ #include <stdio.h> %}	Entre o par %{ %} são adicionadas as bibliotecas C que serão utilizadas, <stdio.h> por causa do printf.
%%	%% Inicio do Analisador, definição dos *tokens*
[0123456789]+ printf("NUMBER\n");	*Token* número, saída NUMBER;
[a-zA-Z][a-zA-Z0-9]* printf("WORD\n");	*Token* palavra, saída WORD;
%%	%% Fim do Analisador

O Yacc é responsável pela análise sintática do compilador, recebendo os *tokens* pré-processados do Lex e verificando se os mesmos cabem na gramática esperada, desse forma ao invés de executar o comando **`printf`** no Lex ao encontrar um *token* será retornado o nome do *token* para o Yacc, veja o exemplo na Tabela 5.

Tabela 5 - Código-fonte do Lex em conjunto com o Yacc

Código	**Detalhes**
%{ #include <stdio.h> %} %%	
[0123456789]+ return NUMBER;	Retorna o *token* NUMBER
[a-zA-Z][a-zA-Z0-9]* return WORD;	Retorna o *token* WORD
%%	

Agora basta ao Yacc verificar se o mesmo se encaixa na gramática descrita no arquivo fonte que será processado pelo Yacc.

4.2 Flex e Bison

O Flex é uma implementação do Lex por Vern Paxon e Bison é a versão GNU do Yacc, dessa forma eles fazem as mesmas coisas com algumas diferenças entre ambos, o Flex é responsável pela análise léxica e o Bison pela análise semântica, as versões mais recentes do Flex e Bison são compatíveis com as versões do Lex e Yacc, para maiores informações veja em http://dinosaur.compilertools.net/.

4.3 JavaCC e Jasmin

O programa JavaCC é uma gerador de analisador léxico e sintático que gera um código Java, dessa forma nosso compilador deve ser desenvolvido na linguagem Java, o JavaCC foi inicialmente desenvolvido pela Sun e atualmente o seu desenvolvimento fica por conta da Metamata Inc (Delamaro,2004).

Este gerador é definido por uma linguagem própria para descrição dos seus afazeres, tanto o analisador léxico como o sintático são descritos no mesmo arquivo, na análise léxica um token pode ser definido na forma de uma expressão regular, observe na Tabela 6.

Tabela 6 - Código-fonte do JavaCC para análise léxica

Expressão Regular Simples

TOKEN :
{
< CLASSE : “classe”>
| < SENAO : “senao”>
| < VIRGULA : “,”>
}

Expressão Regular Complexa

TOKEN :
{
< STRING_CONST : “\””(~[“\””,”\n”,”\r”])* “\””>
}
// constante string, “abcd badc”

Além de *tokens*, podem ser definidos para o analisador léxico quais caracteres ou expressões devem ser ignorados (SKIP) e também os *tokens* especiais que são *tokens* que não são passados para o analisador sintático, mas armazenados e recuperados a partir de um tokem nomal, veja na Tabela 7 a sintaxe do código JavaCC para a análise sintática.

Tabela 7 - Análise Sintática para uma calculador de expressões Matemáticas

Análise Sintática	

void Start():
{}
{
<NUMBER>
(
<PLUS>
<NUMBER>
)*
<EOF>
}
	Válida a entrada de expressões matemáticas, apenas, com a operação de adição:
	Ex: 5+8+45

Segundo Delamaro (2004), o JavaCC utiliza a técnica descendente recursiva de análise. Nessa abordagem cada não-terminal da gramática é implementado por meio de um método que, quando chamado, procura reconhecer na entrada a estrutura do não-terminal. Não existe *backtracking* (retrocesso), ou seja, caso um método seja chamado para reconhecer as produções de certo não-terminal e não consiga fazê-lo, não existe possibilidade de retornar a um ponto anterior na entrada e tentar uma outra opção, ou seja, uma outra produção que possa levar ao reconhecimento da entrada. Existem também algumas ferramentas inclusas no próprio JavaCC que auxiliam na geração da árvore sintática e das classes que fazem as visitas aos nós dessa árvore, o JJTree é um pré-processador para o arquivo da gramática (sem código associado às produções) que insere o código necessário para a criação da árvore sintática.

O Jasmin (Java *Assembler Interface*) é um programa que permite a criação de um arquivo de classe Java a partir de um arquivo fonte, seguindo assim o estilo de um montador para arquiteturas tradicionais, ou seja, Jasmin recebe um arquivo ASCII fonte com instruções da Máquina Virtual Java e produz um arquivo executável JVM (Máquina Virtual Java).

Delamaro (2004), afirma que o Jasmin possui uma sintaxe simples que reflete diretamente a maioria dos atributos de um arquivo de classe. Mesmo assim é uma ferramenta flexível e que se exime o programador da tarefa de formatar e gerar do arquivo de classe. Ele também não efetua muitas checagens sobre o código montado, sendo possível gerarem-se classes que não serão aceitas pela JVM. Por exemplo, ele não efetua consistências externas como verificar se outras classes, referenciadas na classe sendo gerada, realmente existem ou estão disponíveis. Mesmo consistências internas como verificar se uma instrução é sempre alcançada com a pilha na mesma altura – característica requerida pela JVM – não são efetuadas.

Maiores detalhes sobre essas ferramentas são apresentados nos próximos capítulos.

5 CORAL

5.1 Especificação da Linguagem

No desenvolvimento de um compilador, faz-se necessário a elaboração de uma linguagem de programação, onde serão definidos os comandos e funções da linguagem a serem interpretados pelo compilador, dessa forma foi definida a linguagem Coral Orientada a Objetos, semelhante a linguagem Java, porém mais simples.

As linguagens Orientadas a Objeto como especificado no capítulo 3 favorecem a aplicação de diversos conceitos considerados fundamentais para o desenvolvimento de bons programas, no presente trabalho alguns destes conceitos foram retirados da Coral, restando apenas os fundamentais, ou seja, os que garantem que a linguagem seja Orientada a Objetos, entre os conceitos fundamentais estão as classes, atributos e métodos.

- Classes: descreve um grupo de objetos com propriedades similares, comportamentos, relacionamentos comuns com outros objetos e uma semântica comum. A Figura 4 nos mostra um exemplo da classe **Pessoa**.

```
classe Pessoa
{
   <atributos>
   <metodos>
}
```

Figura 4 - Classe em Coral

- Atributos: Um atributo é um valor de dado assumido pelos objetos de uma classe, cada atributo tem um valor para cada instância de objeto, cada nome de atributo é único para uma dada classe, mas não necessariamente único entre todas as classes. Na Figura 5, a declaração da classe *Pessoa* mostra que toda pessoa possui um RG e CPF, porém cada uma possui valores diferentes entre si.

```
classe Pessoa
{
```

```
   inteiro rg,cpf;
   <metodos>
}
```

Figura 5 - Declaração de Atributos

- Métodos: Representado pela Figura 6, os métodos são grupos de instruções em uma classe, que define como os objetos da classe se comportarão. Os métodos são semelhantes a funções em outras linguagens, mas sempre precisam estar localizados dentro de uma classe.

```
classe Pessoa
{
   inteiro rg,cpf;
   inteiro exibir_dados()
   {
      <comandos>
   }
}
```

Figura 6 - Criação de Métodos

- Comandos: Para auxiliar no desenvolvimento de código implementado pela linguagem, veja os comandos:
    - Declaração: A declaração de variáveis tanto de classe (atributos), como locais é feita da mesma forma, em ambas o tipo da variável deve preceder o nome da mesma, na linguagem existem os tipos **inteiro** e **string**, sem contar com os tipos construídos que são as classes definidas pelo usuário. Observe um exemplo na Figura 7.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
      inteiro z,y;
      <comandos>
   }
}
```

Figura 7 - Declaração de Variáveis

    - Entrada de Dados: A alimentação de dados na linguagem é feita pelo comando **ler**, que pode ser visualizado na Figura 8, tendo como parametro a variável que irá receber

um valor fornecido pelo usuário através por meio da entrada padrão.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
      inteiro z,y;
      ler(z);
      <comandos>
   }
}
```

Figura 8 - Entrada de Dados

- Saída de Dados: Mostra ao usuário os resultados obtidos na programação, o comando **`exibir`** fornece ao usuário os resultados de variáveis, expressões numéricas ou de textos, como demonstra a Figura 9.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
      inteiro z,y;
      exibir(“Digite um valor”);
      ler(z);
      exibir(z*a+y);
      <comandos>
   }
}
```

Figura 9 - Saída de Dados

- Atribuição: Atribuição se resume em atribuir um certo dado ou variável para uma outra variável de mesmo tipo, o comando equivalente é o **=** representado na Figura 10.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
      inteiro z,y;
      exibir(“Digite um valor”);
      ler(z);
      y = (z * a) / 5;
   }
}
```

Figura 10 - Atribuição de Dados

- Seleção: Permite a escolha entre dois estados, um verdadeiro e outro falso, por meio de uma expressão

lógica, se a expressão for verdadeira é executado o bloco de comandos pertencentes ao **entao** caso contrario é executado o bloco de comandos pertencentes ao **senao**, a seleção é representada pelo comando **se**, como demonstra a Figura 11.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
      inteiro z,y;
      exibir(“Digite um valor”);
      ler(z);
      se(y<z) entao
      {
         <comandos>
      }
      senao
      {
         <comandos>
      }
   }
}
```

Figura 11 - Seleção

- Repetição: Representada pelo comando **para** na linguagem Coral, tem o significado de executar as mesmas instruções de um certo ponto ate outro, como podemos observar na Figura 12 a sua sintaxe é bem simples, definindo um valor inicial, um limite de repetições, e por fim, de que forma que essa repetição será incrementada.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
      inteiro z = 10;
      para(y=1; y<z; y=y+1)
      {
         <comandos>
      }
   }
}
```

Figura 12 - Repetição

- Parâmentros: Ao construir um método existe a liberdade de passar a ele parâmentros que serão úteis dentro do

seu escopo de funcionamento. Esses parâmetros que nada mais são que variáveis, devem ser precedidos de seus respectivos tipos de dados, na chamada basta informar a variável ou o dado desejado diretamente, como é mostrado na Figura 13.

```
classe Teste
{
   inteiro met_somar(inteiro a,
                     inteiro b)
   {
      inteiro s;
      s = a + b;
      Exibir(“A soma é ”+s);
   }

   inteiro inicio()
   {
      Teste var1;
      var1 = novo Teste();
      var1.met_somar(8,6);
   }
}
```

Figura 13 - Parâmetros

- Retorno: O comando **retorna** é utilizado nos métodos, serve para que o usuário possa retornar algum valor correspondete ao tipo do método, como mostra a Figura 14 que retorna o resultado de uma soma.

```
classe Teste
{
   inteiro a,b;
   inteiro met_soma()
   {
      inteiro z,y;
      exibir(“Digite um valor1”);
      ler(z);
      exibir(“Digite um valor2”);
      ler(y);
      retorna(z+y);
   }
}
```

Figura 14 - Retorno

- Comentários: Os comentários nos servem para explicar trechos do programa, como também para que linhas de código não sejam consideradas pelo compilador. Na Figura 15, pode-se observar a exclusão de uma linha de

código e logo abaixo a exclusão de um grupo ou trecho de código.

```
classe Teste
{
   inteiro a,b;
   inteiro met_imprimir()
   {
//    inteiro z,y;
      exibir(“Digite um valor1”);
/*    ler(z);
      exibir(“Digite um valor2”);
      ler(y);
*/
      retorna(z+y);
   }
}
```

Figura 15 – Comentários

- Construtor de classe: O construtor de classe é acionado no momento em que um objeto é instanciado, realizando assim uma operação nesse momento, a Figura 16 ilustra esse caso.

```
classe Teste
{
   inteiro a,b;
   construtor()
   {
      a = 0;
      b = 0;
   }

 inteiro inicio()
   {
      Teste var1;
      var1 = novo Teste();
  }
}
```

Figura 16 - Construtor de classe

5.2 Construção Léxica

A análise léxica tem como função receber um arquivo texto, o código-fonte, e quebrar a entrada em símbolos verificando se os mesmos pertencem ao alfabeto da linguagem. Essa fase foi implementada utilizando os Geradores de Compiladores JavaCC. Esse gerador utiliza uma linguagem própria onde é possível descrever o analisador Léxico e Sintático.

A especificação formal de uma linguagem pode ser feita a partir da definição da sintaxe da própria linguagem, como já mostrado nos exemplos anteriores. Entretanto, constuma-se também utilizar a BNF que é amplamente utilizada para este propósito. A partir da especificação, é possível começar a programar o JavaCC. A saída desse gerador nada mais é do que um conjunto de classes implementadas em Java que ao serem executadas, desenvolvem o analisador léxico e sintático. Graças à flexibilidade do JavaCC é possível na sua construção inserir código Java e desenvolver as chamadas para as demais fases do compilador.

O nome do *parser* (analisador) também deve ser definido e em seguida a classe correspondente ao *parser*, como demonstra a Figura 17, na linha 10 está o método principal da classe, onde é instanciada a variável analisadores do tipo *parser*, ou seja, **`coral`**. É nesse método que é tratado a entrada do arquivo a ser analisadada e a chamada do primeiro método que valida a nossa linguagem.

```
1  Arquivo coral.jj
2
3  PARSER_BEGIN(coral)
4  package coral;
5  import java.io.*;
6
7  public class coral
8  {
9    final static String Versao = "Compilador Coral";
10   public  static void main(String args[]) throws ParseException
11   {
12      String n_arquivo = "";
13      coral analisadores;
14 .
15 .
16 .
17 PARSER_END(coral)
```

Figura 17 - Início do Arquivo de Configuração do JavaCC

Após a definição do *parser*, o analisador léxico vai tomando forma, um dos primeiros passos é definir o que será ignorado no código-fonte, como espaços, marcações de arquivos, como a tecla *enter*, tabulações, entre outros. Além disso, consegue-se também eliminar os comentarios que podem ser inseridos na linguagem. Os caracteres ignorados são representados na programção pelo comando **`skip`**. Veja na Figura 18 algumas declarações para os caracteres ignorados da linguagem Coral, como por exemplo na linha 6, onde está sendo representado um comentario de várias linhas, dessa forma ele espera encontrar um

/* e chamar a função **multiplas_linhas_comentadas** que está na linha 16 da mesma tabela, nessa função é definida que logo após o **/*** ele deve encontrar para parar de ignorar o ***/** mais caso não encontre ele irá desprezando todos os outros caracteres encontrados.

```
1  SKIP :
2  {
3    " " | "\t"  | "\n"  | "\r"  | "\f"
4  }
5
6  SKIP :
7  {
8    "/*" : multiplas_linhas_comentadas
9  }
10
11 SKIP :
12 {
13   "//" : uma_linha_comentada
14 }
15
16 <multiplas_linhas_comentadas> SKIP:
17 {
18     "*/" : DEFAULT
19   | <~[]>
20 }
21
22 <uma_linha_comentada> SKIP:
23 {
24     <["\n","\r"]> : DEFAULT
25   | <~[]>
26 }
```

Figura 18 - Definindo os caracteres que serão ignorados pelo Compilador

O alfabeto da linguagem é definido por símbolos, conhecidos pelo nome de t*oken*. A definição dos *tokens* no JavaCC é feita de forma bem simples e rápida, e é dividida em palavras reservadas, símbolos especiais e operadores. A Figura 19 mostra as palavras resevadas, como já foi visto na especificação da linguagem Coral.

```
1  /* Palavras reservadas */
2  TOKEN :
3  {
4      < CLASSE: "classe" >
5    | < CONSTRUTOR: "construtor" >
6    | < ENTAO: "entao" >
7    | < SENAO: "senao" >
8    | < PARA: "para" >
9    | < SE: "se" >
10   | < INTEIRO: "inteiro" >
11   | < NOVO: "novo" >
12   | < EXIBIR: "exibir" >
13   | < LER: "ler" >
14   | < RETORNA: "retorna" >
15   | < STRING: "string" >
```

```
16   | < NENHUM: "nenhum">
17 }
```

Figura 19 - *Tokens* da linguagem

Ao se declarar o *token* **CLASSE** na linha 4 da figura anterior, percebe-se que é preciso demostrar como será a grafia desse *token*, que segue como **`classe`**.

Conhecidos como *tokens* também, os identificadores são as palavras que o usuário pode inserir na linguagem, como nome de variáveis, nome de classes, nomes de métodos, seguindo sempre um padrão quanto a sua grafia. Na linguagem, um identificador pode ser composto de letras e números, lembrando que nunca os números podem aparecer como o primeiro símbolo da linguagem, como mostrado na Figura 20.

```
1  /* Identificadores */
2
3  TOKEN :
4  {
5      < IDENT: <LETRA> (<LETRA>|<DIGITO>)* >
6    |
7      < #LETRA:["A"-"Z","a"-"z"] >
8    |
9      < #DIGITO:["0"-"9"] >
10 }
```

Figura 20 - Identificadores da linguagem

Os símbolos especiais são os mais usualmente utilizados em várias linguagens de programação, a saber, ponto, ponto e vírgula, parenteses, colchetes. Os operadores aritméticos e lógicos incluem a adição, subtração, divisão, entre outros. A Figura 21 abaixo resume o conjunto dos símbolos utilizados na linguagem Coral.

```
1  /* Símbolos especiais */
2
3  TOKEN :
4  {
5      < EPAREN: "(" >
6    | < DPAREN: ")" >
7    | < ECHAVE: "{" >
8    | < DCHAVE: "}" >
9    | < ECOLCH: "[" >
10   | < DCOLCH: "]" >
11   | < PVIRGULA: ";" >
12   | < VIRGULA: "," >
13   | < PONTO: "." >
14 }
```

```
15
16 /* Operadores */
17
18 TOKEN :
19 {
20     < ATRIBUI: "=" >
21   | < MAIOR: ">" >
22   | < MENOR: "<" >
23   | < IGUAL: "==" >
24   | < MENORI: "<=" >
25   | < MAIORI: ">=" >
26   | < DIFERENTE: "!=" >
27   | < SOMA: "+" >
28   | < SUBTRAI: "-" >
29   | < MULT: "*" >
30   | < DIVIDE: "/" >
31   | < PERCEN: "%" >
32 }
```

Figura 21 - Símbolos Especiais e Operadores

5.3 Construção Sintática

A construção sintática da linguagem Coral tem papel importante para todo o resto da implementação, ela é feita no JavaCC que utiliza a técnica descendente recursiva de análise, porém faz-se necessário a existência de algumas outras classes em Java para auxiliar nesta fase.

Nesta etapa será definido como foi feita a construção correta dos comandos, a junção dos *tokens*, bem como a criação de métodos em JavaCC para descrever passo-a-passo a síntaxe correta da linguagem Coral. É interessante ressaltar a opção que o JavaCC oferece para que se possa observar quais símbolos não-terminais estão sendo executados e quais os *tokens* que já foram consumidos.

O primeiro método, ou melhor, o primeiro não-terminal da gramática é o **`programa`**, na Figura 22 pode-se observar na linha 5 como se inicia um programa Coral, os colchetes indicam que podem existir ou não a lista de classes, pode ser um programa vazio, caso exista, é feita a chamada do não-terminal **`listaclasses`**, que irá consumir *tokens* e chamar outros não-terminais até concluir a análise por todo o arquivo até a marca de final de arquivo *EOF*.

```
1 void programa() :
2 {
3 }
4 {
5     [ listaclasses() ] <EOF>
6 }
```

Figura 22 - Construção do não-terminal **programa**

A Figura 23, ilustra como é a declaração do não-terminal **declclasses** que define a construção de uma classe em Coral, na linha 5 se espera o consumo do *token* **CLASSE**, representado pelo lexema no código-fonte **classe**, logo após, o *token* **IDENT** representado por quaisquer palavras que contenham letras e números como definido anteriormente, o que seria o nome da classe e por fim a chamada para o não-terminal **corpoclasse** que especifica toda a construção do corpo da classe.

```
1  void declclasses() :
2  {
3  }
4  {
5     <CLASSE> <IDENT> corpoclasse()
6  }
```

Figura 23 - Construção do não-terminal **declclasses**

A Figura 24 mostra a construção do **corpoclasse** que foi citada na figura anterior. Primeiramente, é necessário que a classe se inicie com chaves, então a construção exige o consumo do *token* **ECHAVE**, chave esquerda, logo após, permite a existência de outras classes, chamadas de classes aninhadas, pode ser que não exista, a partir desse ponto vêm a definição da classe em si, no qual se espera a declaração de atributos da classe, que pode não existir como demonstrado na linha 6 com o comando **( )***, em seguida é natural a declaração dos construtores de classe que também podem não estar presente na classes e por fim a construção dos métodos quer por fim pode não existir em uma classe. O não-terminal **DCHAVE** encerra a declaração da classe.

```
1  void corpoclasse():
2  {
3  }
4  {
5     <ECHAVE>
6        [listaclasses()]
7        (declaracao() <PVIRGULA>)*
8        (construtor())*
9        (metodos())*
10    <DCHAVE>
11 }
```

Figura 24 - Construção do não-terminal **corpoclasse**

Como já sabemos o código que o JavaCC gera vai consumindo o código-fonte escrito em Coral e a cada *token* consumido ele tenta encaixá-los à

construção definida. No caso do **corpoclasse** teríamos problemas no casamento dos *tokens* lidos e na construção definida, o programa gerado pelo JavaCC não saberia definir se o não-terminal seria o **declaracao** ou **metodos**, pois ambos possuem como *FIRST* o terminal **inteiro**. Uma declaração de atributo pode ser do tipo **inteiro**, **string** ou do tipo de uma classe, depois é esperado um identificador em seguida, pode se encontrar uma vírgula para um novo atributo, ou um ponto e vírgula indicando o final da declaração. Na declaração de um método, podemos ter o tipo do mesmo como **inteiro**, **string** ou do tipo de uma classe, depois é esperado um identificador seguindo de um parenteses **(**, fica claro então que só podemos identificar diferenças nas duas construções após o terceio símbolo da construção. Para esse problema existe o comando **LOOKAHEAD*(n)*** do JavaCC que gera um método que analisa n símbolos à frente para decidir se deve ou não tentar casar a entrada com o não-terminal **declaracao**, como é observado a aplicação do comando na Figura 25.

```
1  void corpoclasse():
2  {
3  }
4  {
5     <ECHAVE>
6        [listaclasses()]
7        (LOOKAHEAD(3) declaracao() <PVIRGULA>)*
8        (construtor())*
9        (metodos())*
10    <DCHAVE>
11 }
```

Figura 25 - Construção do não-terminal **corpoclasse** com **LOOKAHEAD**

5.3.1 Erros Sintáticos

Toda a construção gramatical garente apenas que um código-fonte correto seja analisado, ou seja, apenas construções prevista na nossa gramática serão associadas e aceitas. Caso o usuário tenha criado um código-fonte que contenha erros nas construções léxicas ou sintáticas, ocorreria simplesmente o termino da análise do compilador, pois o mesmo tentaria fazer o casamento das instruções e não conseguiria, abortando assim a sua execução. Cabe ao usuário então retornar ao código-fonte e corrigir o mesmo no ponto em que o problema foi apresentado. Porém esse processo pode ser entediante, se o número de erros sintáticos for grande, pois a cada execução do compilador, o usuário deve editar o

arquivo de entrada, localizar o erro, corrigi-lo e reiniciar o processo. Seria interessante se o compilador Coral fosse capaz de mostrar todos ou, pelo menos, diversos erros sintáticos de uma só vez, assim o número de interações compila-corrige seria reduzido, sendo assim o compilador deve ser capaz de recuperar-se do erro ocorrido e continuar na análise, quando um erro sintático ocorre, torna-se mais difícil decidir como continuar no reconhecimento do restante da entrada.

O JavaCC não oferece recursos próprios para o desenvolvedor implementar uma recuperação de erros eficiente, mais é bastante flexivo quanto a utlização da linguagem Java, possibilitando então que o método de ressincronização seja construido em Java e adicionado no código do JavaCC.

Tomando como exemplo a gramática descrita abaixo:

S -> aAcd

A -> gh

Considerando a antrada **agabcd**, o analisador sintático descendente recursivo realiza:

- uma chamada ao método **S** que consome o *token a* da entrada;
- uma chamada ao método **A** que consome **g**;
- o método **A** tenta achar na entrada um **h** que não está aí e, então, um erro sintático ocorre.

Segundo Delamaro(2004), a idéia do método de ressincronização é fazer com que o método **S** não seja afetado por esse erro sintático e que possa continuar a analisar a entrada. Porém, se o método **A** simplesmente emitir uma mensagem de erro e retornar a execução para o método que o chamou, um outro erro sintático irá ocorrer, pois o método **S** espera que depois da chamada a **A** exista na entrada o *token* **c**, o que não ocorre, visto que ainda temos na entrada a cadeia **abcd**.

Por isso o método **A**, ao detectar um erro sintático, deve ressincronizar a entrada com o não-terminal que se espera reconhecer. Uma tentativa de se fazer isso é consumindo *tokens* da entrada até que apareça algum que possibilite a continuidade da análise. Mas quais seriam esses *tokens*? Uma boa idéia seria

utilizar o conjunto *FOLLOW* de **A**. Assim, o método **A** deve consumir *tokens* de entrada até que apareça um símbolo pertecente ao seus conjunto FOLLOW e só depois retornar a execução para **S**. Em, tudo se passa como se a execução de **A** tivesse sido bem-sucedida e a análise continua, com a certeza de que, pelo menos, o próximo *token* da entrada irá casar com a produção sendo utilizada.

O JavaCC permite que as produções na BNF sejam colocadas entre construções **try/catch** e caso algum erro sintático seja destectado ao tentar casar a entrada com essas produções, podemos tratar esse erro. Um erro sintático faz com que uma exceção do tipo **ParseException** seja lançada, como demosntra a Figura 26. A produção foi colocada dentro da construção **try/catch** e, portanto, se alguma exceção for propagada pelo método **listaclasses** ou se não existir um *EOF* após a lista de classes, o código dentro do **catch** será executado. É valido lembrar que esse código é Java, e não produções BNF. No caso é feito chamadas ao método **consumeAte** que foi incluido no arquivo *Coral.jj* do JavaCC e, portanto, incluído como um método da classe **coral**.

```
1  void programa() throws ParseEOFException:
2  {
3     RecuperaSet g = new RecuperaSet(EOF);
4  }
5  {
6     try
7     {
8        [ listaclasses(g) ] <EOF>
9     }
10    catch(ParseException e)
11    {
12       consumeAte(g, e, “programa”);
13    }
   }
```

Figura 26 - Não-terminal **programa** com tratamento de erros

Analisando o método **consumeAte** que possui como primeiro parâmetro a variável **g** do tipo **RecuperaSet**, onde representa o conjunto de sincronização a ser utilizado, a classe **RecuperaSet** está definida dentro do pacote **recuperacao**, representado pela Figura 27, no qual implementa um conjunto em que cada elemento é um objeto do tipo **java.lang.Integer**, que representa um tipo de *token* (**CLASSE** ou **IDENT**, etc). Os seus métodos representam as operações normais sobre conjuntos normais como união, pertinência de uma elemento, etc.

O segundo parâmetro do método **consumeAte** é um objeto do tipo **ParseException**, que descreve qual foi o erro sintático que ocorreu. Esse parâmetro é extraído do **catch** no método que chama o **consumeAte**. E o último parâmetro é o nome do não-terminal que está fazendo a recuperação de erros. No caso do não-terminal inicial da nossa gramática, tem-se o conjunto de sincronização formado por um único *token*, que é o *EOF*, e o terceiro parâmetro é obviamente uma string com o nome do não-terminal **programa**.

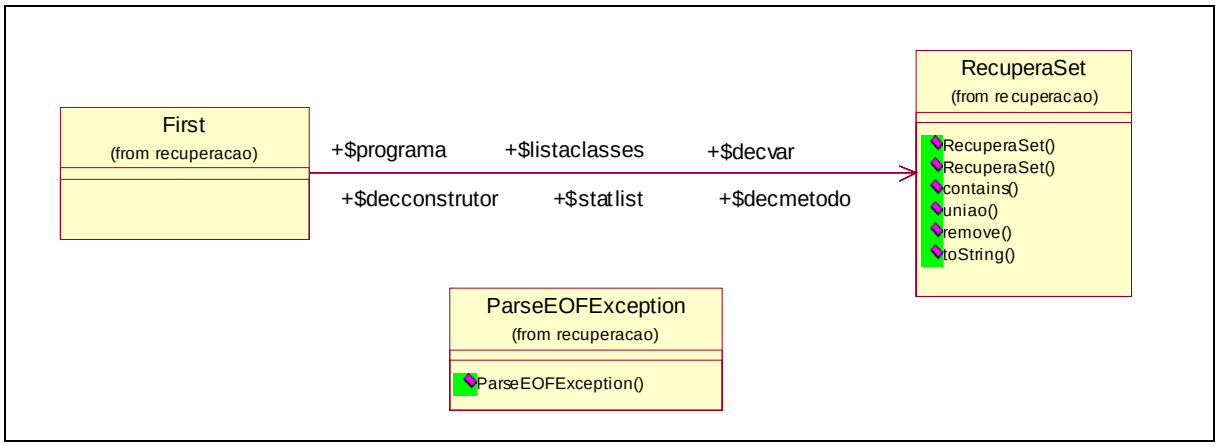

Figura 27 - Diagrama de classes do pacote **recuperacao**

O não-terminal **listaclasses**, Figura 28, não muda muito em relação ao definido anteriormente, ele chama o **declclasses** passando como o conjunto de sincronização o conjunto **g**, que recebeu como parâmetro, e mais todos os terminais que estão no *FIRST*(**listaclasses**). Isso porque depois do **declclasses** tem-se um **listaclasses** ou, então, o final da aplicação dessa produção. Nesse último caso, o conjunto de sincronização deve ser o mesmo do não-terminal corrente. Situação semelhante ocorre na chamada para **listaclasses**, seu conjunto de sincronização é igual ao do método corrente. Isso se justifica, pois se ocorrer um erro dentro da chamada mais interna, o não-terminal que foi chamado deverá procurar um terminal que está definido na produção corrente, mas, sim, numa produção do não-terminal que fez a chamada. Utiliza-se nesse não-terminal o conjunto **First.listaclasses**, esse conjunto foi definido dentro do pacote **recuperacao**. A classe **First** é uma classe que possui diversas variáveis estáticas que representam alguns dos conjuntos *FIRST* que são usados na recuperação de erros.

```
1  void listaclasses(RecuperaSet g) throws ParseEOFException:
2  {
3     RecuperaSet f = First.listaclasses.uniao(g);
4  }
5  {
6     declclasses(f) [ listaclasses(g) ]
7  }
```

Figura 28 - Não-terminal **`listaclasses`** com recuperação de erros

5.3.2 Árvore Sintática

O analisador de um compilador deve, além de identificar se a entrada fornecida pertence ou não à linguagem de programação desejada e reportar os possíveis erros existentes, produzir como saída uma representação do programa que irá servir para que as fases sucessivas do compilador possam ser executadas, essa representação é a árvore sintática ou árvore de derivação. Observando o código

```
classe teste
{
     ...
}
```

com esse exemplo de código, fica fácil observar como seria a árvore sintática, ilustrada na Figura 29, correspondente ao código anterior. No topo da árvore encontra-se o primeiro não-terminal, **`programa`**, que deriva em **`listaclasses`**, em seguida **`declclases`** e por fim encotra o *token* **`classe`** o nome da classe e a chamada ao não-terminal **`corpoclasse`**, finalizando com o encontro das chaves que delimitam o início e o fim do corpo de uma clase.

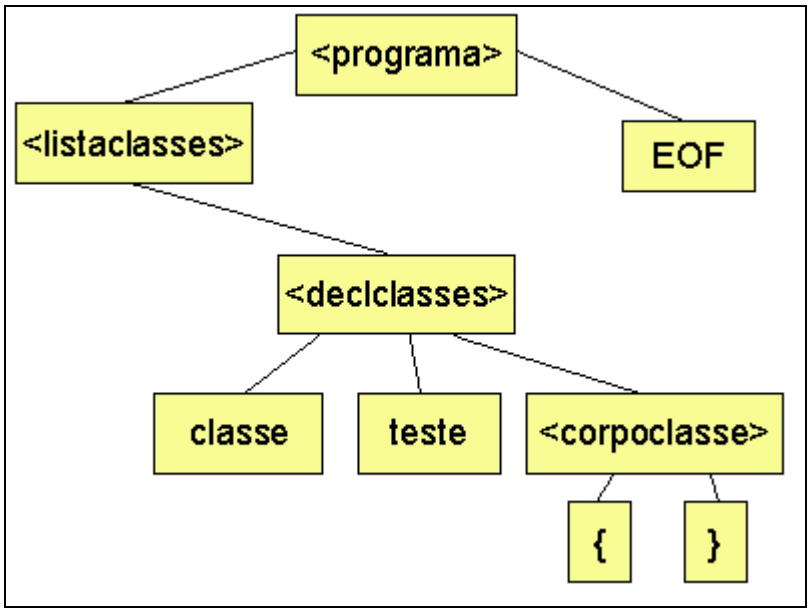

Figura 29 - Exemplo de árvore sintática

A construção da árvore sintática neste compilador ocorre à medida que a análise sintática se processa. Cada execução de um dos métodos associados aos não-terminais cria um nó da árvore e "pendura" nele os nós filhos. Esses nós filhos são formados pelos *tokens* reconhecidos naquele método e pelos os outros nós criados pelas chamadas a outro métodos de não-terminais. Por exemplo, uma chamada ao não-terminal **`declclasses`** irá criar um nó que tem como filhos cada um dos *tokens* que ele consumiu (como **`IDENT`**) e um outro nó criado pela chamada ao não-terminal **corpoclasse**. Dessa forma, embora a análise sintática seja descendente, a construção da árvore sintática é feita de baixo para cima, ou seja, dos nós mais profundos, em direção à raiz. Cada nó da árvore será representado por um objeto Java, esse mesmo objeto, possui referências a outros objetos, dependendo do nó a ser representado uma classe é responsável pelo seu gerenciamento.

Todas as classes são declaradas dentro do pacote **`arvoreSintatica`** e as mesmas são subclasses de **`GeralNo`** como demosntra a Figura 30.

```
1  package arvoreSintatica;
2  import coral.*;
3
4  abstract public class GeralNo
5  {
6      public Token position;
7      public int number;
8
9      public GeralNo(Token x)
```

```
10      {
11          position = x;
12          number = 0;
13      }
14  }
```

Figura 30 - Classe **GeralNo**

A construção da árvore será feita através do código programado para o JavaCC, cada método correspondente aos não-terminais retornará o nó que construiu, permitindo que o nó que fez a chamada possa utilizar esse nó, os *tokens* consumidos pelo analisador será utlizado para construir o nó da árvore, atribuido o *token* a uma variável e depois utilizando a mesma na construção do nó, a Figura 31 ilustra o não terminal **declclasses** pronto para construção a árvore.

```
1   DecClasseNo declclasses(RecuperaSet g) throws ParseEOFException :
2   {
3      Token t = null, n = null;
4      CorpoClasseNo c = null;
5   }
6   {
7      try
8      {
9         t = <CLASSE>  n = <IDENT> c = corpoclasse(g)
10        {return new DecClasseNo(t,n,c);}
11     }
12     catch (ParseException e)
13     {
14        consumeAte(g, e, "declclasses");
15        return new DecClasseNo(t,n,c);
16     }
17  }
```

Figura 31 - Construção do não-terminal **declclasses** para árvore sintática

A assinatura do método foi alterada e agora deve retornar um objeto da classe **DeclClasseNo**, que pertence ao pacote **arvoreSintatica**, a variável **t**, na linha 3, do tipo *token* armazena o *token* que casou com o não-terminal **CLASSE** da produção, esse *token* será utilizado como referência desse nó, a variável **n** contém o nome da classe sendo declarada, informação importante para o compilador, por fim a variável **c** corresponde a um **CorpoClasseNo**, que é retornado pelo método **corpoclasse**, com esses elementos, constrói-se um nó **DeclClasseNo** no códiogo Java que foi associado com a chamada **corpoclasse**, qualquer chamada a outro não-terminal ou qualquer *token* pode ter um trecho de código associado a si. Basta colocar o código entre chaves logo após qualquer um desses elementos . A classe **DeclClasseNo** é bastante simples, como em geral

são todas as classes do pacote **arvoresSintatica**, a classe pode ser observada na Figura 32.

```
1  package arvoreSintatica;
2  import coral.*;
3
4  public class DecClasseNo extends GeralNo
5  {
6      public Token name;
7      public CorpoClasseNo body;
8
9      public DecClasseNo(Token t1, Token t2, CorpoClasseNo c)
10     {
11         super(t1);
12         name = t2;
13         body = c;
14     }
15 }
```

Figura 32 - Classe **DeclClasseNo**

5.3.3 Tabela de Símbolos

A tabela de símbolos se mostra como umas das mais importantes estruturas do compilador, essa estrutura reúne os nomes dos diversos elementos utilizados no código-fonte, como classes, variáveis e métodos. Informações sobre cada nome, ou identificador, utilizado. Assim, ao analisar uma declaração do tipo

```
classe Empregado
{
  ...
}
```

o compilador deve ser capaz de reconhecer que **Empregado** está sendo declarado como uma classe e quando uma outra declaração como

```
Empregado p1;
```

aparecer, ele poderá reconhecê-la como válida, pois **Empregado** é um tipo de dado conhecido.

Além disso, deve-se também armazenar algumas informações importantes, relacionadas aos identificadores. Para cada tipo de identificador, as informações a serem armazenadas diferem. Por exemplo, para

- classes: nome, quais são as variáveis dessa classe, quais são as classes aninhadas, quais são os métodos e construtores;

- variáveis: nome, tipo, dimensão no caso de vetores, se são locais ou não;
- métodos: nome, parâmetros, variáveis locais, tipo de retor

Com a reunião dessas informações, a tabela de símbolos auxilia a análise semântica e a geração de código de diversas maneiras, onde:

- ao declarar a classe

  ```
  classe Empregado ...
  ```

  é necessário saber se **Empregado** ainda não foi declarado;

- ao declarar a variável

  ```
  Empregado y;
  ```

  é necessário saber se **Empregado** é um tipo ou classe válido e se **y** pode ser declarada nesse ponto;

- ao declarar o método

  ```
  Empregado imprimir(inteiro k, string j);
  ```

  é necessário saber se **Empregado** é um tipo ou classe válido, se **imprimir** pode ser declarado nesse ponto e se seus parâmetros são legais;

- ao utilizar o comando

  ```
  var = emp.salario(10, b);
  ```

  é preciso saber se a classe de **emp** possui o método **salario** com dois parâmetros, qual é o tipo de retorno do método **salario** e se os argumentos utilizados na sua chamada combinam com os parâmetros declarados e se os dois lados da atribuição possuem o mesmo tipo ou tipos compatíveis.

Esses são alguns dos inúmeros casos em que se faz necessário as informações da tabela de símbolos.

Abaixo, na Figura 33 encontra-se uma visão abrangente sobre o diagrama de classes do pacote implementado para a tabela de símbolos.

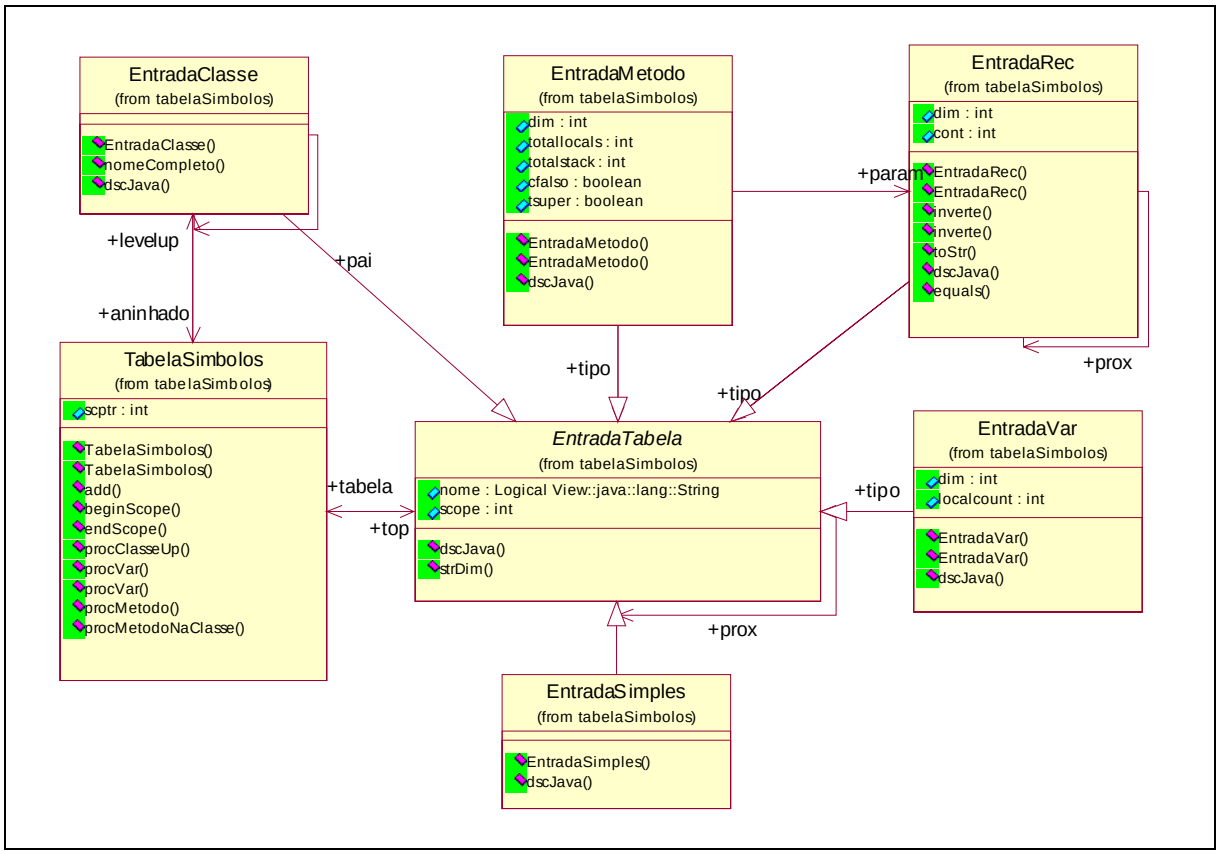

Figura 33 - Diagrama de classes do pacote **`tabelaSimbolos`**

5.4 Construção Semântica

A análise semântica fica com a responsabilidade de verificar os aspectos do programa relacionados ao significado de cada comando, um código-fonte pode estar correto de acordo com as regras de sintaxe definidas pela gramática, mas apresentar problemas com a semântica da linguagem. Por exemplo:

```
inteiro a,b;
string c;
c = a;
```

Na linguagem Coral, de acordo com as regras sintáticas está tudo correto, porém semanticamente não é permitido associações de tipos diferentes, como uma **`string`** receber um número **`inteiro`**, essa verificação é feita com um passeio pela árvore sintática, que é realizada em diversas fases, a cada fase a árvore sintática e percorrida e dados são levantados para serem utilizados na fase seguinte. O ideal seria que a análise fosse feita em um único percurso pela árvore

sintática, economizando tempo de compilação, porém isso é bastante complicado, considerando que para se fazer a verificação semântica propriamente dita é necessário previamente a colheita de informações sobre as classes declaradas.

As fases de análise podem ser divididas em três etapas, como descrito a seguir:

- Primeira Etapa
    - Análise da declaração de classes: tem como objetivo a criação da tabela de símbolos e análise da árvore sintática à procura das declarações de classes, dessa forma a árvore sintática não precisa ser totalmente percorrida, somente os seus níveis mais altos, com intuito de encontrar apenas as classes, ao encontrar tal nó, o analisador semântico deve incluir a classe correspondente na tabela de símbolos.
- Segunda Etapa
    - Análise da declaração de variáveis, métodos e construtores: quanto as variáveis, trata das declarações das variáveis da classe, ao encontrar uma variável é verificado o seu tipo de declaração, uma busca fica responsável por tentar encontrar na tabela de símbolos a classe utilizada ou um tipo básico, como **`inteiro`** ou **`string`** que já foram inseridos no topo da tabela de símbolos, no caso da classe da declaração não for encontrada, um erro ocorre, caso contrário, para cada variável na declaração é inserida uma entrada na tabela de símbolos corrente, indicando seu nome e tipo. Quanto aos métodos e contrutores, para cada declaração de um deles, inicialmente são analisados os seus parâmetros declarados, verificando se os tipos correspondentes existem.
- Terceira Etapa

- Checagem de tipos: verificar as associações feitas entre variáveis e dados, de forma que os tipos das mesmas sejam compatíveis, essa etapa envolve diretamente o análise dos comandos e das expressões associadas ao código-fonte.

Estas etapas poderão ser melhor visualizadas no diagrama de classes ilustrado na Figura 34.

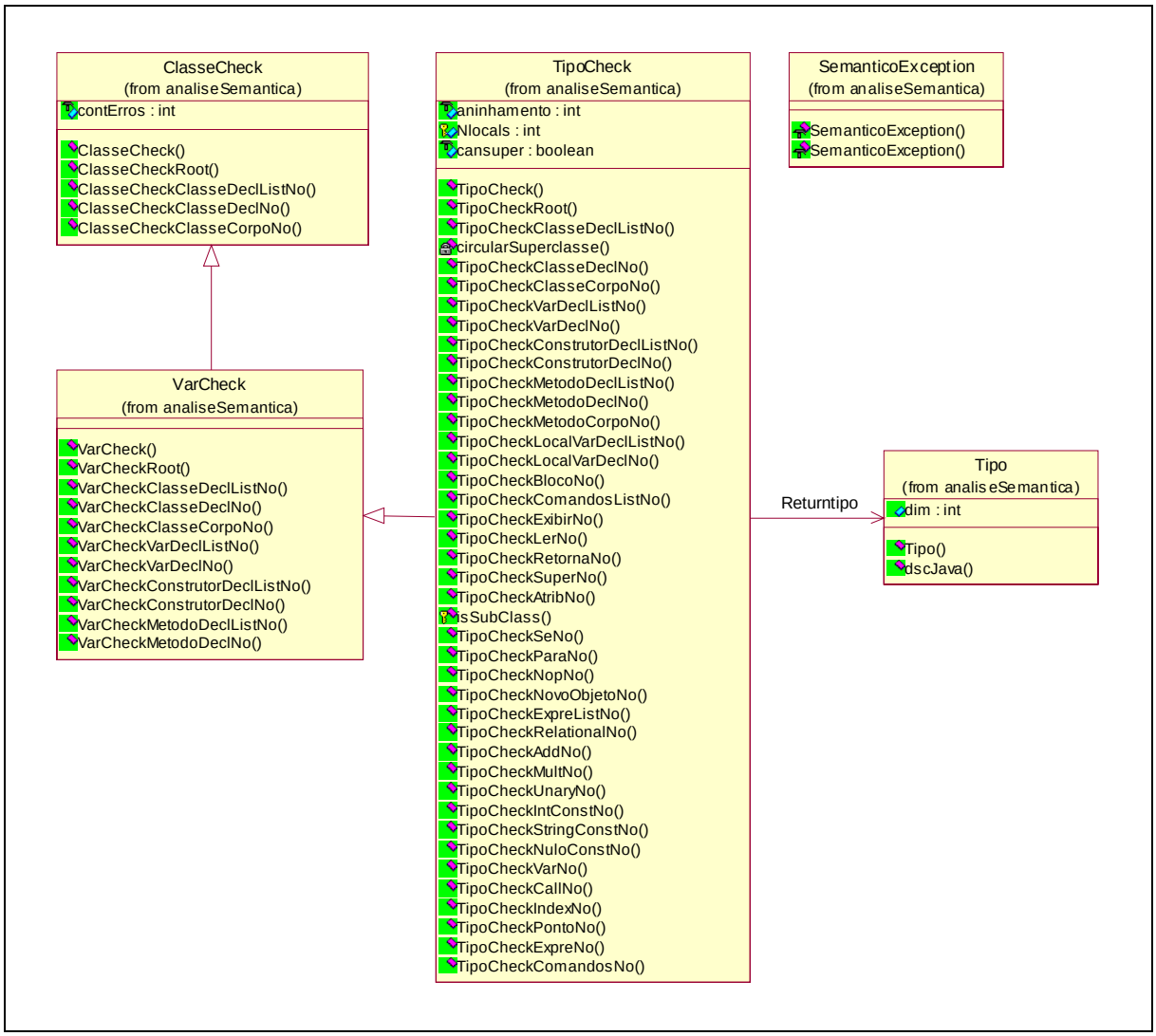

Figura 34 - Diagrama de classes do pacote **`analiseSemantica`**

5.5 Geração de Código

A geração de código é praticamente a fase final de um compilador, dessa forma é importante demostrar a verdadeira importância dessa fase, como ilustra a Figura 35 que demonstra toda a trajetória percorrida pelo programa fonte, passando pelo tratamento da fase de análise e finalizando na síntese, onde é devolvido como produto final uma arquivo executável da JVM, também conhecido como *class file*, nesse arquivo estão reunidos descrições sobre a classe em si, variáveis de instância, variáveis de classe, métodos e construtores dessa classe.

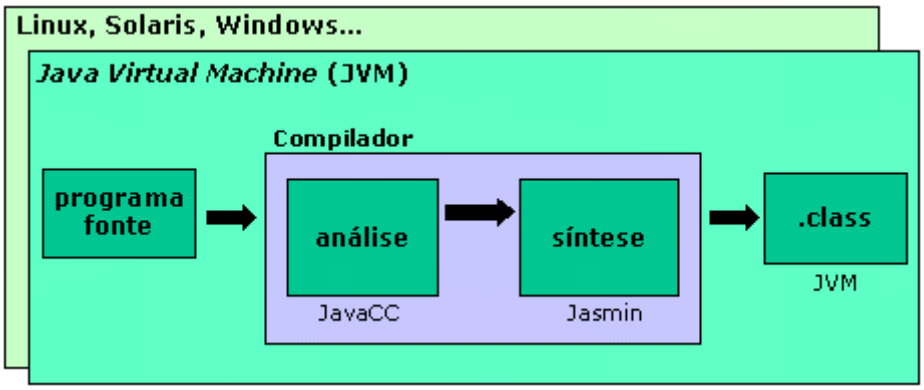

Figura 35 - Esquema do Compilador para a linguagem Coral

Segundo Delamaro(2004), a JVM possui 227 instruções definidas para a construção de arquivos de classes, com minuciosos detalhes, o que torna a construção de um arquivo de classe seja bastante trabalhoso. Para facilitar essa tarefa, existem alguns programas e bibliotecas que oferecem uma interface mais agradável para a programação, como é o caso do Jasmin, que é um montador para a JVM, através de um arquivo texto, que descreve a classe entre outras coisas, transforma o mesmo em um arquivo executável, o *class file*.

Para descrever os componentes de classe, o Jasmin define um conjunto de diretivas entra elas a diretiva .**`source`**, .**`class`** e .**`super`**. A primeira diz qual o nome do código-fonte que originou o arquivo de classe, o segundo é o nome da classe que está descrita no arquivo e a terceira, o nome da superclasse, no qual será sempre a **`java/lang/Object`** uma que vez que toda classe deriva de **`java.lang.Object`**.

A Tabela 8 ilustra a representação dos tipos de variáveis da linguagem Coral para os tipos da JVM, uma variável do tipo **`inteiro`** em Coral seria representado por **`I`** na JVM, com uma **`string`**, o **`Ljava/lang/String`** e com uma classe a adição do símbolo **`L`** ao nome da classe.

Tabela 8 - Representação dos tipos em Coral para a JVM

Tipo	Representação
`inteiro`	`I`
`string`	`Ljava/lang/String;`
`MinhaClasse`	`LminhaClasse`

A diretiva .**field** é utilizada para definir uma variável não local, as variáveis que em Java seriam definidas como

```
public int var;
static private Vector vetor;
```

em Jasmin seriam definidas como

```
.field public var I
.field private static vetor Ljava.util.Vector
```

Duas diretivas definem os métodos da seguinte forma

```
.method static public main([Ljava/lang/String;) V
  ; instruções
.end method
```

No caso de um construtor de classe

```
construtor(int a)
```

em Jasmin ficaria

```
.method public <init> (I)V
```

A impressão de dados na tela que em Coral é definida como

```
exibir(“Olá Mundo!”);
```

em Jasmin se resume em

```
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc “Olá Mundo!”
invoke virtual
java/io/PrintStream/println(Ljava/lang/String;)V
```

A geração de código é baseada na visita aos nós da árvore sintática, é feita pela classe **GeracaoCodigo**, que embora faça parte do pacote **geracaoCodigo** é uma subclasse do analisador semântico, uma vez que as informações computadas durante a análise semântica, em particular a tabela de símbolos, são utilizados na geração de código. Associando então os nós

encontrados na árvore sintática com suas respectivas características da tabela de símbolos com os comandos correspondetes do Jasmin, gerando assim um arquivo de extensão *.jas* com todas as informações da classe, dessa forma o Jasmin compila esse arquivo gerando então o arquivo executável da JVM, o *class file* de extensão *.class* que pode ser executado em qualquel sistema operacional desde que o mesmo possua a JVM instalada.

5.6 IDE

Um compilador contém, na sua essência, apenas processamento “interno”, ou seja, não é necessário que se tenha uma interface para que o compilador funcione a contento. O mesmo então apenas recebe um arquivo, onde contém a descrição do código-fonte a ser processado, e responde como saída com a criação de um arquivo executável, que reproduz todas as exigências descritas no código-fonte.

Para facilita e tornar o processo de desenvolvimento de softwares mais ágil e produtivo, foram criadas as IDE’s (*Integrated Development Enviroment*), ou seja, Ambiente de Desenvolvimento Integrado, onde o desenvolvedor conta com uma série de ferramentas para lhe auxiliar na construção de seu software, como um editor de texto, que funcione de acordo com as características pertinentes a linguagem utilizada, acesso a debugação de erros, onde se pode acompanhar passo a passo a execução de um software, entre outras funções. Como exemplo de IDE’s temos: Eclipse, IBM WSAD, Borland JBuilder, Microsoft Visual Studio, Delphi, entre outros.

Visando facilitar a escrita, compilação e execução de um programa Coral, foi construido uma IDE para a mesma, ilustrada na Figura 36, no qual possui funções simples, mais que serão utéis para aqueles que vão utilizar a ferramenta.

O editor de texto, marca em negrito todas as palavras reservadas da linguagem, auxiliando a visualização e a construção dos comandos. Para compilar o código construído, basta selecionar a janela onde o mesmo está presente, e no menu **`Executar`**, existe a opção de **`Compilar`** que fará a compilação do programa, listando na janela **`Console`** todas as informações pertinentes aos resultados da compilação, como número de erros Léxicos, Sintáticos, Semânticos e se eles

existirem em qual linha ocorreu, informa também o sucesso da geração do arquivo executável, no qual pode ser iniciado pelo menu **Executar**, na opção **Iniciar**.

Uma descrição de toda a linguagem Orientada a Objetos Coral, se encontra no menu **Ajuda** da IDE.

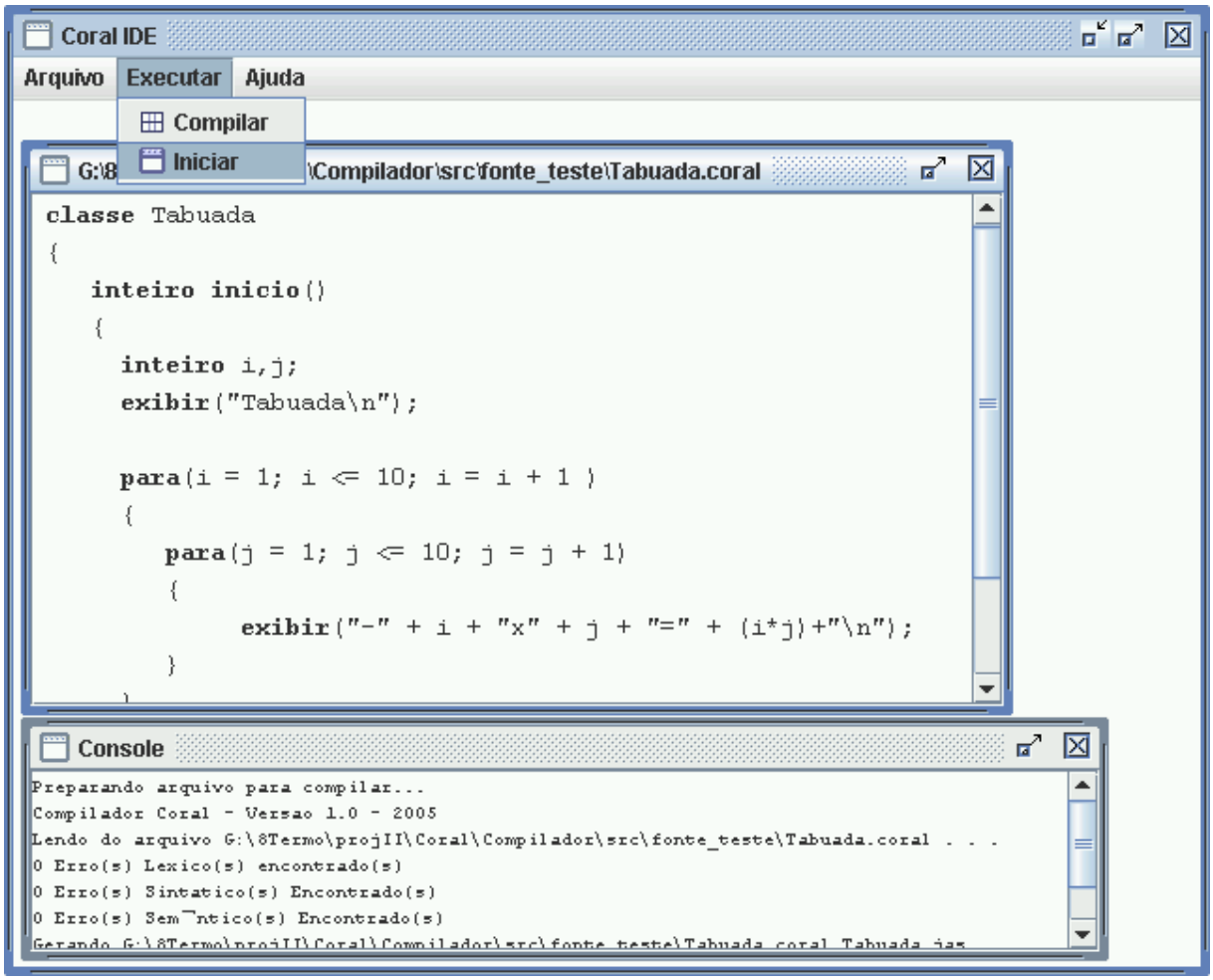

Figura 36 - IDE Coral

6 CONCLUSÃO

Este projeto teve como maior incetivo a possiblidade do compilador gerar um arquivo executável, resultante de um código-fonte escrito na linguagem Coral, como Takehana(2004) havia sugerido como extensão de seu trabalho e a oportunidade da pesquisa sobre o paradigma de programação Orientado a Objetos, já que o mesmo peca em escassez de bibliografia quanto ao seu funcionamento como compilador.

Durante a pesquisa surgiu a possibilidade de se trabalhar com os geradores de compiladores, já citados neste trabalho, com o intuito de avaliá-los, quanto ao seu desempenho e auxílio na construção de compiladores, o que possibilitou a geração do código executável com a ferramenta Jasmin. A partir desse ponto, com todas as ferramentas definidas e a metodologia de trabalho coerente, foi possível obter respostas quanto as semelhanças entre o paradigma Orientado a Objetos e o paradigma Imperativo de programação, que se acentuaram em torno da definição teórica do funcionamento de ambos, e do modelo de estrutra adotado por Aho (1986). As diferenças entre esses paradigmas também ficou bastante clara e compreendida durante a pesquisa, principalmente quando se trata da complexidade envolvida na construção gramatical de uma linguagem Orientada a Objetos, que nada tem de semelhante a de uma linguagem Imperativa, isso reflete na construção e no tratamento das operações nas fases de análise e síntese do compilador.

Quanto aos Geradores de Compiladores, sua utilização foi bastante vantajosa, pois sem o mesmo, a construção de um compilador de tal grau de complexidade, seria bastante limitada, já que o mesmo oferece, de forma simples, a possibilidade de se construir gramáticas complexas e funções que maximizam o desempenho de um compilador, tornando o mesmo mais eficiente, além de tornar o desenvolvimento muito mais rápido, o que é interessante para desenvolvedores. O resultado apresentado pelo gerador é o código-fonte em Java pronto para ser executado, no caso da fase de análise, o código-fonte resultante é bastante complexo e demanda tempo para extrair informações do mesmo, para ajustes e retoques pessoais. Já na fase de síntese, é importante observar que, com o uso da máquina virtual JVM se torna muito mais fácil a implementação do código para a máquina-alvo, lembrando que se este fosse feito para o montador *Assembler*, o grau

de complexidade seria bastante alto, devido ao problema da otimização dos registradores, o que não acontece no Java Assembler, já que a codificação segue o padrão Java e o montador Jasmin se encarrega de trabalhar com endereços de memória e outras funções.

6.1 Trabalhos Futuros

Para futuros pesquisadores, como extensão deste trabalho, podem ser citadas as seguintes sugestões:

- Implementar todas as características do paradigma Orientado a Objetos, herança, polimorfismo, encapsulamento entre outros, já que neste trabalho foram implementadas as classes, atributos e métodos. Tendo em vista, que o maior objetivo era a geração do programa executável a partir da geração de código.
- Continuidade no desenvovimento da IDE, com a implementação de um *debug* para a linguagem Coral.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO. A. V.: SETHI, R. & ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. Guanaban Koogan 1986, 1995

DELAMARO, Márcio Eduardo. **Como construir um compilador, utilizando ferramentas Java**. Novatec. 2004.

GAARDER, Jostein. **O mundo de Sofia, Romance da história da filosofia**. Cia. Das Letras 1991, 1995.

PRICE, ANA MARIA de A.; TOSCANI, Simão S. **Implementação de linguagens de programação: Compiladores**. Sagra Luzzatto. 2001.

SEBESTA, Robert W. **Conceitos de Linguagem de Programação**. Bookman 1999, 2000.

TAKEHANA, Aline Miyamura. **CLI: Compilador para uma Linguagem Imperativa**. Presidente Prudente, 2004. Monografia (Bacharelado em Ciência da Computação). UNOESTE - Universidade do Oeste Paulista