CENTRO UNIVERSITÁRIO FEEVALE

LUCAS ESKEFF FREITAS

A ELABORAÇÃO DE ESTRUTURAS DE MAPEAMENTO FINITO COMO EXTENSÃO PARA O COMPILADOR VERTO

Novo Hamburgo, junho de 2010.

LUCAS ESKEFF FREITAS

A ELABORAÇÃO DE ESTRUTURAS DE MAPEAMENTO FINITO COMO EXTENSÃO PARA O COMPILADOR VERTO

Centro Universitário Feevale
Instituto de Ciências Exatas e Tecnológicas
Curso de Ciência da Computação
Trabalho de Conclusão de Curso

Professor Orientador: Ricardo Ferreira de Oliveira

Novo Hamburgo, julho de 2010.

AGRADECIMENTOS

Aqui deixo todo meu carinho pela minha família que sempre me apoio nessa jornada, minha namorada Manuela que sempre esteve junto comigo em todos os momentos, e que sem eles não chegaria onde estou hoje.

E um agradecimento ao professor e orientador Ricardo pelo apoio na conclusão desse trabalho.

RESUMO

Para aperfeiçoar a aprendizagem de alunos nas disciplinas que envolvem compiladores foi criado o Compilador Educativo Verto, uma ferramenta criada para ser usada como auxilio pedagógico mostrando as etapas de compilação passo a passo. Atualmente o Verto conta com estruturas de dados de seleção (se), laços condicionais (enquanto), laços condicionas (para), controles de seleção múltipla (caso). E com os tipos de dados inteiro, lógico e caractere. A proposta desse trabalho é a inclusão de novas estrutura de dados, vetores e constantes, saltos incondicionais e melhorias no código macro-assembler. Desde sua ultima versão o aplicativo sofreu uma serie de atualizações, mesmo assim ainda há a necessidade de ser implementadas tais estruturas visando tornar a ferramenta mais próxima de uma linguagem de programação imperativa. Tais avanços contribuem para o auxílio do ensino na disciplina de compiladores.

Palavras chaves: Compiladores. Arquitetura Cesar. Apoio ao ensino de compiladores.

ABSTRACT

To improve student learning in subjects that involve compilers was created Educational Verto Compiler, a tool designed to be used as a teaching aid showing the steps the build step by step. Currently Verto has data structures of selection (if), conditional loops (while), conditional loops (for), controls multiple selection (case). And with the data types, logical and character. The purpose of this study is the inclusion of new data structures, vectors and constant unconditional jumps and improvements in code macro-assembler. Since its last version, the application has undergone a series of updates, yet there is still a need for such structures to be implemented in order to make the tool closer to an imperative programming language. These advances contribute to the aid of education in the discipline of compilers.

Key words: Compilers. Architecture Cesar. Support for teaching compilers.

LISTA DE FIGURAS

Figura 1: Esquema de um compilador (Muchnick, 1997) 14
Figura 2: Interação do analisador léxico com o *parser*. 17
Figura 3: O fluxo da construção de programas em Verto para CESAR 25
Figura 4: Ambiente de programação e execução para a máquina CESAR. 26
Figura 5: Resultado da compilação 27
Figura 6: Saída do léxico 28
Figura 7: Saída do Sintático 28
Figura 8: Saída do Semântico 28
Figura 9: Saída da pilha semântica 29
Figura 10: Tabela de símbolos gerada. 29
Figura 11: editor do código fonte. 30
Figura 12: código macro-assembler. 31
Figura 13: Tela de ajuda 32
Figura 14: Exemplo de definição dos *tokens* 39
Figura 15: Método ***buscaProximoToken*** 41
Figura 16: Erro Sintático 42
Figura 17: Método comandoVaPara 43
Figura 18: Método ***acaoSemantica*** 44
Figura 19: Método ***montaArquivo***, da classe ***VertoAsm*** 45
Figura 20: Menu de contexto em funcionamento 48
Figura 21: Menu no editor do código *MacroAssembler* 49
Figura 22: Abertura de um arquivo do tipo vrt. 50
Figura 23: Pesquisa na ajuda. 51
Figura 24: Opção novo. 52
Figura 25: Número de linhas 54

Figura 26: Status da compilação 55
Figura 27: Ordenação de vetor 55
Figura 28: exemplo da estrutura atual de pacotes 57
Figura 29: Erro sintático causado pela não atribuição de uma constante 60
Figura 30: Erro de atribuição de constante 60
Figura 31: Erro sintático causado pela falta de um rotulo 62
Figura 32: Erro sintático causado por atribuição 63
Figura 33: Erro semântico causado por tipos diferentes 64
Figura 34: Código macro-assembler com vetor 65

LISTA DE ABREVIATURAS E SIGLAS

ASRD	Análise Sintática Recursiva Descendente
BNF	*Backus-Naur Form*
API	*Application Programming Interface*

SUMÁRIO

INTRODUÇÃO **11**

1 COMPILADORES **14**
1.1 Estrutura de um compilador 15
1.2 Gramáticas 15
1.3 Análise Léxica 16
1.4 Análise Sintática 17
1.5 Análise Semântica 19
1.6 Geração de Código Intermediário 19
1.6.1 Código de três endereços 20
1.6.2 P-código 21
1.7 Geração de Código 21
1.8 Otimização de Código 21
1.9 Tratamento de Erros 22

2 O COMPILADOR VERTO **24**
2.1 Compilador Verto 24
2.2 A Máquina CESAR 26
2.3 A Interface do Verto 26
2.3.1 Swing 32
2.4 A Estrutura Sintática 33
2.4.1 Declaração de variáveis 33
2.4.2 Atribuição 33
2.4.3 Seleção 34
2.4.4 Laço condicional 34
2.4.5 Laço *For* (para) 34
2.4.6 Comando Repita Ate 35
2.4.7 Comando Repita Enquanto 35
2.4.8 Apagar tela 35
2.4.9 Escrever 36
2.4.10 Leitura 36
2.4.11 Declaração de função 36
2.4.12 Conversão de tipos 36

3 ESTRUTURA DO COMPILADOR VERTO **38**
3.1 A gramática 38
3.2 A Classe Léxico 39
3.3 A Classe Sintático 41

3.4 A Classe Semantico 43
3.4.1 A classe NodoPilhaSemantica 44
3.4.2 A classe NodoEstrutura 44
3.5 A Classe VertoAsm 45
3.6 Classe Verto 46
3.7 Padrões de Projeto 46
4 NOVAS FUNCIONALIDADES 47
4.1 Menu De Contexto 47
4.2 Copiar e Colar 49
4.3 Filtro. 49
4.4 Ajuda 50
4.5 Acessibilidade 51
4.5.1 Inicialização 52
4.5.2 Botão novo 52
4.5.3 Editor 53
4.5.4 Número de Linhas 53
4.5.5 Menu Aparência 54
4.5.6 Mensagem de aviso 54
4.5.7 Exemplo Vetor 55
4.5.8 Tabela de símbolos 56
4.6 Código fonte 56
4.7 Tratamento de erros 57
5 NOVAS ESTRUTURAS DO VERTO 59
5.1 Constantes 59
5.1.1 Implementação no Verto 59
5.2 Salto incondicional 61
5.2.1 Rótulo 61
5.2.2 Macro-assembler 62
5.3 Vetor. 62
5.3.1 Declaração de vetores 63
5.3.2 Atribuição de vetores 63
5.3.3 Manipulação de Vetores 64
5.3.4 Vetor no Macro-assembler 65
CONSIDERAÇÕES FINAIS 66
REFERÊNCIAS BIBLIOGRÁFICAS 68
APENDICE 70

INTRODUÇÃO

Para aperfeiçoar a aprendizagem de alunos nas disciplinas que envolvem compiladores foi criado O Compilador Educativo Verto, uma ferramenta criada para ser usada como auxilio pedagógico (Schneider; Passerino; Oliveira, 2005).

O aplicativo sofreu uma serie de atualizações desde sua origem, mesmo assim foram identificadas certas deficiências, que ao serem corrigidas iram ajudar no ensino dessa disciplina.

A linguagem Verto consiste em um conjunto de rotinas bastante reduzido de instruções, com adaptações nas funções de entrada e saída devido ao fato da máquina Cesar possuir um visor limitado (Oliveira, 2005).

Foi desenvolvido usando a linguagem de programação Java e é um software com licença livre GPL (GNU Public License). Assim, os estudantes dispõem de uma ferramenta de projeto aberto e documentada.

Para melhor entender todos os passos da compilação de um código, utiliza-se um editor de texto, o aluno introduz um código fonte em português estruturado que será compilado para um código macro-assembler intermediário e também o código objeto da máquina hipotética Cesar, cada etapa do processo é mostrada separadamente em abas no Verto.

A máquina hipotética Cesar foi criada pelo professor da UFRGS Raul Fernando, com fins didáticos, é um simulador de um processador simples, que contem um contador de programa, que é o endereço da próxima instrução, um registrador de instrução, a instrução executada no momento e um acumulador, uma memória temporária.

Atualmente o Verto suporta as seguintes estruturas de dados: Estruturas de seleção (se), laços condicionais (enquanto), laços condicionas (para), controles de seleção múltipla (caso).

Os tipos de dados suportados são: caractere, inteiro e lógico.

Conforme (Glaser, 2008) algumas das melhorias planejadas para aversão 2.0 não puderam ser concretizadas devido ao tempo e complexidade envolvida. Entre estas melhorias, a mais significativa é a implementação de estruturas de mapeamento finito ou seja, arrays na linguagem. As dificuldades de implementação são provenientes da forma de armazenamento e indexação de cada elemento. Outro aspecto que necessita ser melhorado é a questão do macro-assembler. Por questões de tempo, não se conseguiu até a versão atual que as rotinas da biblioteca tenham seu código fonte ligado ao código macro-assembler, sendo que esta ligação só acontece quando da geração do código de máquina. Esta lacuna prejudica a visão do aluno quanto ao escopo total do código macro-assembler gerado, prejudicando, de certa forma, a percepção da abrangência e volume do código.

As melhorias propostas nesse trabalho são:

(1) Vetores

Conhecido também como array é uma estrutura de dados com tamanho pré-definido, que armazena uma seqüência consecutiva de objetos na memória. Seus dados podem ser acessados informando sua posição, que é um índice normalmente utilizando valores inteiros.

(2) Constantes

Uma estrutura de dado usada para guardar um objeto, sendo que o mesmo não poderá ser modificado, geralmente utilizado para passagem de parâmetros em um método no código.

(3) Saltos incondicionais

É um desvio na execução sem condição para saltar de uma região de código para outra com o uso de marcadores e/ou número de linha muito usada em laços aninhados nas primeiras linguagens de programação estruturada.

(4) Melhoria no código macro-assembler

Incorporação das rotinas da biblioteca no código macro-assembler gerado, visando explanar o processo de ligação de rotinas de biblioteca para o aluno.

Visando a melhor compreensão sobre o funcionamento de compiladores é importante que todos ou a maior parte dos aspectos estejam presentes, a inclusão dos itens apresentados ajudaram a ampliar o conhecimento geral da área de compiladores.

1 COMPILADORES

Segundo Muchnick (1997), Um compilador, com a definição estrita, é constituído por uma série de fases que analisar seqüencialmente dada forma de um programa e sintetizar novos, começando com a seqüência de caracteres que constituem um programa fonte para ser compilado e produzindo, em última instância, na maioria dos casos, um módulo de objeto realocado que pode ser ligado a outros e carregados na memória de uma máquina para ser executado. Como qualquer texto básico sobre construção de compiladores nos diz, há pelo menos quatro fases do processo de compilação, como mostrado na figura 1.

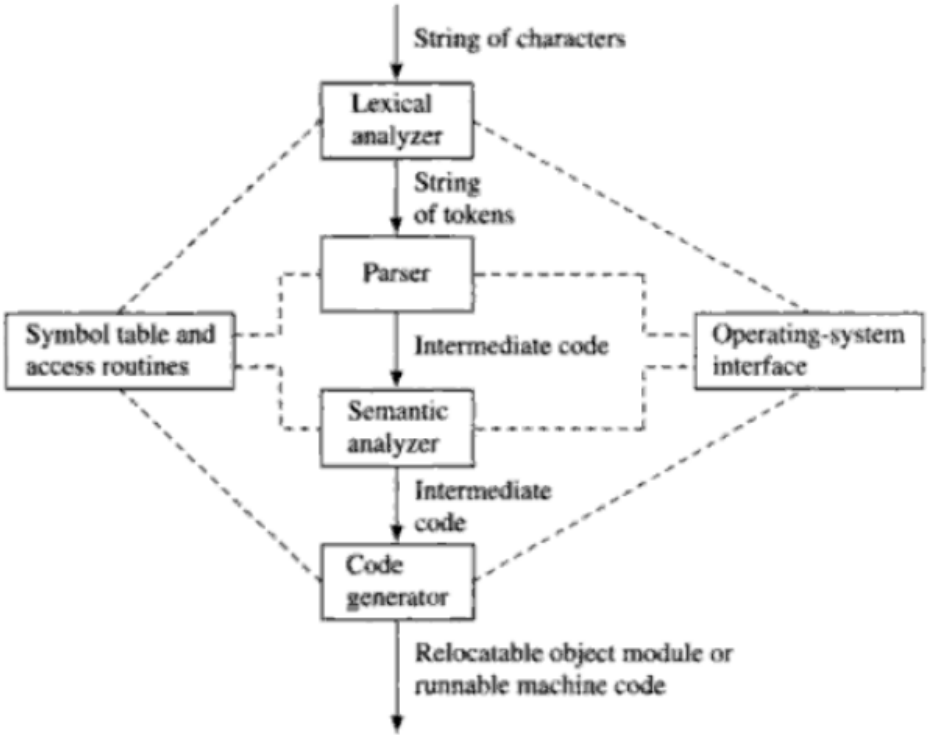

Figura 1: Esquema de um compilador (Muchnick, 1997)

1.1 Estrutura de um compilador

Os compiladores em geral, recebem um código fonte em texto em uma linguagem de alto nível, geralmente com alto nível de abstração a traduzindo geralmente para um código objeto, em uma linguagem de baixo nível, como uma seqüência de instruções a serem processadas por um sistema operacional.(Glaser, 2008)

1.2 Gramáticas

RANGEL (1999) afirma que quase universalmente, a sintaxe das linguagens de programação é descrita por gramáticas livres de contexto, em uma notação chamada BNF (Forma de Backus-Naur ou ainda Forma Normal de Backus), ou em alguma variante ou extensão dessa notação.

As gramáticas são um mecanismo potente e, em simultâneo, legível de representação de linguagens de programação (CRESPO, 1998). Mais especificamente, as gramáticas são mecanismos úteis para descrição das fases de análise léxica e sintática. Conforme proposto por John Backus e Noam Chomsky, as linguagens classificam-se em:

- linguagens enumeráveis recursivamente, ou de tipo 0;
- linguagens sensíveis ao contexto, ou de tipo 1;
- linguagens livres de contexto, ou de tipo 2; e
- linguagens regulares, ou de tipo 3.

Segundo Crespo (1998), uma gramática BNF é formada por um conjunto finito de regras visando definir uma linguagem formal. Uma gramática pode ser vista como um mecanismo para gerar sentenças ou elementos de uma linguagem. Segundo Lewis (2000) uma gramática é uma quádrupla consistindo de quatro componentes:

- um conjunto de símbolos terminais;
- um conjunto de símbolos não terminais;
- um conjunto de regras de produção representando um mapeamento finito entre os não terminais e uma seqüência de símbolos terminais e/ou não terminais; e
- um símbolo inicial pertencendo ao conjunto de não terminais.

A título de exemplo, está ilustrado um trecho da gramática atual do compilador Verto no formato BNF:

```
<programa>              :-  <declaracoes_globais>  <prototipos>
<funcoes>

<declaracoes_globais>   :-  <declaracao_global>                ;
<declaracoes_globais>                  |  <declaracao_global> ;

<prototipos>        :-  <prototipo> <prototipos>

                        |  <prototipo>

<funcoes>           :-  <funcao> <funcoes>
```

1.3 Análise Léxica

Segundo Muchnick (1997), A analise léxica analisa uma seqüência de caracteres e a divide em *tokens* que são membros permitidos na gramática da linguagem em que o programa esta sendo escrito e pode produzir mensagens de erros se a seqüência de caracteres não estiver presente nos *tokens*.

Segundo Segundo Aho; Sethi; Ullman (1995), os analisadores léxicos são divididos em duas fases em cascata , a primeira chamada de “varredura”(*scanning*) e a segunda de “análise léxica”. O scanner é responsável por realizar tarefas simples, enquanto o analisador léxico propriamente dito realiza as tarefas mais complexas.Um exemplo é mostrado na figura 2:

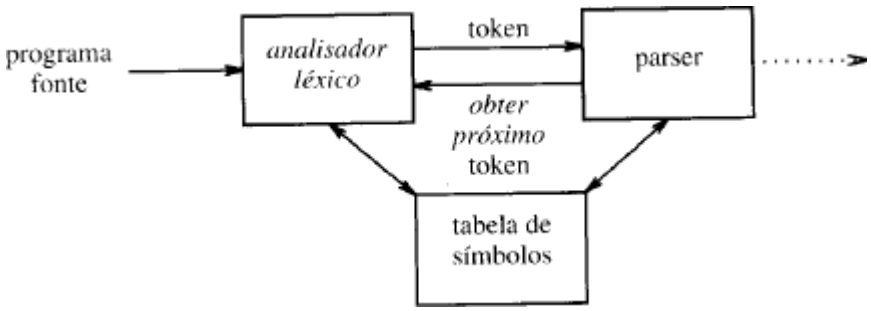

Figura 2: Interação do analisador léxico com o *parser*.

Segundo Muchnick (1997), a combinação de duas técnicas de implementação da analise léxica é o mais indicado:

(1) Um analisador léxico é dito para operar diretamente se, dada uma seqüência de caracteres , o analisador ira determinar o *token* imediatamente a direita do local especificado e mover o ponteiro para a direita da porção do texto formando o *token*.

(2) Um analisador léxico é dito que operam indiretamente se, dada uma seqüência de caracteres, um ponteiro para o texto, e um tipo de *token*, ele irá determinar se a seqüência que aparece imediatamente a direita do ponteiro é do tipo determinado. Se for, o ponteiro é movido para a direita da seqüência.

1.4 Análise Sintática

Segundo Muchnick (1997), analise sintática ou analise, é o que processa a seqüência de *tokens* e produz uma representação de nível intermediário, como uma árvore de análise ou de um código seqüencial intermediário, e uma tabela de símbolos que registra os identificadores usados no programa e seus atributos (e pode gerar erro mensagens se o *token* contém erros de sintaxe)

Segundo Aho; Sethi; Ullman (1995), cada linguagem de programação possui as regras que descrevem a estrutura sintática dos programas bem-formados. Em Pascal, por exemplo, um programa é constituído por blocos, um bloco por comandos, um comando por

expressões, uma expressão por *tokens* e assim por diante. A sintaxe das construções de uma linguagem de programação pode ser descrita pelas gramáticas livres de contexto ou pela notação BNF(Forma de *Backus-Naur*).

Segundo Segundo Aho; Sethi; Ullman (1995), existem três tipos gerais de analisadores sintáticos. Tais como o algoritmo de *Cocke-Younger-Kasami* e o de *Earley*, podem tratar qualquer gramática. Esses métodos , entretanto, são muito ineficientes para se usar num compilador de produção.

Os métodos mais comumente usados nos compiladores são classificados como *top-down* ou *buttom-up*. Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem arvores do topo (raiz) para o fundo (folhas), enquanto que os *bottom-up* começam pelas folhas e trabalham árvore acima até a raiz. Em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

O Compilador Verto utiliza a análise sintática recursiva descendente (ASRD) por motivos de simplicidade e facilidade de aprendizado e compreensão (Oliveira, 2005).

Nesta técnica, é elaborada uma rotina para cada símbolo não terminal da gramática. A medida que os *tokens* correspondentes são encontrados pelo analisador léxico, o analisador percorre uma das regras da gramática e invoca o procedimento correspondente ao não terminal esperado. Cada *token* na entrada pode fazer com que um ASRD tome uma das seguintes ações (WATT, 1999):

- Consumir um *token* após reconhecê-lo como entrada esperada;
- Invocar a rotina correspondente ao não-terminal esperado pela regra sintática;
- Invocar uma rotina de tratamento de erros.

Segundo Watt (1999), cabe ao analisador sintático recursivo descendente decidir que regra de produção deve aplicar em um dado momento da análise de uma sentença.

Programas podem conter erros em muitos níveis diferentes. Por exemplo, os erros podem ser:

- léxicos, tais como errar a grafia de um identificador, palavra-chave ou operador

- sintáticos. tais como uma expressão aritmética com parênteses não-balanceados
- semânticos, tais como um operador aplicado a um operador incompatível
- lógico, tais como uma chamada infinitamente recursiva

Freqüentemente, boa parte da detecção e recuperação de erros num compilador gira em torno da fase de análise sintática.

1.5 Análise Semântica

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

Segundo Delamaro (2004), cabe ao analisador semântico verificar erros semânticos nas construções sintáticas encontradas. Os tipos de inconsistências peculiares que o analisador semântico identifica são:

- Variáveis redeclaradas ou não declaradas;
- Tipos incompatíveis em expressões e atribuições;
- Procedimentos redeclarados ou não declarados;

Uma das técnicas usadas para a análise semântica é denominada Tradução Dirigida pela Sintaxe. Para por a termo o processo de análise semântica e posteriormente a geração de código, define-se uma gramática de atributos que registra as ações semânticas a serem verificadas cada vez que uma regra é reconhecida (KAKDE, 2003).

1.6 Geração de Código Intermediário

Segundo Louden (2004), em razão da complexidade da geração de código, um compilador normalmente quebra essa fase em várias partes, utilizando diversas estruturas de dados intermediárias, que normalmente requerem alguma forma de código abstrato. Um

compilador pode também chegar a não gerar código executável, e em vez disso, gerar um código de montagem, que necessite da ajuda de um montador, um vinculador e um carregador, os quais pode ser fornecidos pelo sistema operacional, ou juntamente com o compilador. O código intermediário é particularmente útil quando existe a necessidade de se produzir código altamente eficiente, que requer uma quantidade significativa de análise das propriedades do código alvo, o que é facilitado pelo código intermediário. Em particular, as estruturas de dados adicionais que incorporem informações de análise detalhada posterior a análise sintática podem ser geradas facilmente a partir do código intermediário, embora isso não possa ser feito a partir da árvore sintática.

Ainda segundo Louden (2004), o código intermediário também pode ser útil para facilitar os redirecionamentos de um compilador para outras linguagens alvo diferentes. Pois gerar código executável para outra linguagem alvo exigirá apenas que o tradutor do código intermediário para o código alvo seja reescrito, o que em geral é mais fácil de reescrever do que todo o gerador de código.

Segundo Louden (2004), duas das mais populares formas de código intermediário são: Código de três endereços e P-Codigo.

1.6.1 Código de três endereços

A instrução mais básica do código de três endereços é projetada para a avaliação das de expressões aritméticas. O código de três endereços requer que o compilador gere nomes para os temporários. Esses temporários correspondem aos nós interiores de uma árvore sintática. E representam seus valores computados, com o temporário final representando o valor da raíz.. A forma como esses temporários são armazenados na memória não é especificado por esse código, e em geral eles são atribuídos a registradores, mas também podem ser mantidos em registros de ativação. (Louden, 2004)

Ainda segundo Louden (2004), para acomodar todas as construções de linguagens de programação padrão, é preciso variar a forma de código de três endereços para cada construção. A principal razão para não existir uma forma padrão de código de três são as

características específicas de algumas linguagens, que faz necessária a criação de de novas formas de código de três para expressar essas características.

1.6.2 P-código

O P-código começou um como um código de montagem-alvo padrão produzido para compiladores Pascal, entre os anos de 1970 e 1980. Ele foi projetado como uma máquina hipotética baseada em pilhas, denominada P-máquina, para a qual foi escrito um interpretador para diversas máquinas reais. O P-código se mostrou muito útil também como código intermediário, e diversas modificações são usadas em diversos compiladores de código nativo, sendo a maioria delas para código pascal. (Louden, 2004)

1.7 Geração de Código

Segundo Rangel (1999), enquanto a fase de análise (léxica, sintática e semântica), é dependente essencialmente da linguagem de programação em questão, a geração de código é fortemente dependente da máquina alvo, a máquina onde o código compilado deverá ser executado. Não necessariamente a compilação se dará na máquina alvo, que por seu tamanho ou características não permitem a execução de um compilador, que é o caso do software embutido, ou seja, onde o software gerado será gravado em uma ROM e incluído no aparelho para funcionar.

Ainda segundo Rangel (1999) a divisão de um compilador em *front-end* (análise) e *back-end* (geração e otimização de código) reflete estas características. È possível gerar um *front-end* para cada linguagem diferente e gerar um *back-end* para cada máquina alvo.

1.8 Otimização de Código

Segundo Rangel (1999), várias tarefas e técnicas podem ser reunidas sob o nome de Otimização. Para ser razoável, é impossível dizer que um programa pode ser otimizado, isto é, recebendo na entrada um programa e devolvendo na saída o mesmo programa sendo este último, o melhor possível. O que na maioria das vezes acontece, é que o otimizador recebe um programa, e na sua saída tem, sob alguns critérios específicos, em sua saída um programa

melhor que o da entrada. Normalmente é muito fácil fazer a transformação de um programa em um programa otimizado. O difícil é obter a informação necessária para garantir que essa otimização pode ser aplicada, sem modificar as funcionalidades do programa original e saber se essa otimização realmente irá ficar mais rápida.. As otimizações que um compilador deve oferecer dependem muito da finalidade do compilador. Por exemplo, caso o compilador deva ser apresentar mensagens de erro bem específicas para o programador, essa compilação não deve ser totalmente otimizada. Já se o programa final for compilado apenas uma vez e executado muitas vezes, o ideal é que se otimize ele o máximo possível, a fim de fazer sua execução a melhor e mais rápida possível.

Segundo Louden (2004), embora a otimização possa melhorar a as características de um programa, ela não resolve todos os problemas, de forma que um programa mal escrito, mesmo passando por uma série de otimizações ele ainda continuará mal escrito, pois a lógica do programa continua a mesma.

1.9 Tratamento de Erros

Segundo Crespo (1998), no contexto dos compiladores, entende-se por tratamento de erros o processo que é desencadeado pelo reconhecimento logo após a detecção de um erro na frase que está sendo analisada. Os tipos de reações tomadas a cada erro podem ser de dois tipos:

- Sinalização do erro: Notificação enviada ao programador que alerta para o fato de ter sido encontrada alguma violação das regras, que invalidam a seqüência a ser processada pelo compilador.
- Correção / Recuperação: Permite superar uma falha detectada e prosseguir a análise, até a aceitação da seqüência ou a detecção de novos erros.

A detecção de erro é uma tarefa inerente à análise. Quando um compilador está analisando uma frase, tentando verificar se ela foi corretamente escrita, os aparecimentos dos erros ocorrem por três razões:

- Erro léxico: O aparecimento e caracteres inválidos que impedem a detecção de qualquer símbolo terminal pertencente à linguagem
- Erro sintático: O aparecimento de um símbolo terminal combinado com um outro símbolo válido, mas combinados de uma forma inválida.
- Erro Semântico: O aparecimento de um símbolo terminal, que apesar de válido, está em uma posição inválida, ou infringe regras de tipo ou concordância, que tem que ser verificadas para identificar completamente o significado da frase.

Ainda, segundo Crespo (1998) a sinalização do erro é de importância crucial para que o programador possa localizar com eficiência, onde o erro aconteceu e porque está acontecendo. Para isso, os requisitos mínimos para o envido da mensagem de erro devem ser:

- A posição: Local onde ocorreu o erro.
- O símbolo do erro: Símbolo encontrado no momento em que ocorreu o erro.
- Causa: A causa provável que gerou o erro. Pode indicar qual era o símbolo esperado, ou a concordância que é pretendida.

2 O COMPILADOR VERTO

Neste capítulo será descrito o funcionamento do Compilador Verto, e o funcionamento da máquina hipotética de *César*.

2.1 Compilador Verto

O Compilador Educativo Verto, disponível para download em http://verto.sf.net, desde o mês de junho de 2005, que já teve um total de 3413 downloads (uma média de cerca de 64 downloads por mês), foi projetado para o auxílio do ensino de compiladores levando o estudante a exercitar e compreender as principais fases de um compilador e sua relação com a linguagem de montagem. (OLIVEIRA, 2005).

O estudo de compiladores, apesar de comumente envolver o projeto e construção de um compilador, tem como meta não obrigatoriamente a construção de um compilador em si, mas compreender e analisar os princípios, ferramentas e técnicas usadas. O planejamento de uma disciplina de compiladores deve considerar que o curso se consolide nestes objetivos e não somente na construção do projeto em si. (OLIVEIRA, 2005).

O foco principal do Compilador Educativo Verto são as fases finais do processo de compilação. Por motivos educacionais, optou-se por uma técnica de análise léxica não automatizada, e o método de análise sintática recursiva descendente (OLIVEIRA, 2005).

Segundo (Schneider et al, 2005) O Compilador Educativo Verto surgiu da necessidade de desenvolver uma ferramenta de apoio pedagógico para a disciplina de Compiladores do Centro Universitário Feevale. A disciplina de Compiladores objetiva levar o aluno a perceber concretamente as diversas etapas envolvidas no processo de compilação, elaborado no final da disciplina o projeto de um compilador funcional para uma linguagem

simples. O Compilador Educativo Verto foi escrito na linguagem *Java* e elaborado conforme licença *GPL (GNU Public License)*.

O processo de compilação do verto se dá em duas etapas distintas. Inicialmente é gerado um código intermediário, em formato *macro-assembler*, formato esse que objetiva facilitar a compreensão das estruturas compiladas, dispondo de instruções com mais simplificadas das instruções da máquina CESAR. Desse modo, promove-se um melhor entendimento de como as estruturas da linguagem Verto, elaboradas pelo aluno, são transformadas para instruções próximas à da máquina objeto. A partir dessa forma intermediária, gera-se o arquivo destino final contendo as instruções no formato da máquina hipotética Cesar, permitindo que o aluno execute e analise o algoritmo (SCHNEIDER et al. 2005).

A figura 3 ilustra o fluxo de construção de um programa para a Máquina CESAR por meio do Compilador Verto.

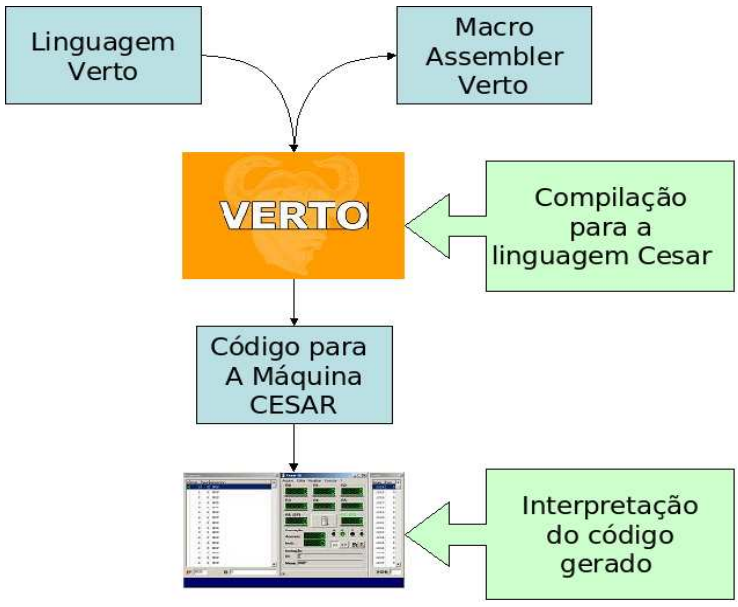

Figura 3: O fluxo da construção de programas em Verto para CESAR

Segundo Schneider et al. (2005), para elaborar um algoritmo, o aluno dispõe de um editor de texto embutido no Verto. Uma vez finalizado a codificação do algoritmo (código-fonte), é possível submetê-lo ao compilador, que gera o código macro-assembler

intermediário e o código-objeto na linguagem da máquina hipotética CESAR. Uma vez realizado isso, o aluno pode consultar a saída de cada uma das fases da compilação.

2.2 A Máquina CESAR

Segundo Weber (2001), o computador hipotético CESAR é uma simplificação de uma das arquiteturas mais populares para processadores de pequeno porte, o PDP-11 (*Programmed Data Processor*). Essa arquitetura serviu como inspiração para projetos de microprocessadores de 4, 8 e 16 bits. Com esse computador hipotético é possível facilitar o processo de ensino-aprendizagem de programação de baixo nível. Mas, mesmo com uma máquina simulada, com um conjunto reduzido de instruções e componentes, ainda assim, programar neste nível não é uma tarefa trivial. A figura 4 ilustra o interpretador construído para o código objeto para CESAR. Para maiores detalhes, aconselha-se a leitura de (WEBER 2001).

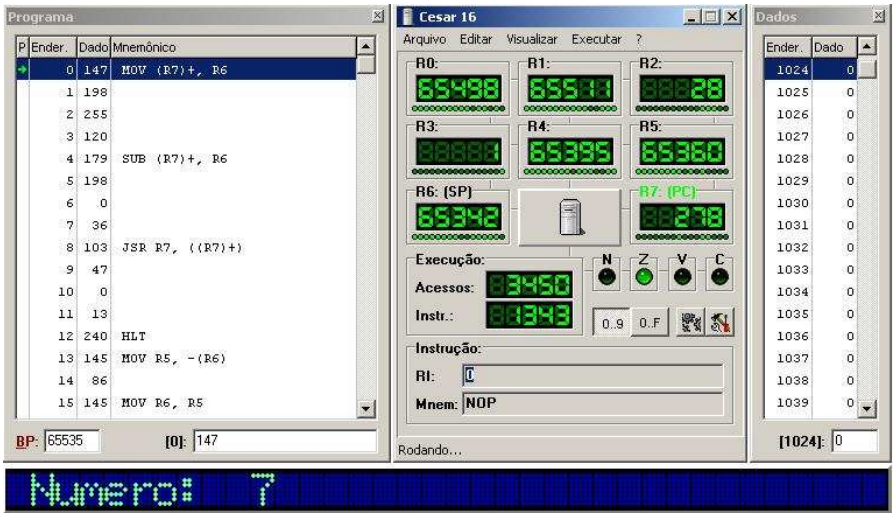

Figura 4: Ambiente de programação e execução para a máquina CESAR.

2.3 A Interface do Verto

Para detalhar o funcionamento do programa, será mostrado abaixo uma serie de telas do Verto efetuando a compilação de um código fonte para o calculo do quadrado de um numero.

.

Na figura 5, o código fonte escrito é compilado sem erros.

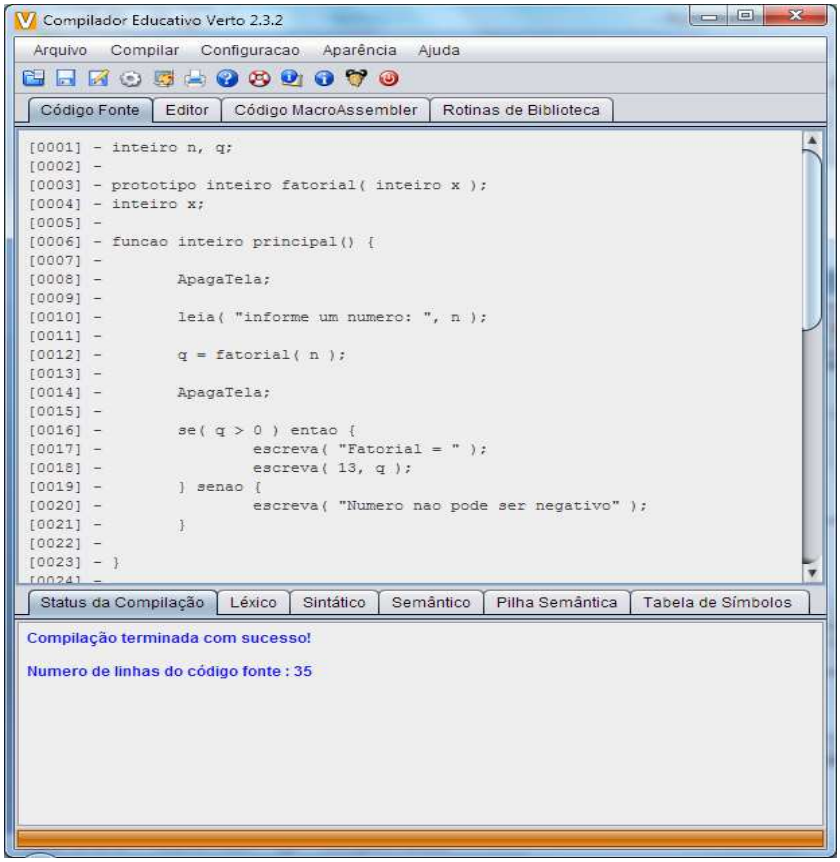

Figura 5: Resultado da compilação

Na figura 6, apresenta a saída do analisador léxico.

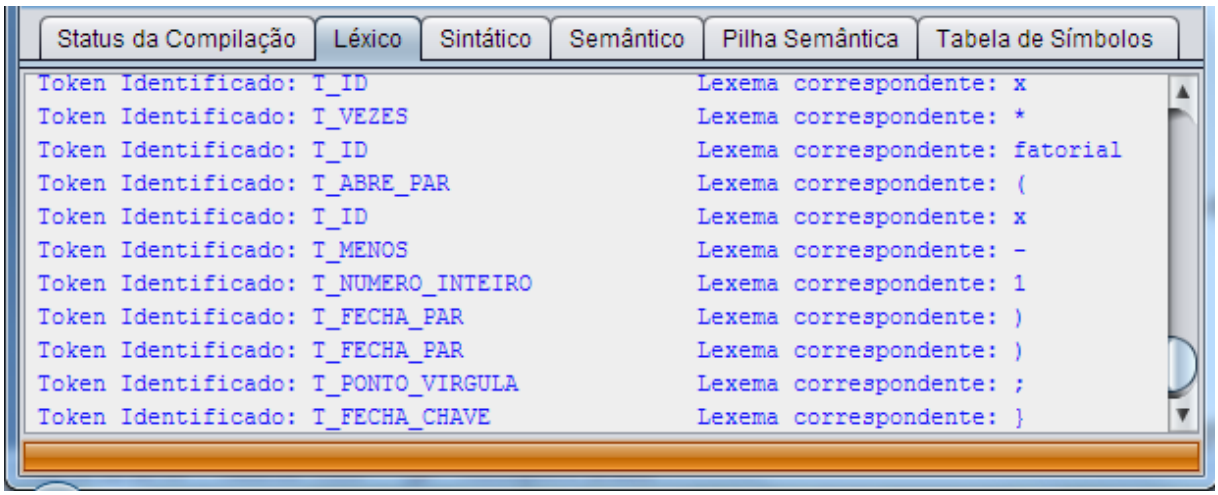

Figura 6: Saída do léxico

Na figura 7, apresenta a saída do analisador sintático.

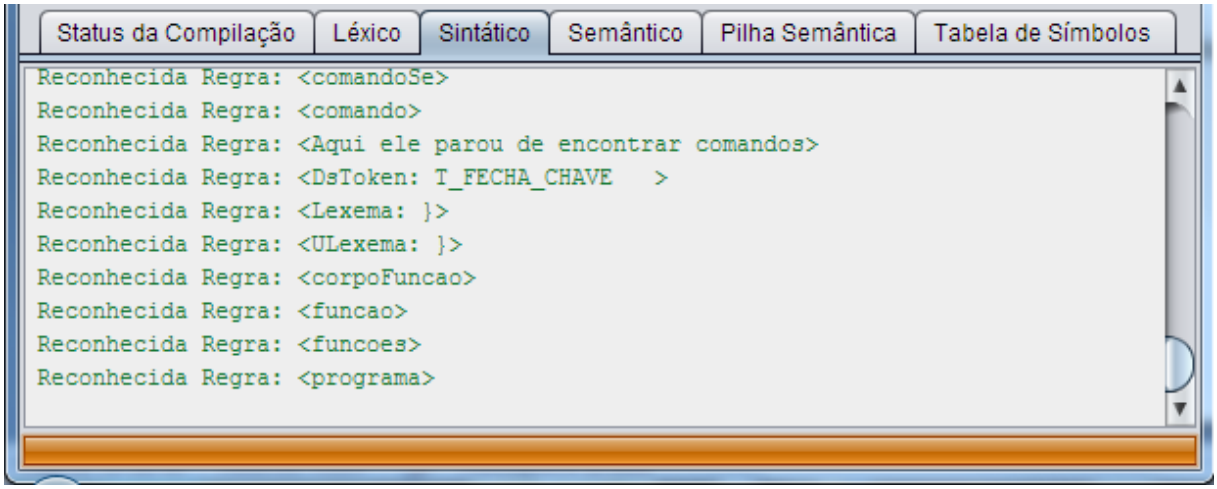

Figura 7: Saída do Sintático

Na figura 8, demonstra a estrutura de saída do analisador semântico.

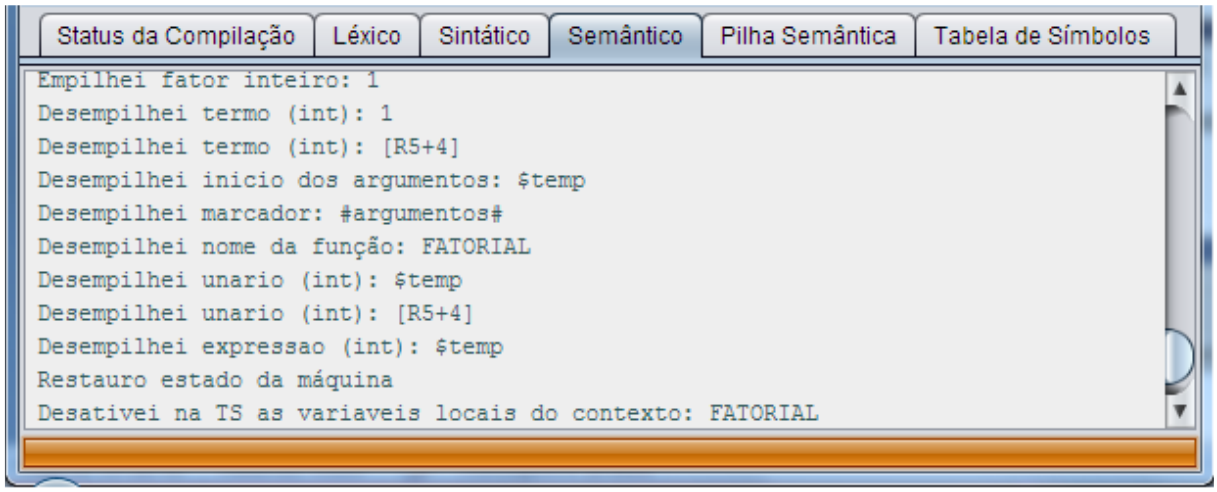

Figura 8: Saída do Semântico

Na figura 9, apresenta a saída da Pilha Semântica.

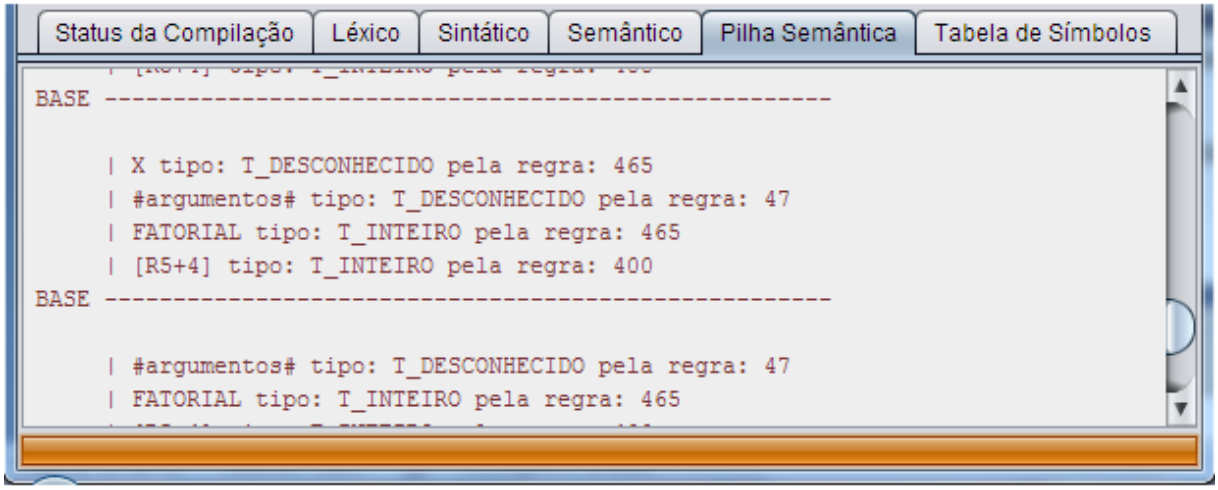

Figura 9: Saída da pilha semântica

Na figura 10, a tabela de símbolos gerada.

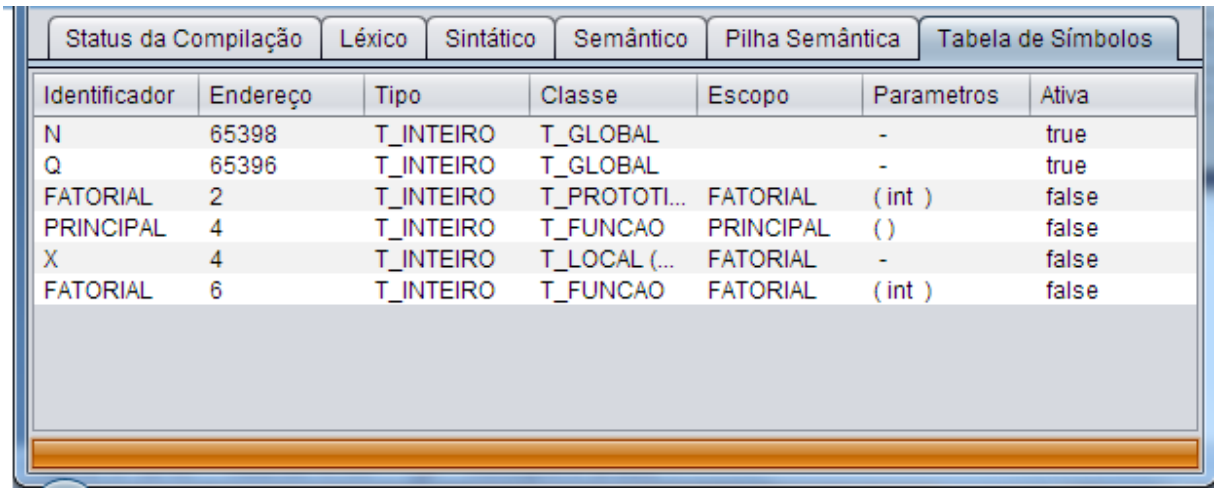

Figura 10: Tabela de símbolos gerada.

Na figura 11, mostra o editor.

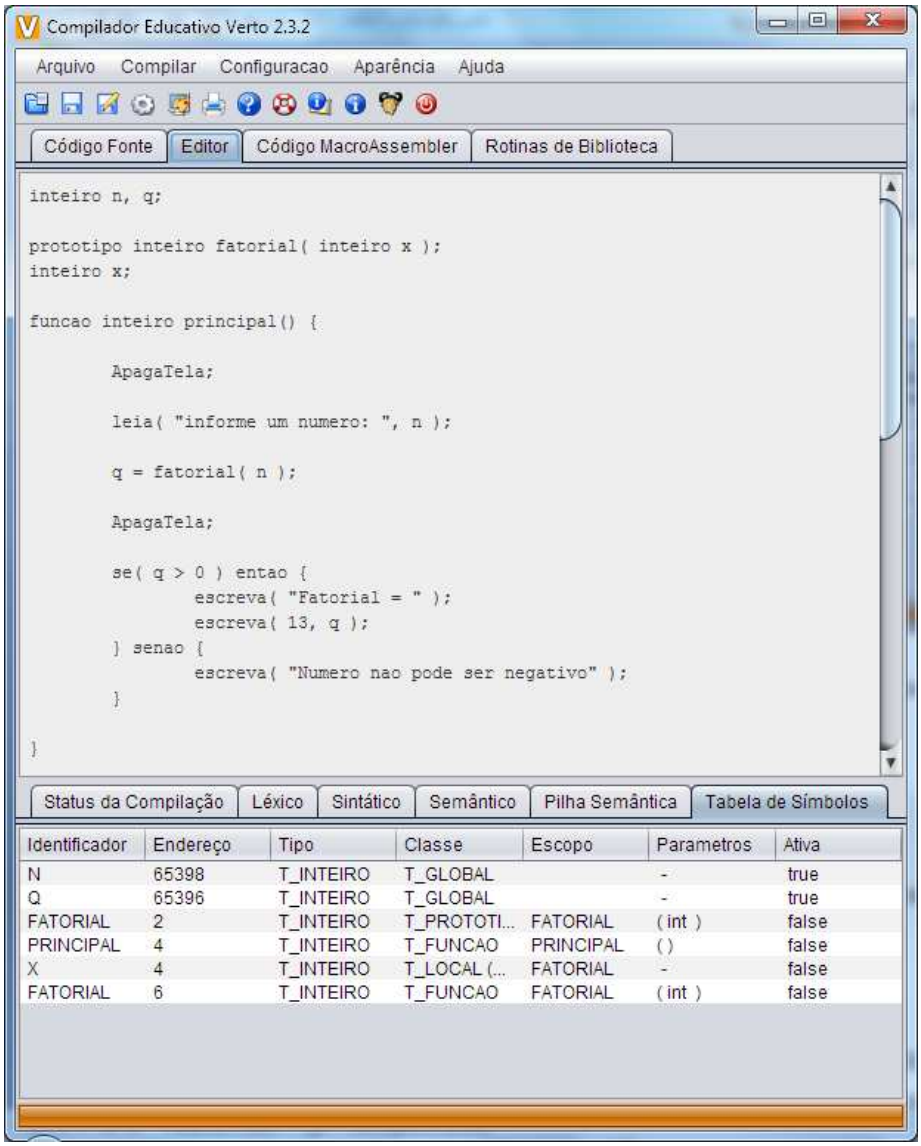

Figura 11: editor do código fonte.

Na figura 12, mostra o macro-assembler.

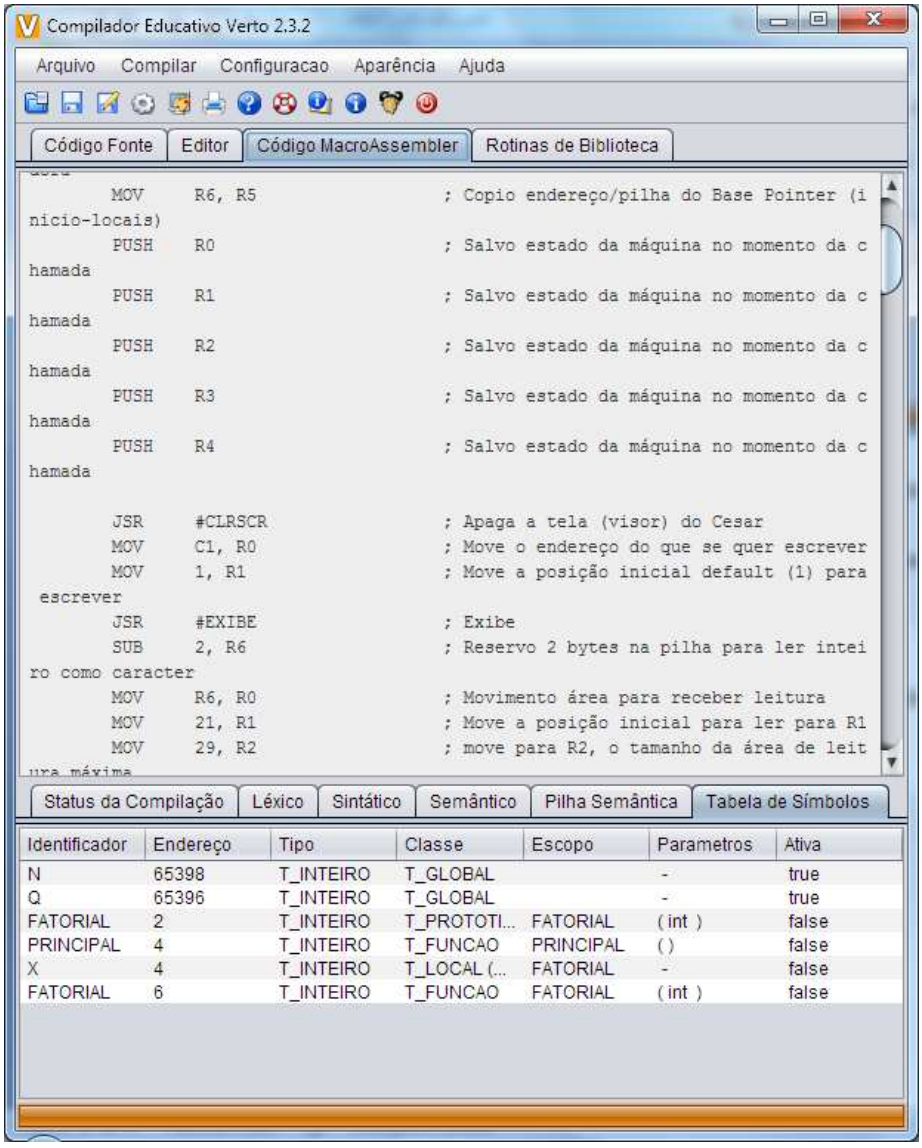

Figura 12: código macro-assembler.

Tela de ajuda do Verto, na figura 13.

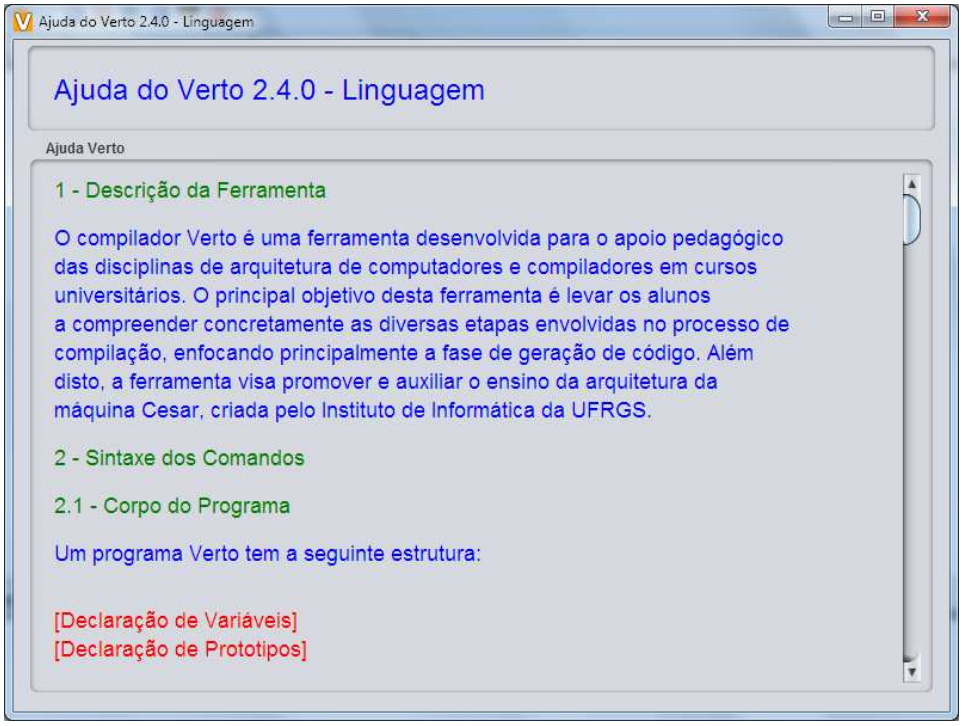

Figura 13: Tela de ajuda

2.3.1 Swing

Segundo Serson (2007) no momento em que um desenvolvedor Java constrói uma aplicação *desktop*, provavelmente a primeira opção que ele tem para construir as telas de sua aplicação é a API(*Application Programming Interface*), ou Interface de Programação de Aplicativos padrão que já vem com o JDK(*Java Development Kit*), ou conjunto de desenvolvimento Java. Essa API se chama *Swing*.

Segundo a Sun, o projeto *Swing* tem como principal objetivo construir um conjunto extensível de componente para GUIs (*Graphical user interface*), ou interface de usuário gráfica, para possibilitar que os desenvolvedores escrevam mais rapidamente a parte gráfica(*Front End*) das aplicações.

Para isso, o projeto estabeleceu algumas diretivas para orientar a criação da API Swing:

1 – totalmente implementado em Java;

2 – prover uma única API capaz de suportar múltiplas aparências e comportamentos(*multiple look-and-feels*), para não restringir os desenvolvedores e nem mesmo os usuários finais;

3 – promover o poder de uma arquitetura voltada ao uso de camadas modelos, mas sem exigir uma API de altíssimo nível;

2.4 A Estrutura Sintática

O Compilador Verto apresenta estruturas sintáticas básicas. A seguir, serão demonstradas estas estruturas sintáticas como propriamente escritas no programa e definir a funcionalidade de cada uma delas.

2.4.1 Declaração de variáveis

Variável de acordo com Horstmann é um item de informação na memória cujo posição é identificada por um nome simbólico. Sendo assim, toda variável é definida com um tipo e os tipos de dados suportados pelo compilador são: Inteiro (numérico), Caractere (alfanumérico) e Lógico(Booleano).

A forma de declaração de cada uma delas é:

```
inteiro a, b;

caracter c;

lógico d, e;
```

2.4.2 Atribuição

Na atribuição a informação é guardada na variável definida, caso o tipo do dado não corresponde com o tipo da variável declarada um erro semântico é notificado ao usuário.

A sintaxe de atribuição em uma variável declarada que será utilizada posteriormente no Verto é expressa da seguinte forma:

```
<variável> = <expressão>;
```

2.4.3 Seleção

Utilizado para definir que fluxo o programa ira tomar, a partir do momento que uma condição for verdadeira o trecho de código escrito no bloco entre a chave aberta após a palavra reservada então será executado, caso seja uma condição falsa, o bloco executado será o escrito após a palavra reservada senão.

A sintaxe da seleção no Verto é expressa da seguinte forma:

```
se ( <condição> ) então {

} senão {

}
```

2.4.4 Laço condicional

No laço condicional, o trecho de código escrito no bloco definido pela palavra reservada enquanto, irá ser executado até que a condição seja falsa. O bloco pode não ser executado já que o teste é efetuado antes do inicio da execução.

A sintaxe do laço condicional pré-teste é expressa da seguinte forma:

```
enquanto ( <condicao> ) {
}
```

2.4.5 Laço *For* (para)

No laço do tipo para, uma variável de controle que é incrementada a cada iteração e é utilizada como controle desse laço que é definido utilizando uma expressão na mesma estrutura.

O laço do tipo para,é amplamente utilizado nas linguagens de programação atuais pode ser utilizado por exemplo para percorrer um vetor, posição por posição utilizando a variável incrementada como identificador do endereço desse vetor

A sintaxe do laço do tipo *for* é expressa da seguinte forma:

```
para <variável> = <expressão> ate <expressão> {
        }
```

2.4.6 Comando Repita Ate

O trecho de código definido é executado até que a condição verdadeira seja alcançada, esse tipo de laço é comumente utilizado quando se quer que pelo menos o bloco seja executado uma vez.

A sintaxe do repita ate é expressa da seguinte forma:

```
repita {
<comandos>
        }
ate (<condição>)
```

2.4.7 Comando Repita Enquanto

Esse laço é parecido com o repita ate, só que uma condição falsa é utilizada para interromper as iterações efetuadas pelo laço.

A sintaxe do repita enquanto é expressa da seguinte forma:

```
repita {
<comandos>
      }
enquanto (<condição>)
```

2.4.8 Apagar tela

Quando utilizada no programa, essa rotina apaga caractere por caractere o visor na máquina hipotética Cesar.

```
ApagaTela;
```

2.4.9 Escrever

A Rotina utilizada para escrever um texto no viso da máquina hipotética César está ilustrada abaixo. A posição é necessária pois o visor da máquina César permite apenas uma linha.(Glaser, 2008)

```
escreva ( <posição>, <lista_expressoes> )

ou

escreva( <lista_expressoes> )
```

2.4.10 Leitura

Utilizada para ler um caractere digitado na maquina César, e colocar o seu valor em uma variável. O não terminal <posição> indica a posição onde se iniciará a leitura da variável. .(Glaser, 2008)

```
leia (<posição>, <variavel> );
```

2.4.11 Declaração de função

A declaração de função é expressa da seguinte forma

```
função <tipo> <nome_funcao> ( <parâmetros> ) {
}
```

2.4.12 Conversão de tipos

Utilizada para converter um caracter em inteiro no Compilador Verto

```
paraInteiro ( <expressao_caracter> )
}
```

Utilizada para converter um inteiro para caracter no Compilador Verto

```
paraCaracter ( <expressao_caracter> )
}
```

3 ESTRUTURA DO COMPILADOR VERTO

Dada a gramática que foi feita na versão 1.0.3 do Compilador Educacional Verto, criou-se a estrutura necessária para que a compilação ocorra de forma satisfatória. Desde sua primeira versão, já ocorreram atualizações para o suporte a novos elementos, para que o compilador se torne mais completo e cumpra sua função pedagógica de forma plena.

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

Neste capitulo será mostrado como todo o processo de compilação descrito anteriormente ocorre no compilador Verto. E também como foi estruturado a codificação necessária em código Java.

3.1 A gramática

A gramática do verto é de certa forma simples. As regras tem um número correspondente, que é utilizador posteriormente pelo analisador semântico cada vez que uma regra é reconhecida.

Por convenção, os símbolos não-terminais desta gramática estão definidos entre os caracteres ‘<’ e ‘>’´, sendo assim, qualquer símbolo que não estejam entre estes sinais, representam símbolos terminais e devem ser assumidos como *tokens.* (Glaser, 2008)

Toda a informação da gramática fica registrada em um arquivo de texto nomeado de “Gramatica.txt” localizado na raiz do executável do Compilador Educacional Verto. Sua utilização é meramente ilustrativa, já que o mesmo não é utilizado efetivamente pelo Verto. Já que suas regras foram implementadas diretamente no código-fonte.

3.2 A Classe Léxico

Seguindo o desenvolvimento da versão 1.0 do Compilador Educacional Verto, onde o analisador léxico foi desenvolvido de forma manual (sem utilização de programas como *Lex* e *Yacc*), em uma classe Java chamada de ***Léxico***. Esta classe define uma lista de *tokens* e métodos voltados para percorrer o código fonte, indicando os *lexemas* e classificando os mesmos como *tokens*. (Glaser, 2008)

Para cada *token* existente na gramática, uma constante inteira na classe ***Lexico*** é criada. Na figura 14 é possível visualizar um exemplo de como isso ocorre.

```
public static final int T_FALSO          = 0;
public static final int T_VERDADEIRO     = 1;
public static final int T_ESCREVA        = 2;
public static final int T_SE             = 3;
public static final int T_ENTAO          = 4;
public static final int T_SENAO          = 5;
public static final int T_ENQUANTO       = 6;
public static final int T_FUNCAO         = 7;
public static final int T_OU             = 8;
public static final int T_E              = 9;
public static final int T_NEGUE          = 10;
public static final int T_INTEIRO        = 11;
public static final int T_CARACTER       = 12;
public static final int T_LOGICO         = 13;
public static final int T_VAZIO          = 14;
public static final int T_LEIA           = 15;
public static final int T_APAGATELA      = 16;
public static final int T_RETORNE        = 17;
public static final int T_PROTOTIPO      = 18;
public static final int T_PARAINTEIRO    = 19;
public static final int T_PARA           = 20;
public static final int T_REPITA         = 21;
public static final int T_ATE            = 22;
public static final int T_PARACARACTER   = 23;
```

Figura 14: Exemplo de definição dos *tokens*

Se durante a execução do analisador léxico ocorra algo inesperado, é levantada uma exceção, que é tratada de forma adequada pela classe ***ErroLexicoException***

Na classe **Lexico,** quando instanciada, no seu construtor que é um método utilizado pelo Java para quando um novo objeto é criado, é passado como parâmetro todo o código fonte editado no pelo Compilador Educativo Verto.

A classe contem um método chamado de ***buscaProximoToken,*** que fica responsável por eliminar os caracteres em branco e marcadores no código fonte sendo compilado pelo Compilador Educativo Verto, o método então a partir de um laço varre todas as palavras do código caractere por caractere, através do método ***moveLookahead*** para encontrar um lexema, que seja reconhecido como um *token* existente na gramática.

Se o lexema não forma um *token* esperado pelo sistema, uma exceção do tipo ***ErroLexicoException*** é apresentada para o usuário, como uma mensagem alertando sobre o problema encontrado.

Na figura a seguir, é mostrado um trecho de código do método ***buscaProximoToken,*** e também pode-se observar a utilização do método ***moveLookahead*** chamado dentro de um laço do tipo *while*, no Compilador Educacional Verto a estrutura é semelhante e foi chamado do tipo enquanto.

```
private boolean buscaProximoToken( boolean exibe ) {
    ultimoToken  = token;
    ultimoLexema = lexema;
    lexema = "";
    StringBuffer esteToken = new StringBuffer();
    while( brancos.toString().indexOf( lookahead ) >= 0 ) {
        moveLookahead();
    }
    if( isAlpha( lookahead ) ) {
        while( isAlpha( lookahead ) ||
               isDigit( lookahead ) ||
               isPonteiro( lookahead ) ) {
                esteToken.append( lookahead );
                moveLookahead();
        }
        lexema = esteToken.toString().toUpperCase();
        if( lexema.equals( "LEIA" ) ) {
            token = T_LEIA;
        } else if( lexema.equals( "ESCREVA" ) ) {
            token = T_ESCREVA;
        } else if( lexema.equals( "IMPRIMA" ) ) {
            token = T_ESCREVA;
        } else if( lexema.equals( "SE" ) ) {
            token = T_SE;
        } else if( lexema.equals( "ENTAO" ) ) {
            token = T_ENTAO;
        } else if( lexema.equals( "SENAO" ) ) {
            token = T_SENAO;
```

Figura 15: Método ***buscaProximoToken***

3.3 A Classe Sintático

O Compilador Educacional Verto, usa a Análise Sintática Recursiva Descendente, e ainda utiliza essa técnica na sua versão atual, apenas incorporando novas funcionalidades nessa nova versão do Compilador Educacional Verto.

Para cada *token* reconhecido pelo analisador léxico, o mesmo é identificado pelo analisador sintático, dentro de uma seqüência de *tokens* coerentes com a gramática. São

atribuídos métodos para consumir os *tokens* encontrados. A cada *token* consumido, busca-se o próximo. (Glaser, 2008)

Se o *token* que esta sendo analisado pelo método correspondente não esta presente na definição da gramática, é então levantada uma exceção do tipo ***ErroSintaticoException,*** que mostra ao aluno uma mensagem de erro, exibindo que na linha que ocorreu o erro se espera um lexema diferente daquele inserido no código fonte. A figura 16 mostra como isso ocorre.

Figura 16: Erro Sintático

Caso não ocorra um erro sintático, cada vez que um *token* é consumido, é executado um método presente na classe ***Semantico*** chamado de ***acaoSemantica,*** passando o identificador da regra associada a gramática que é exclusivo. Assim garantido que a regra semântica seja executada de forma correta. Na figura 17 é mostrado como é feita a implementação da analise sintática do comando *VaPara().*

```
/**
 * Metodo que implementa a regra [160]:
 * <cmd_va_para> :- va para <rotulo>
 */
private void comandoVaPara() {
    if ( lexico.getToken() == Lexico.T_VA ) {
        lexico.proximoToken();
        if ( lexico.getToken() == Lexico.T_PARA ) {
            lexico.proximoToken();
            if ( lexico.getToken() == Lexico.T_ROTULO ) {
                lexico.proximoToken();
                semantico.acaoSemantica( 160 );
                reconheceRegra( "comandoVaPara" );
            } else {
                throw new ErroSintaticoException( "Eu esperava encontrar aqui um 'r[otulo', mas encontrei: " +
                                                  lexico.getLexema() + " na linha: " +
                                                  lexico.getLinha() );
            }
        } else {
            throw new ErroSintaticoException( "Eu esperava encontrar aqui um 'para', mas encontrei: " +
                                              lexico.getLexema() + " na linha: " +
                                              lexico.getLinha() );
        }
    } else {
        throw new ErroSintaticoException( "Eu esperava encontrar aqui um 'va', mas encontrei: " +
                                          lexico.getLexema() + " na linha: " +
                                          lexico.getLinha() );
    }

}
```

Figura 17: Método comandoVaPara

3.4 A Classe Semantico

A análise semântica utiliza uma gramática de atributos para estabelecimento das regras semânticas. A partir do reconhecimento dos *tokens* essas regras são invocadas pelo analisador sintático.(Glaser, 2008)

Cada vez que uma regra é invocada, o código correspondente em linguagem Macro Assembler é concatenado, ate todos os tokens serem identificados, se tudo ocorrer sem problemas é gerado um arquivo que posteriormente gerar o código objeto para a máquina hipotética César.

O método ***acaoSemantica*** que recebe como parâmetro uma variável inteira que no caso, é uma regra atribuída pelo analisador sintático**,** e de acordo com ela chama um método correspondente para a concatenação do código Macro Assembler, que se encontra na classe ***Semantico,*** pode ser visualizado na figura 18.

```
public void acaoSemantica( int regra ) {

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

    switch ( regra ) {
    case 0      : criaAsm(); break;
    case 2      : somaTamanhosGlobais(); break;
    case 3      : invocaProcedimentoPrincipal(); break;
    case 20     : inicioAreaFuncoes(); break;
    case 25     : finalizaDeclaracaoFuncao(); break;
    case 32     : somaTamanhosLocais(); break;
    case 33     : armazenaEstadoMemoria(); break;
    case 40     : trataDeclaracaoGlobal(); break;
    case 45     : trataDeclaracaoLocal(); break;
    case 47     : trataDeclaracaoIdFuncao(); break;
    case 48     : trataDeclaracaoIdPrototipo(); break;
    case 51     : alimentaTSComParametros(); break;
    case 53     : alimentaTSPorPrototipo(); break;
    case 64     : acumulaTamanhoVariaveis(); break;
    case 65     : declaraIdentificador(); break;
    case 75     : declaraParametro(); break;
    case 100    : empilhaPosicaoLeitura(); break;
    case 101    : implementaLeitura(); break;
```

Figura 18: Método ***acaoSemantica***

3.4.1 A classe NodoPilhaSemantica

Esta classe foi implementada com o propósito de fazer o controle da pilha semântica. O objetivo dela é empilhar as formas intermediárias de código Macro Assembler para posterior concatenação no código objeto. (Glaser, 2008)

Sua estrutura consiste em montar um objeto que será utilizado pela classe ***PilhaSemantica,*** com o código definido na gramática, o tipo do dado que será manipulado e qual regra que foi chamada

3.4.2 A classe NodoEstrutura

Esta classe foi implementada com o propósito de ir armazenando os rótulos gerados para serem aplicados no código objeto. É usado amplamente em laços de repetição como, por exemplo, o RepitaAte. (Glaser, 2008)

Sua estrutura consiste basicamente em armazenar dois rótulos, um utilizado para definir o inicio do laço, e o outro para marcar o fim do laço, e também definir que tipo de estrutura esta sendo manipulada.

3.5 A Classe VertoAsm

A classe ***VertoAsm*** é a classe responsável pela geração do código-objeto do compilador educacional Verto. É nesta classe que é feita a leitura do código *macro-assembler* gerado pela classe ***Semantico*** e gerado o arquivo *.mem* que pode ser aberto e executado diretamente na máquina hipotética *Cesar*. (Glaser, 2008)

No construtor dessa classe é passado dois parâmetros, no primeiro o caminho que será salvo o arquivo *.asm* e no segundo o caminho do arquivo *.men*

Na figura a seguir, vemos um pedaço da classe ***VertoAsm***. O método ***montaArquivo***.

```
private void montaArquivo( String nmArquivoAsm, String nmArquivoSaida ) throws ErroGeracaoCodigoException {

    File f = new File( nmArquivoAsm );
    StringBuffer fonteAsm = null;

    if( !f.exists() ) {
        throw new ErroGeracaoCodigoException( "Arquivo fonte inexistente (" + nmArquivoAsm + ")." );
    }

    try {
        InputStreamReader isr = new InputStreamReader( new FileInputStream( f ), "LATIN1" );

        try {

            fonteAsm = new StringBuffer( (int) f.length() );
            int ch;

            while( ( ch = isr.read() ) != -1 ) {
                fonteAsm.append( (char) ch );
            }
        } finally {
            isr.close();
        }

    } catch( Exception e ) {
        e.printStackTrace();
        throw new ErroGeracaoCodigoException( "Erro ao carregar arquivo asm: " + e.getMessage() );
    }

    monta( fonteAsm );
    gravaArquivoMem( nmArquivoSaida );
}
```

Figura 19: Método ***montaArquivo***, da classe ***VertoAsm***.

3.6 Classe Verto

Classe principal do programa, onde esta localizado o método *main*, que no Java é o primeiro método executado. Nessa classe toda a estrutura visual e funcional do Compilador Educacional Verto esta implementada.

Toda a informação salva, como diretórios de compilação, cores do texto e das janelas desenhadas pelo *API Swing*, fica registrada em um arquivo de texto nomeado de "parametrosVerto.txt".

Localizado na raiz do executável do Compilador Educacional Verto, que é, um arquivo de extensão *.jar*, comumente utilizado em aplicações Java onde armazena todo o código fonte compilado.

Ele é então carregado na inicialização do aplicativo, para que possa ser utilizado como referencia das configurações definidas pelo usuário.

3.7 Padrões de Projeto

Toda a estrutura de projeto do Verto, se baseia no *Abstract Factor*, que como diz (SHALLOWAY, 2004) é utilizado nas ocasiões em que você deve coordenar a criação de famílias de objetos. Isso abre espaço para remover as regras relacionadas ao modo como efetuar a instanciação para fora do objeto cliente que está usando esses objetos criados.

Primeiro, identificar as regras para instanciação e definir uma classe abstrata como interface que tenha um método para cada objeto que necessita ser instanciado. Então, implementar classes concretas, a partir dessa classe, para cada família. O objeto cliente utiliza esse objeto fábrica para criar os objetos servidores de que ele necessita.

4 NOVAS FUNCIONALIDADES

.No decorrer das implementações do Compilador Educacional Verto, nessa nova versão, foi identificado uma série de novas necessidades que não estavam previstas e também alguns *bugs*(erros causados pela implementação).

Para melhorar a interação do aluno com o programa, algumas delas visam o aperfeiçoamento da experiência como um todo, para facilitar o uso do aplicativo, que vai desde a sua *interface* até conceitos de acessibilidade, então algumas delas foram implementadas e serão mostradas nesse capitulo.

4.1 Menu De Contexto

Foi criado um menu de contexto que é acionado com o botão direito do *mouse* no editor de código, nele foi inserido a opção de ”Auxilio Sintaxe”, e as opções de “Copiar” e “Colar” que serão mostradas na sessão a seguir, para facilitar o uso do sistema. No momento que a opção desejada é acionada, o texto correspondente ao comando é inserido no editor onde o cursor estava localizado no momento do clique.

Na figura 20 mostra-se o menu de contexto no editor do código Verto.

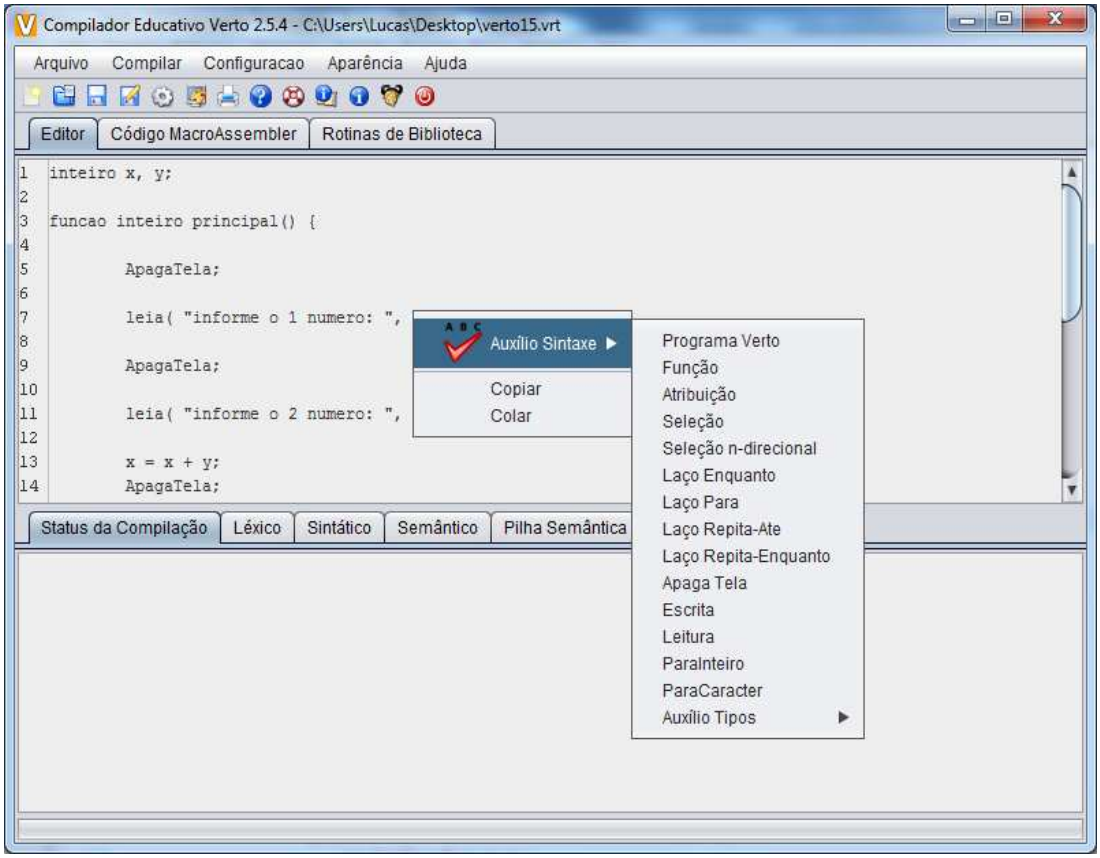

Figura 20: Menu de contexto em funcionamento

O menu de contexto também pode ser acionado no editor do código macroAssembler, como esse editor tem a função de apenas exibir o código *macro-assembler* gerado pelo processo de compilação não se tem a necessidade de manter o ”Auxilio Sintaxe”, apenas as opções de “Copiar” e “Colar” torna-se útil para eventuais edições desse código gerado ou visualização em outro editor de código fonte.

Quando o menu é acionando novamente no Editor a opção de ”Auxilio Sintaxe” retorna como uma opção no menu. Como pode ser visto na figura 21 o menu de contexto só apresenta o “Copiar” e “Colar”.

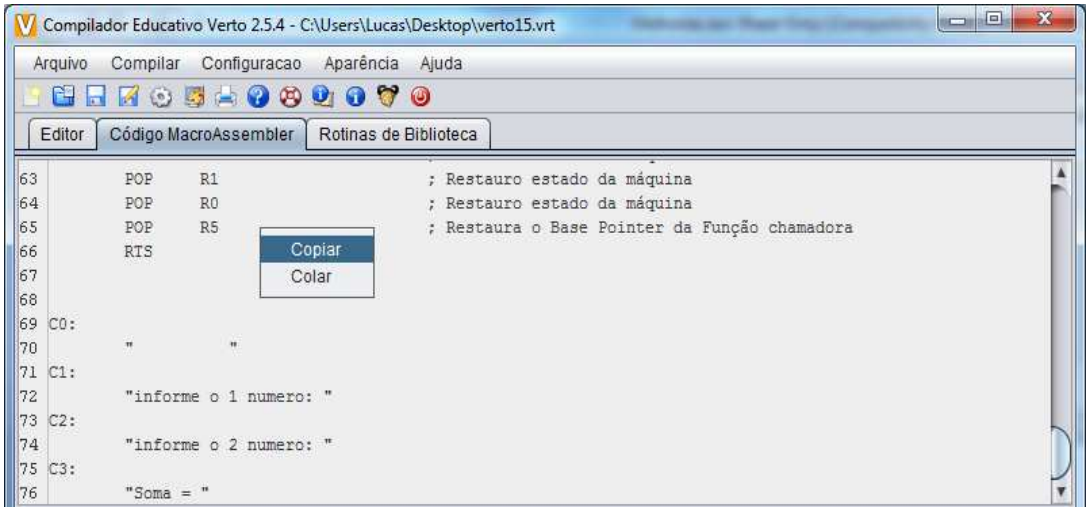

Figura 21: Menu no editor do código *MacroAssembler*

4.2 Copiar e Colar

No menu de contexto do editor de código, foi adicionada as funcionalidades de “Copiar” e “Colar”. Que ao selecionar um texto no editor, acionar o menu e escolher a opção “Copiar” a sequência de caracteres é copiada para o *clipboard* (área de transferência de dados entre aplicativos do sistema operacional) do sistema operacional.

No momento que a opção de “Colar” é acionada todo o conteúdo que se encontra no *clipboard* é inserido no editor na posição do cursor. Exemplos das opções mostrado nas figuras 20 e 21.

4.3 Filtro

Quando fosse solicitada a abertura de um arquivo, o programa responsável pelo gerenciamento de arquivos comumente utilizado em programas feitos em Java, chamado de *JFileChooser* exibia todos os arquivos existentes no diretório padrão, sendo que a única extensão necessária é a “vrt” que é utilizada pelo Compilador Educacional Verto. Na figura 22 é mostrado um exemplo de procura de arquivo e o filtro sendo utilizado.

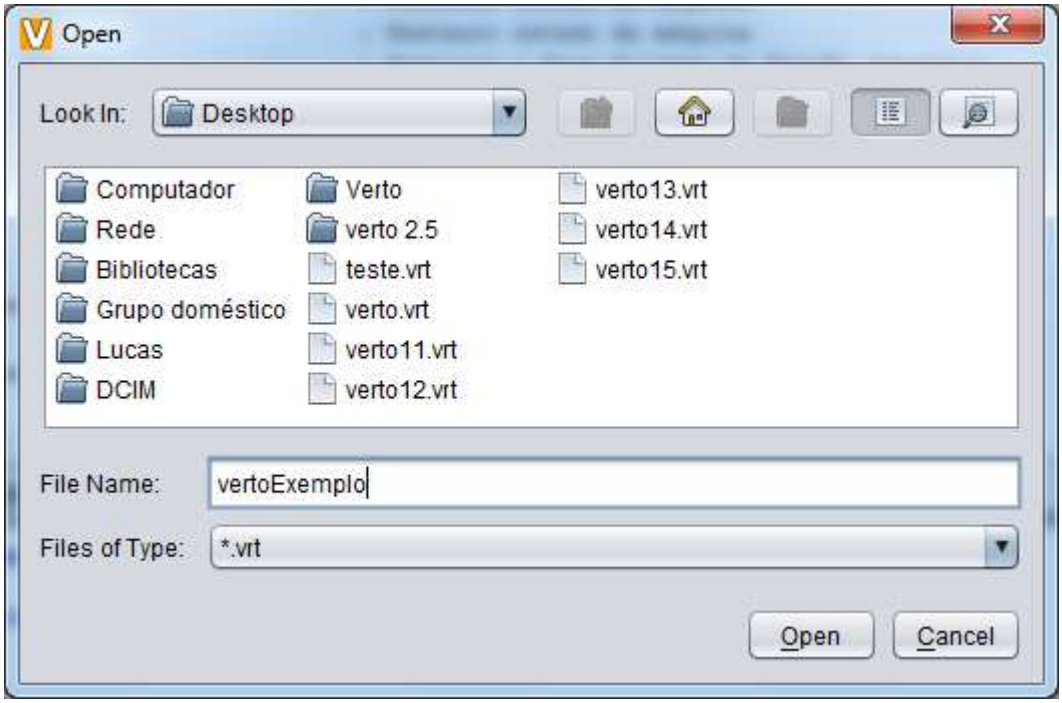

Figura 22: Abertura de um arquivo do tipo vrt.

4.4 Ajuda

Com as novas funcionalidades novos tópicos foram adicionados para exemplificar o uso de vetores, constantes e saltos incondicionais. Algumas melhorias também foram feitas como a mudança de *layout* (arranjo dos componentes dentro de uma tela no sistema) para possibilitar uma melhor visualização do conteúdo ali presente.

Na versão atual, a visualização da ajuda tem o suporte de pesquisa por palavras, que ao pressionar a tecla f, abre-se uma pequena tela para se informar a palavra ou frase desejada para efetuar a pesquisa, destacando no texto quando o conteúdo da pesquisa for encontrado.

Como o texto da ajuda ficou muito extenso, esse componente facilita o uso quando o aluno sentir a necessidade de obter alguma informação referente ao conteúdo de compiladores e de alguma funcionalidade do Compilador Educacional Verto.

A figura 23 mostra a tela para inserir o texto para a pesquisa, e o resultado da mesma.

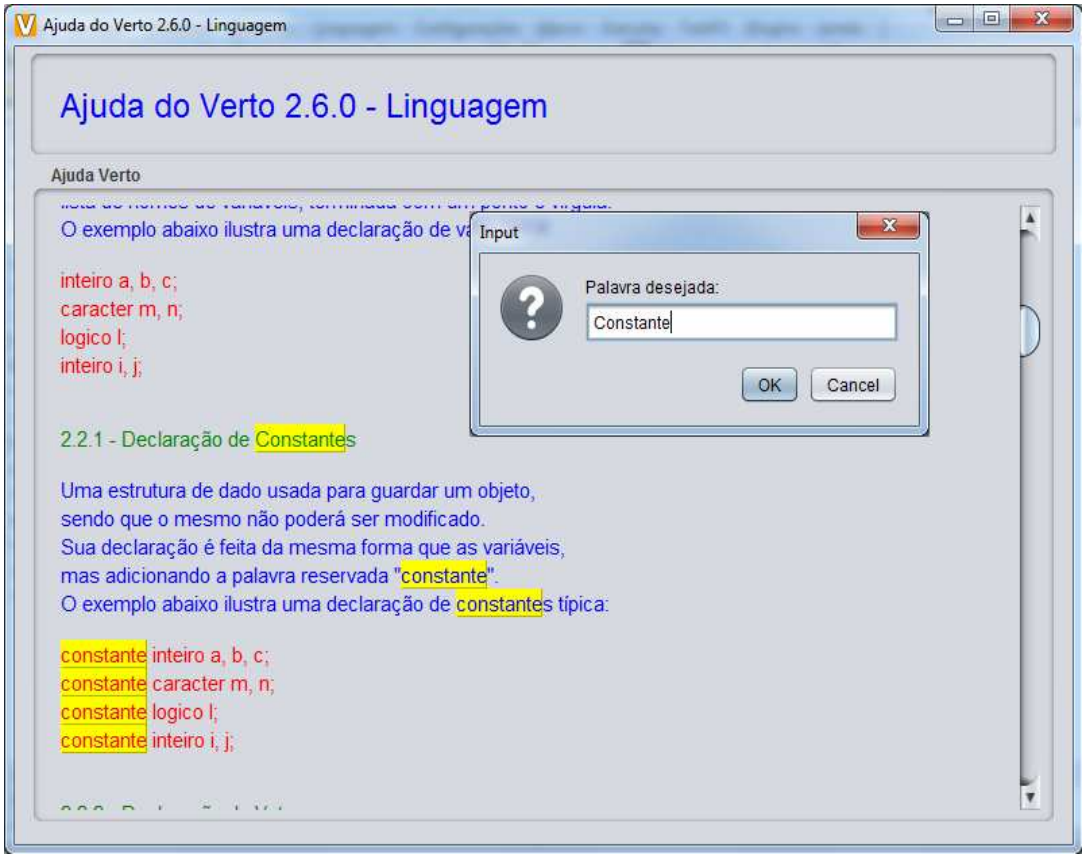

Figura 23: Pesquisa na ajuda.

4.5 Acessibilidade

Com o crescente uso dos computadores ou qualquer outro dispositivo que possa rodar uma aplicação Java, por exemplo, que ficam cada vez mais acessíveis a população em geral, cada vez mais os programadores estão pensando em como seus aplicativos devem se comportar aos mais diversos tipos de usuários.

Pra tentar facilitar o uso do Compilador Educacional Verto, algumas medidas foram tomadas, como atalhos, configuração de cores, novos menus, um arquivo de ajuda mais completo e fácil de utilizar, e outras funcionalidade que visam a produtividade, para que não seja frustrante para o aluno, cada vez que tenha que vir utilizar o compilador, assim melhorando a experiência do usuário.

O Compilador ainda não suporta ferramentas para utilização de pessoas com deficiências visuais e/ou motoras, mas em suas futuras versões possa vir ter essas funcionalidades.

4.5.1 Inicialização

Nas versões anteriores do Verto, cada vez que o programa fosse novamente aberto, ele inicializava em branco, ou seja, com nenhum código fonte carregado pelo sistema, na versão atual do Verto o ultimo código fonte editado já é automaticamente exibido pelo sistema.

Na versão atual do verto também foi adicionado a visualização do caminho do arquivo, onde fica salvo as modificações efetuadas no código fonte que também pode ser visto na barra de titulo na figura 24.

4.5.2 Botão novo

Devido a essa funcionalidade se viu necessário a criação de uma nova opção, que foi a adição de um botão no menu que se chama “Novo”, que acionado, limpa todos os editores e consoles do registro atual, e prepara o programa para uma nova edição. Na figura 24 o menu pode ser visualizado.

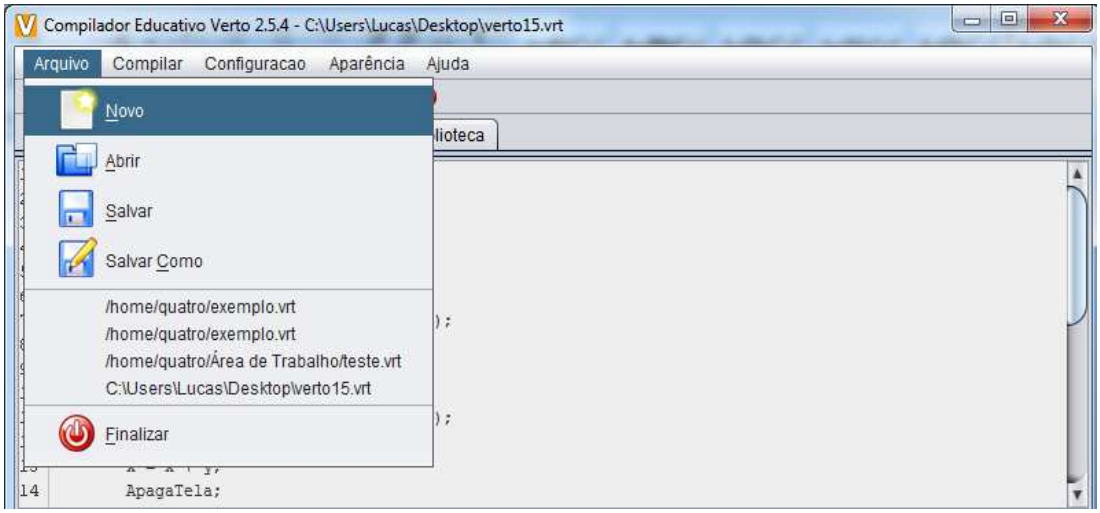

Figura 24: Opção novo.

Outro botão que tem a mesma funcionalidade do “Novo”, também foi adicionado a barra de atalhos presente em baixo da barra de menus. Ele pode ser visualizado na figura 21, sendo o primeiro botão.

4.5.3 Editor

Para melhorar a experiência ao utilizar o Verto, houve uma mudança no editor, anteriormente havia uma aba a mais chamada de “Código Fonte” que pode ser visualizada na imagem na figura 5, que servia basicamente para exibir o número de linhas do código.

Mas seu funcionamento tinha algumas limitações, que poderiam causar algum problema de uso do Verto, quando fosse efetuada alguma ação no “Código Fonte”, o foco era mudado de forma automática para a aba “Editor”.

Esse processo devido às ordens de execução de *threads* efetuados pelo *swing* poderia eventualmente causar algumas ações inesperadas pelo usuário dependendo da forma de uso.

A aba e o editor ” Código Fonte” foi retirado, mantendo somente o “Editor”, para a visualização do código fonte.

4.5.4 Número de Linhas

Devido à retirada da aba “Código fonte”, o editor passou a ter suporte a exibição de números de linhas em tempo real, o número da linha agora é exibido de forma independente do texto, não atrapalhando o uso do editor.

Os números das linhas ficam localizados a esquerda do editor, separando-se do texto por uma linha divisória, se o código fonte passar do número de linhas já apresentado, o mesmo é incrementado, da mesma forma, quando o texto passa a conter menos linhas que já tem no editor, o numero decresce. Como mostra a figura 25.

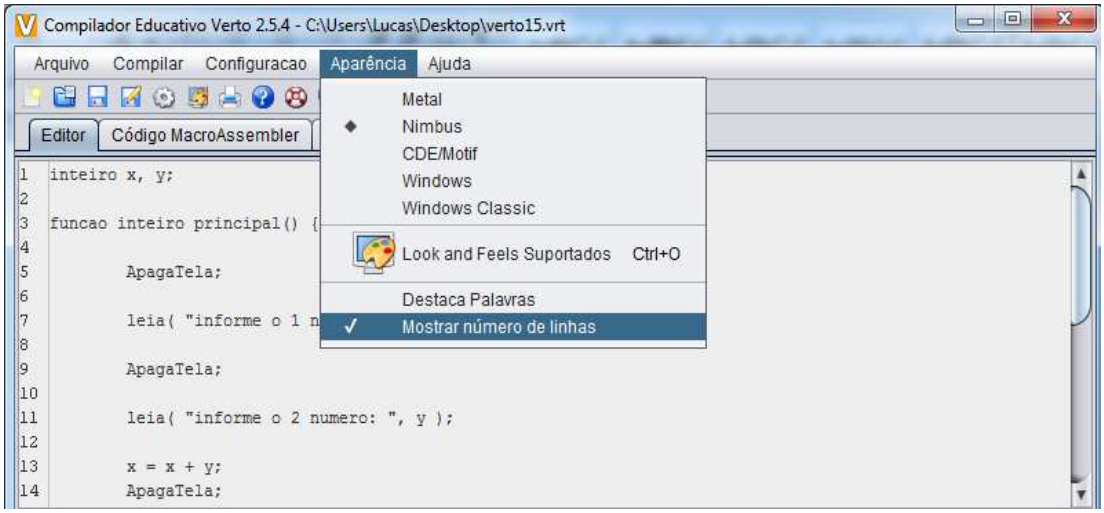

Figura 25: Número de linhas

4.5.5 Menu Aparência

No menu “Aparência”, lista-se os *lookandfeels* também conhecidos como temas, apresentados pelo sistema de forma dinâmica, em versões anteriores isso não ocorria.

Por exemplo, se o Verto for executado em um sistema operacional que não tenha suporte ao tema *Nimbus*, o mesmo não será apresentado pelo menu.

A opção para mostrar o número de linhas também foi adicionada, se o aluno sentir a necessidade de ter uma área de visão maior do editor, pode obter mais espaço horizontal desmarcando a opção.

A figura 25 mostra como o processo ocorre.

4.5.6 Mensagem de aviso

Na versão anterior do Compilador Educacional Verto, quando a compilação fosse efetuada de forma correta, uma tela exibindo uma mensagem de “Compilação efetuada com sucesso” era exibida, e aguardava a confirmação do usuário para que o fluxo prosseguisse.

Para não interromper o fluxo do processo, agora essa tela foi retirada, e apenas o console onde é exibida informações para o aluno, contem a informação que a compilação ocorreu sem problemas.

Uma pequena animação foi feita, para chamar atenção do usuário, que o processo terminou, as letras trocam de cores alternando entro o vermelho e o azul por alguns centésimos de segundos. A figura 26 mostra o processo.

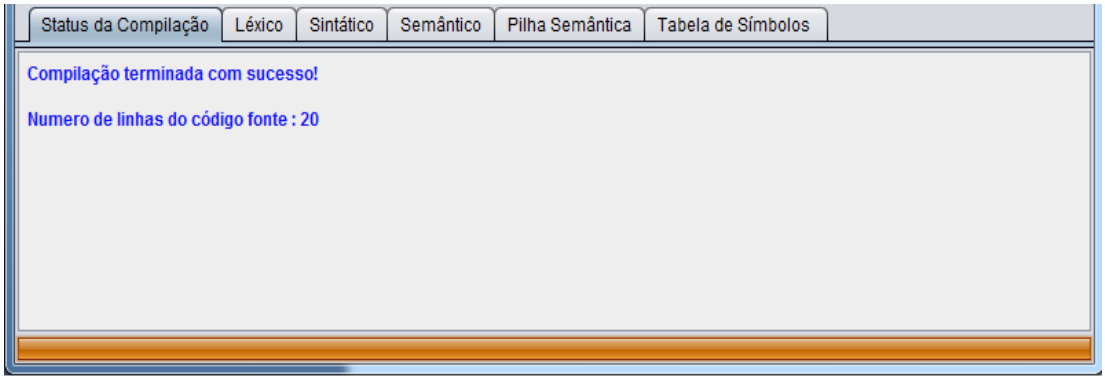

Figura 26: Status da compilação

4.5.7 Exemplo Vetor

No sub-menu “Exemplos de Programas”, existente no menu “Ajuda”, foi adicionado mais uma opção que é a inserção de um programa exemplo de ordenação de vetores, para poder demonstrar como funciona o uso de vetores no Compilador Educacional Verto. Na figura 27 tem a opção selecionada.

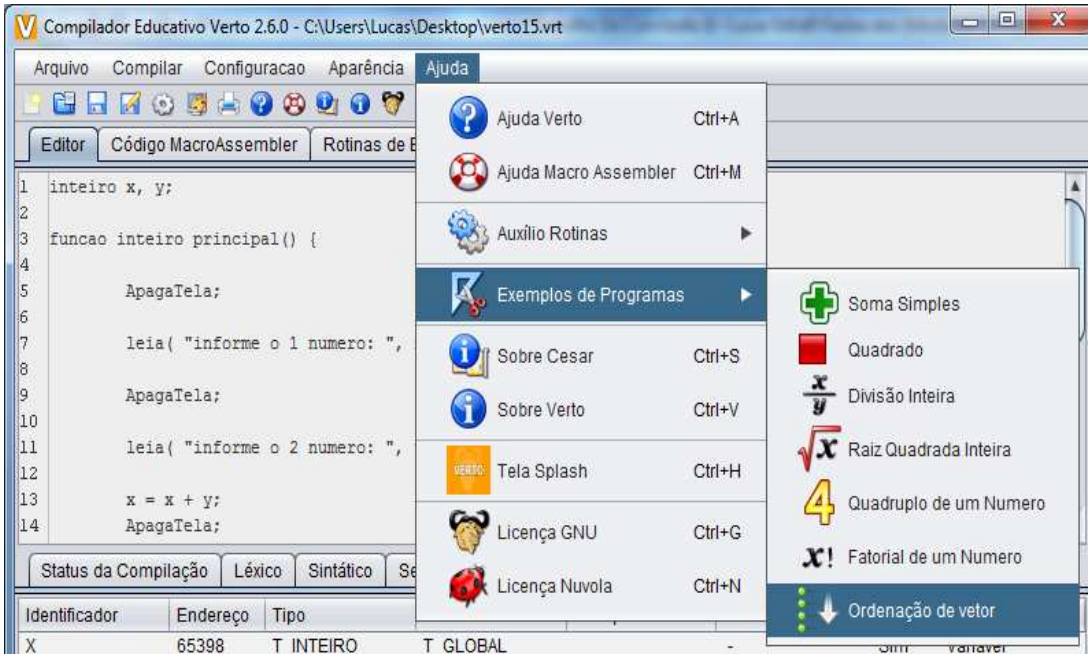

Figura 27: Ordenação de vetor

4.5.8 Tabela de símbolos

Devido às novas estruturas inseridas no Compilador Educacional Verto, a tabela de símbolos recebeu uma nova coluna chamada de “Tipo Identificador”, que exibe o tipo de dado é aquele mostrado na linha correspondente. Que pode ser uma variável, uma constante, um vetor ou uma função

Ao inserir essa nova coluna, foi necessário o redimensionamento das outras colunas já presentes na tabela, para poder ser exibida de forma clara as informações presentes em todas as colunas.

A coluna “Ativa” exibia o seu conteúdo de forma booleana, como *true* e *false*, sendo que deve ser uma mensagem informativa para o aluno, então, as mensagens foram trocadas para “Sim” e “Não” respectivamente.

4.6 Código fonte

Visando facilitar a leitura do código para futuras alterações, sendo que o mesmo é *open source* foi efetuada uma mudança na estrutura de pacotes do código fonte do verto.

Pacotes segundo (Horstmann, 2004) é um conjunto de classes relacionadas, mas a importância da organização dos pacotes se da à medida que os programas ficam maiores, simplesmente distribuir as classes em vários arquivos não é o suficiente. É necessário um mecanismo de estruturação adicional. Em Java, os pacotes fornecem esse mecanismo de estruturação.

.Criando pacotes para assim organizar as classes com suas devidas funções, facilitou a visão geral do programa. Aqui na figura 28 o esquema de pacotes nessa versão do Verto

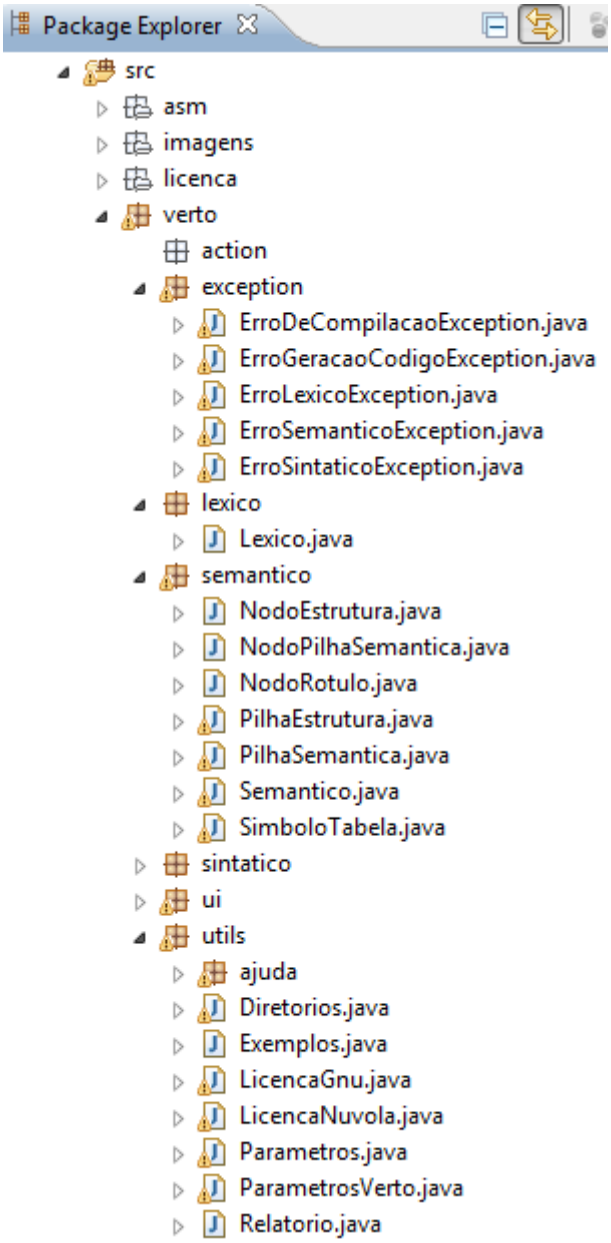

Figura 28: exemplo da estrutura atual de pacotes

4.7 Tratamento de erros

Alguns erros foram encontrados no decorrer do desenvolvimento, alguns deles serão relatados a seguir:

(1) Quando um novo programa fosse escrito e efetuasse a compilação dos fontes sem um arquivo de compilação salvo, levantava uma exceção de *null pointer* que é causado quando o programa tenta acessar um objeto que ainda não foi referenciado

em memória ou que deixou de ser. Utilizando um tratamento de exceção existente no Java foi possível contornar o problema.

(2) No menu “Arquivo”, existe uma lista dos últimos 4 arquivos salvos em disco pelo usuário, quando um quinto arquivo fosse salvo, o mesmo não era mais listado no menu. Com um controle adequado das variáveis de controle o problema foi sanado.

(3) Todos os editores e consoles com exceção do editor do código fonte não são mais editáveis, como anteriormente.

5 NOVAS ESTRUTURAS DO VERTO

Nesse capitulo será demonstrado como foi construída as novas estruturas do Compilador Educacional Verto no código Java propostas no inicio desse trabalho, e o impacto que essas modificações trouxeram para o aplicativo.

5.1 Constantes

Constantes se tornam úteis no momento que se deseja utilizar um dado que será acessível por todo o programa, e o mesmo nunca se modificara durante a execução.

Sendo assim o compilador é capaz de reserva um endereço de memória para uma constante, e não ter a necessidade de controlar uma eventual escrita nesse endereço. Otimizando a execução do programa.

Seu uso normalmente é mais expressivo quando é necessário a utilização de um dado de controle para uma passagem de método, ou em linguagens de programação de alto nível, utilizar um nome mais familiar para identificar essa constante e ser facilmente visualizada durante o desenvolvimento do *software*.

5.1.1 Implementação no Verto

No Compilador Educacional Verto a declaração de uma constante se da pela palavra chave “constante” escrito no editor antes do tipo da variável.

```
constante inteiro x = 10;
```

Se o aluno não atribuir um valor para essa variável que foi declarada globalmente, o compilador ira emitir uma mensagem de erro sintático que espera uma atribuição como é mostrado na figura 29.

Figura 29: Erro sintático causado pela não atribuição de uma constante

Se a atribuição foi feita corretamente, mas durante a utilização da constante no programa ela receber uma atribuição, outro erro, dessa vez semântica é mostrado para o aluno.

Informando que uma constante não pode ser modificada, ou seja, não pode receber qualquer tipo de atribuição. O erro apresentado é mostrado na imagem 30.

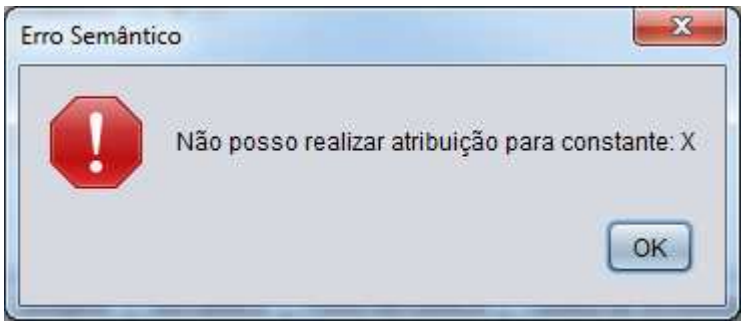

Figura 30: Erro de atribuição de constante

No macro assembler não precisou ser feita nenhuma mudança, já que todo o controle de atribuição fica a cargo do analisador semântico.

5.2 Salto incondicional

Também conhecido como desvio incondicional ou *goto*, deixou de ser amplamente utilizado nas linguagens modernas em sua forma literal, na linguagem Java por exemplo, não tem o comando, mas se tem instruções a mais de controle para laços como o *break*.

Segundo (Dijkstra, 1968), o uso do salto incondicional deve ser descontinuado devido sua forma muito primitiva, assim podendo causar uma dificuldade na legibilidade do código, já que o *goto* força qualquer instrução de programa a seguir qualquer outra sequência de instrução, independente de se essa instrução precede ou se vem depois da primeira na ordem textual.

Então (Knuth,1974) demonstra alguns exemplos que o uso do salto incondicional se torna mais eficiente de forma computacional para resolver um problema lógico. E defende seu uso. Um exemplo prático seria a utilização de vários laços de repetição encadeados e no laço central, contendo uma condição favorável, se deseja sair de todos os laços, a forma mais eficiente seria a utilização da instrução *goto* fazendo o fluxo seguir para depois dos laços.

Apesar dos problemas relatados, para o ensino da disciplina de compiladores que é o foco do Compilador Educacional Verto é de suma importância entender o funcionamento de um salto incondicional, já que todo o laço na sua forma primitiva, ou seja, em linguagem de maquina não deixa de ser uma série de desvios incondicionais.

5.2.1 Rótulo

No Compilador Educacional Verto foi utilizado rótulos, para definir um desvio incondicional no código fonte, rótulo é utilizado em diversas linguagens como C, e tem como objetivo definir um identificador para que seja utilizado como uma marca no código.

Assim quando é utilizado um comando ordenando a mudança do fluxo para aquele rotulo, o compilador saiba para onde o novo endereço será definido no decorrer da execução do programa.

A definição utilizada no Verto foi feita da seguinte forma:

```
#rotulo;
```

Quando o rotulo é inserido no meio de um trecho de código, fica definido para onde o fluxo continuara a execução, o comando que fará esse papel é feito dessa forma:

```
va para #rotulo;
```

Caso o compilador encontre o comando “va para”, e logo em seguida não tenha o sinal de #, com uma palavra em seguida, o analisador sintático mostrara a mensagem exibida na figura 31.

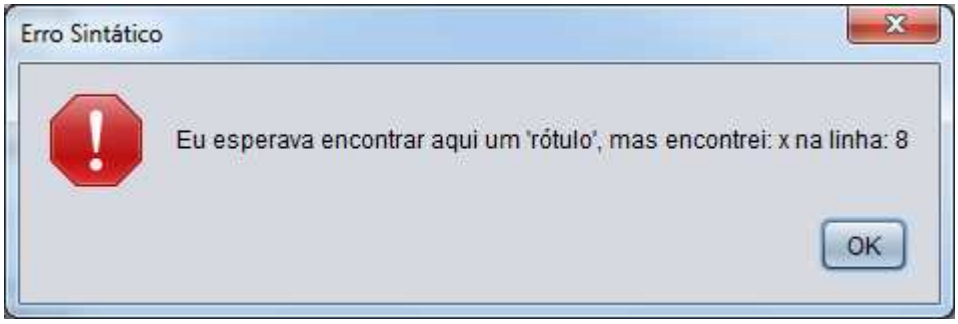

Figura 31: Erro sintático causado pela falta de um rotulo

5.2.2 Macro-assembler

Na estrutura macro-assembler o seguinte código é adicionado quando encontrado um rotulo no código fonte.

```
JMP                    L1; Vai incondicionalmente para: L1( #ROTULO )
```

O *JMP* efetua um salto incondicional para o rótulo definido, que no exemplo acima, é o L1.

5.3 Vetor

Vetores também conhecidos como *arrays*, são extremamente úteis, e indispensáveis em qualquer linguagem moderna, sua utilizarão pratica se dá de diversas formas. Como é uma estrutura de dados com uma ou mais dimensões, que no Compilador Educacional Verto será

utilizado somente unidimensional, normalmente do mesmo tipo, pode-se armazenar uma quantia enorme de dados, que pode ser facilmente recuperada posteriormente.

A organização de um vetor se da, na utilização de um índice, para armazena e recupera os dados daquele vetor.

Se o programador precisar utilizar mais de mil dados em variáveis, seria praticamente inviável isso se feito declarando variável por variável até o numero desejado.

5.3.1 Declaração de vetores

O Compilador Educacional Verto suporta dois tipos de dados para vetores, inteiro e caractere, a declaração é feita adicionando colchetes no final do identificador, entre os colchetes um número deve ser informado, que define o tamanho de *bytes* que esse vetor irá utilizar na memória.

Caso um número inteiro não seja utilizado para a definição do tamanho do vetor, um erro sintático é mostrado ao aluno. Que pode ser observado na figura 32.

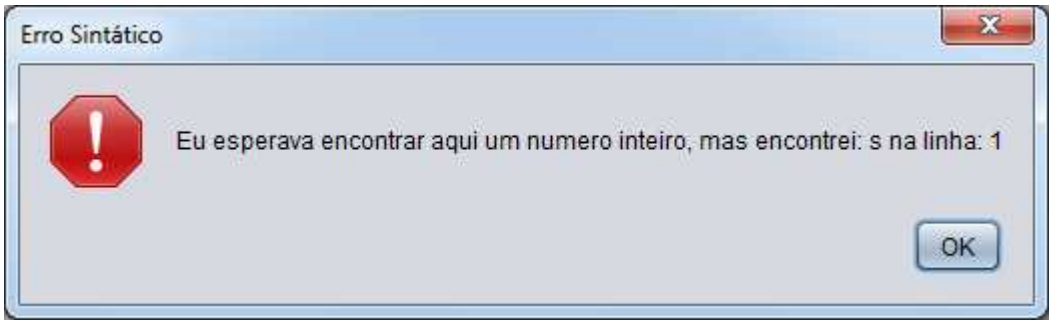

Figura 32: Erro sintático causado por atribuição

5.3.2 Atribuição de vetores

A atribuição é feita da mesma forma que uma variável do tipo inteiro ou caractere, a diferença é que o identificador da posição da memória tem que ser utilizado, uma forma de atribuição pode ser vista no trecho de código a baixo.

```
x[0] = 0;
```

Caso a atribuição efetuada em um determinado vetor seja diferente do tipo que ele foi declarado um erro semântico é mostrado em uma janela de erro para o usuário, como pode ser observado na figura 33.

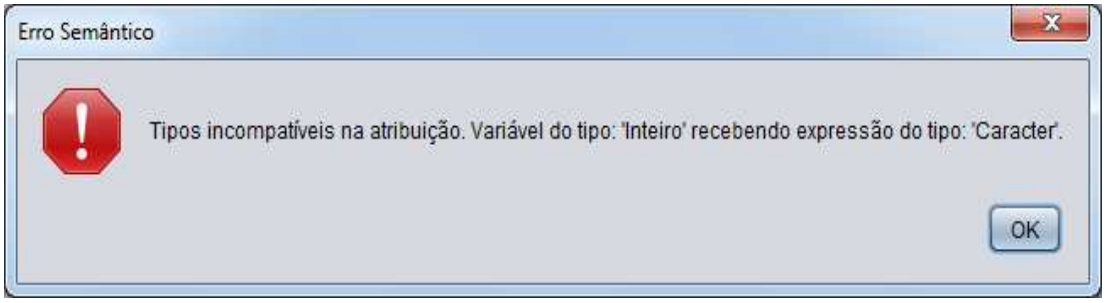

Figura 33: Erro semântico causado por tipos diferentes

5.3.3 Manipulação de Vetores

Depois que um vetor é definido e se quer buscar um dado de determinada posição, é possível a utilização de expressões para buscar tal posição, e mesmo um vetor pode ser utlizado para busca informação de outro vetor.

Nos trechos a seguir são mostradas diversas formas de manipulação suportadas pelo Verto, como a subtração de dois vetores e a atribuição do resultado em um terceiro vetor.

```
x[0] = x[1] - x[2];
```

Também é possível a utilização de expressões na manipulação de vetores que retornem um numero inteiro.

```
x[ 1+2 ] = 0;
```

A utilização do resultado da busca da posição de um vetor como índice para um segundo vetor.

```
x[ y[ 0 ] ] = 0;
```

Quando se utiliza vetores no código, normalmente em algum momento, precisa-se percorrer esse vetor, para isso é utilizado laços, como o “para”. Se necessita incrementar

```
para i = 0 ate tamanho {
        x[ i ] = x[ i ]+1;
}
```

5.3.4 Vetor no Macro-assembler

Na versão do anterior do verto foi usado uma tentativa de utilização do registrador R4 para a manipulação de vetores na maquina hipotética Cesar, que não foi bem sucedida, então nessa versão somente os registrador R0 e R1 foram utilizados para efetuar a manipulação de vetores de forma que se consiga uma utilidade pratica do mesmo.

Na figura 34 um trecho de código do macro-assembler pode ser visualizado com o código contendo uma atribuição simples a um vetor.

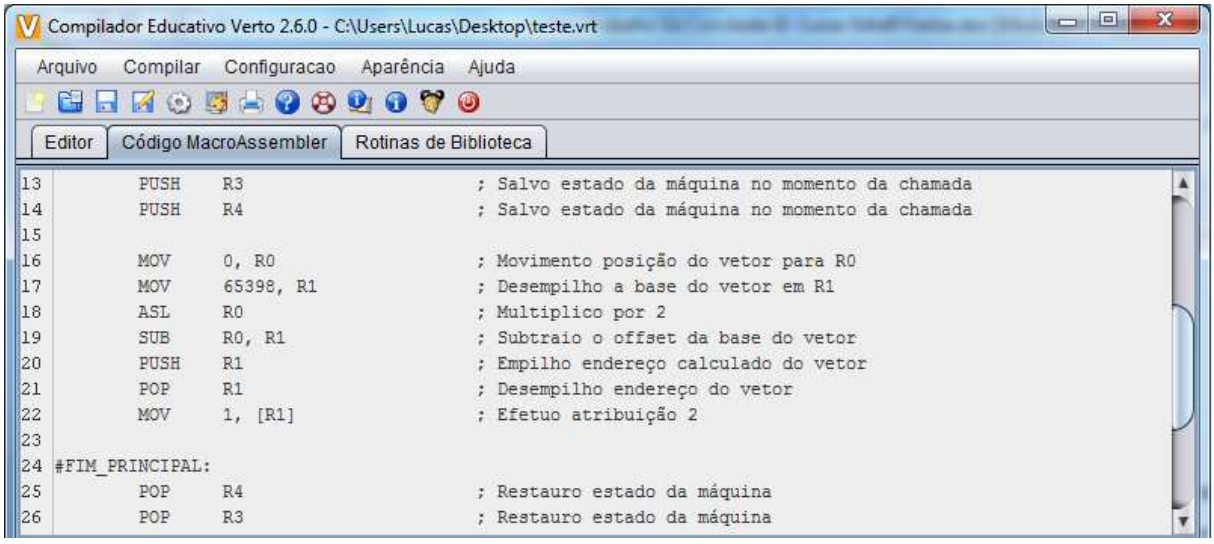

Figura 34: Código macro-assembler com vetor

A utilização do endereço do vetor pode ser observada na linha 17, na figura acima, que o registrador R1 recebe a informação contida naquele endereço.

CONSIDERAÇÕES FINAIS

O estudo de diversas técnicas para aperfeiçoar o Compilador Educacional Verto, tanto na sua construção por meio de métodos de programação visando uma fácil manutenção futura quanto melhorias em sua estrutura visual baseado em telas de sistema e funcional facilitando a interação com o usuário final, foram vitais para que a conclusão desse trabalho fosse alcançada.

A elaboração da fase inicial do projeto teve como premissa o funcionamento das fases sintáticas, léxicas e semânticas encontradas em um compilador. E também a apresentação do aplicativo por meio de amostragem de telas anterior ao desenvolvimento de suas novas estruturas propostas nesse mesmo período. Assim foram apresentadas novas estruturas como vetores, constantes e saltos incondicionais para tornar a ferramenta de ensino mais completa em sua função.

Na segunda parte, o desenvolvimento das estruturas propostas na primeira parte desse trabalho foi concluído com relativo sucesso. A implementação de variáveis do tipo constante não teve problemas e esta funcionando como o proposto. Salto incondicional devido ao tempo de desenvolvimento ficou limitado ao uso de rótulos, não foi possível o desenvolvimento de controle de laços como o comando “Sai”. A elaboração de estrutura de mapeamento finito foi finalizada atingindo todos os objetivos propostos, sendo que foi a maior parte do tempo dispêndio no projeto.

Além das estruturas principais definidas, no decorrer do desenvolvimento observou-se a necessidade de novas implementações não esperadas no anteprojeto, e que também tiveram uma importância para a eficiência da aplicação. Erros resultantes de versões anteriores tiveram que ser corrigidos para não comprometer a usabilidade do aplicativo, e assim garantir uma versão estável para ser submetida ao publico geral.

O problema principal encontrado no desenvolvimento desse projeto foi a implementação da estrutura de vetores. Apesar de ter tomado um tempo maior que as outras tarefas propostas, foi concluída com sucesso.

O Compilador Educacional Verto está com sua nova versão disponível em *http://verto.sourceforge.net/* para o público em geral poder efetuar o *download*, ou a transferência dos arquivos localizados no servidor para o computador. Os arquivos binários que possibilitam a execução do Verto, e os códigos-fonte também se encontram disponíveis para quem tenha interesse poder modificá-lo.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores: princípios, técnicas e ferramentas.** Rio de Janeiro: LTC, 1995. 338p.

CRESPO, Rui Gustavo. Processadores de linguagens: da concepção à implementação. Lisboa: IST Press, 1998. 435p.

DIJKSTRA, W. Edsger. **A case against the GO TO Statement**. The Netherlands: ACM, 1968

DELAMARO, Márcio Edurdo. Como construir um compilador utilizando ferramentas JAVA. São Paulo: Novatec, 2004. 308p.

HORSTMANN, Cay. **Big Java**. Porto Alegre: Bookman, 2004. 1111p.

KAKDE, O. G. **Algorithms for compiler design**. Massachusetts: Charles River Media, 2003. 334p.

KNUTH, E. Donald. **Structured Programming with go to Statements**. California: Computing Surveys, 1974. 301p.

LEVINE, R. John; MASON, Tony; BROWN, Doug. **Lex & Yacc**. Beijing: O´Reilly, 1995. 363p.

LEWIS, R. Harry; PAPADIMITRIOU, H. Christos. **Elementos de teoria da computação**. Porto Alegre: Bookman, 2000.

LOUDEN, Kenneth C. **Compiladores: princípios e práticas**. São Paulo: Thomson, 2004. 569p.

MEC. **Diretrizes Curriculares de Cursos da área de Computação e Informática**. Disponível em <http://www.mec.gov.br/sesu/ftp/curdiretriz/computacao/co_diretriz.rtf> Acesso em: 10 de outubro de 2007.

MUCHNICK, Steven S.**Advanced Compiler Design Implementation**. San Francisco: California: Morgan Kaufmann Publishers, 1997. 856p.

NETO, João J. **Introdução à Compilação**. Rio de Janeiro: LTC: 1987.

OLIVEIRA, Ricardo Ferreira de. **Compiladores – Aula 7.** 2000. Disponível em: <http://www.quatro.com.br/compila/>. Acesso em: 17 out. 2007.

OLIVEIRA, Ricardo F.; SCHNEIDER, Carlos S.; FERREIRA, Paulo R.; KORNDORFER, Fernando O. **Verto Compiler**. Disponível em <http://sourceforge.net/projects/verto> Acesso em: 12 de setembro de 2007.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. **Implentaçãod e linguagens de programação: compiladores**. Porto Alegre: Sagra-Luzzatto: 2000. 216p.

RANGEL, J. L. M. **Material didático relativo a disciplina de Compiladores**. 1999. Disponível em: <http://www-di.inf.pucrio. br/~rangel/comp.html >. Acesso em: 7 set. 2007.

SALOMAA, Arto. **Formal languages.** New York: Academic Press, Inc.l, 1973. 322p.

SCHNEIDER, C. S.; PASSERINO, L. M.; OLIVEIRA, R. F. (2005) **Compilador Educativo VERTO: ambiente para aprendizagem de compiladores**. RENOTE - Revista Novas Tecnologias na Educação, Porto Alegre, v. 3, n. 2.

SEBESTA, Robert W. **Conceitos de linguagem de programação.** São Paulo: Bookman 2002. 623p.

SERSON, Roberto Rubinstein. **Programação orientada a objetos com java 6.** Rio de Janeiro: Brasport 2007. 463p.

SMITH, James; FRANK, Thomas. **Introduction to programming concepts and methods with Ada.** New York: McGrawHill, 1994. 545p.

SHALLOWAY, Alan; TROTT, James. **Explicando Padrões de Projeto.** Porto Alegre: Bookman, 2004. 328p.

WATT, David A.; BROWN, Deryck F. **Programming language processors in Java.** Edinburg: Pearson, 1999. 436p.

_WEBER, R. F. **Fundamentos de arquitetura de computadores**. Porto Alegre: Sagra Luzzato. 2001. 299 p.

WEBER, R. F. **Material didático sobre *Cesar***. 1999, Disponível em <ftp://ftp.inf.ufrgs.br/pub/inf108/Cesar.ppt>

APENDICE

Programa Fatorial, escrito no Compilador Verto.

```
inteiro a, b;

caracter c;


prototipo inteiro fatorial( inteiro x );


funcao inteiro principal () {


        LimpaVisor;


        escreva( 1, "Numero: " );

        leia ( 10, c );


        limpaVisor;

        a = paraInteiro( c );

        b = fatorial( a );
```

```
        escreva( 1, "Fatorial: " );
        escreva( 12, b );
    }

    funcao inteiro fatorial( inteiro x ) {
        se ( x < 2 ) entao {
            retorne 1;
        } senao {
            retorne fatorial( x - 1 ) * x;
        }
    }
```

Programa fatorial, já compilado pelo Verto para a máquina hipotética César:

```
; FATORIAL.ASM
        MOV   65400, R6          ; Inicializa ponteiro da Pilha
        SUB   36, R6             ; Reserva espaço para variáveis globais
        JSR   PRINCIPAL          ; Chama rotina principal
        HLT                      ; Encerra execução

PRINCIPAL:
        PUSH  R5                 ; Preservo o Base Pointer da função chamadora
        MOV   R6, R5             ; Copio endereço/pilha do Base Pointer (inicio-
locais)
```

```
        PUSH R0         ; Salvo estado da máquina no momento da
chamada
        PUSH R1         ; Salvo estado da máquina no momento da
chamada
        PUSH R2         ; Salvo estado da máquina no momento da
chamada
        PUSH R3         ; Salvo estado da máquina no momento da
chamada
        PUSH R4         ; Salvo estado da máquina no momento da
chamada

        JSR  #CLRSCR            ; Limpa o visor do Cesar
        MOV  C1, R0             ; Move o endereço do que se quer escrever
        MOV  1, R1              ; Move a posição inicial para escrever
        JSR  #EXIBE             ; Exibe
        MOV  65394, R0          ; Movimento área para receber leitura
        MOV  10, R1             ; Move a posição inicial para ler para R1
        MOV  29, R2             ; move para R2, o tamanho da área de leitura
máxima
        JSR  #GETS              ; Chamo rotina de leitura
        JSR  #CLRSCR            ; Limpa o visor do Cesar
        MOV  65394, R0          ; Movimento expressão a escrever para R0
        JSR  #ATOI              ; Converte de cadeia de caracteres para inteiro
```

```
        PUSH R0             ; Empilho resultado da conversão da string para
inteiro

        POP  R0             ; Desempilho resultado da expressão

        MOV  R0, [65398]    ; Efetuo atribuição

        SUB  2, R6          ; Abro espaço para o valor do retorno

        PUSH [65398]        ; Empilho argumento

        JSR  FATORIAL       ; Realizo a chamada da função

        ADD  2, R6          ; Desempilho espaço usado pelos argumentos

        POP  R0             ; Desempilho resultado da expressão

        MOV  R0, [65396]    ; Efetuo atribuição

        MOV  C2, R0         ; Move o endereço do que se quer escrever

        MOV  1, R1          ; Move a posição inicial para escrever

        JSR  #EXIBE                 ; Exibe

        MOV  [65396], R0    ; Movimento expressão a escrever para R0

        MOV  C0, R1         ; move para R1, o endereço da posição onde a
variavel será convertida

        JSR  #ITOA          ; Converte de inteiro para cadeia de caracteres

        MOV  C0, R0         ; Move o endereço do que se quer escrever

        MOV  12, R1         ; Move a posição inicial para escrever

        JSR  #EXIBE                 ; Exibe


    #FIM_PRINCIPAL:
```

```
        POP    R4          ; Restauro estado da máquina
        POP    R3          ; Restauro estado da máquina
        POP    R2          ; Restauro estado da máquina
        POP    R1          ; Restauro estado da máquina
        POP    R0          ; Restauro estado da máquina
        POP    R5          ; Restaura o Base Pointer da Função chamadora
        RTS

    FATORIAL:
        PUSH   R5          ; Preservo o Base Pointer da função chamadora
        MOV    R6, R5      ; Copio endereço/pilha do Base Pointer (inicio-locais)
        PUSH   R0          ; Salvo estado da máquina no momento da chamada
        PUSH   R1          ; Salvo estado da máquina no momento da chamada
        PUSH   R2          ; Salvo estado da máquina no momento da chamada
        PUSH   R3          ; Salvo estado da máquina no momento da chamada
        PUSH   R4          ; Salvo estado da máquina no momento da chamada

        MOV    2, R0       ; Movimento primeira expressão simples para R0
```

```
        CMP   [R5+4], R0        ; Efetuo comparação de Inteiros
        BLT   L1                ; Compara se menor
        PUSH  0                 ; Falso
        JMP   L2                ;
L1:
        PUSH  1                 ; Verdadeiro
L2:
        NOP                     ;
        POP   R0                ; Desempilho resultado da expressão
        CMP   0, R0             ; Testo se falso
        BEQ   L3                ; Se falso, vai para: L3
        MOV   1, [R5+6]         ; Transfiro valor de retorno
        JMP   L4                ; Vai para o fim do verdadeiro: L4
L3:
        NOP                     ; Inicio do Senao: L3
        SUB   2, R6             ; Abro espaço para o valor do retorno
        MOV   [R5+4], R0        ; Movimento primeiro operador da adição para
R0
        SUB   1, R0             ; Efetuo subtração de Inteiros
        PUSH  R0                ; Empilho temporária inteira
        JSR   FATORIAL          ; Realizo a chamada da função
        ADD   2, R6             ; Desempilho espaço usado pelos argumentos
```

```
        POP   R0            ; Desempilho temporária inteira
        MOV   [R5+4], R1    ; Movimento segundo operador da multiplicação para R1
        JSR   #IMUL         ; Chamo rotina de multiplicação de Inteiros
        PUSH  R0            ; Empilho temporária inteira
        POP   R0            ; Desempilho resultado da expressão
        MOV   R0, [R5+6]    ; Transfiro valor de retorno
    L4:
        NOP                 ; fim do se: L4

    #FIM_FATORIAL:
        POP   R4            ; Restauro estado da máquina
        POP   R3            ; Restauro estado da máquina
        POP   R2            ; Restauro estado da máquina
        POP   R1            ; Restauro estado da máquina
        POP   R0            ; Restauro estado da máquina
        POP   R5            ; Restaura o Base Pointer da Função chamadora
        RTS


    C0:
```

" "

C1:

"Numero: "

C2:

"Fatorial: "