UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

IMPLEMENTAÇÃO DE MAPEAMENTO FINITO (ARRAY'S) NO AMBIENTE FURBOL

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

ANDERSON ADRIANO

BLUMENAU JULHO/2001

2001/1-03

IMPLEMENTAÇÃO DE MAPEAMENTO FINITO (ARRAY'S) NO AMBIENTE FURBOL

ANDERSON ADRIANO

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. José Roque Voltolini da Silva — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. José Roque Voltolini da Silva

_____

Prof. Antônio Carlos Tavares

_____

Prof. Dalton Solano dos Reis

SUMÁRIO

LISTA DE FIGURAS VI

LISTA DE QUADROS VII

RESUMO XII

ABSTRACT XIII

1 INTRODUÇÃO 1

1.1 OBJETIVOS 2

1.2 ORGANIZAÇÃO DO TEXTO 3

2 CONCEITOS BÁSICOS 4

2.1 COMPILADORES 4

2.1.1 ANÁLISE LÉXICA 5

2.1.2 ANÁLISE SINTÁTICA 5

2.1.2.1 GRAMÁTICAS LIVRES DE CONTEXTO 7

2.1.2.2 ANÁLISE SINTÁTICA *TOP DOWN* 8

2.1.2.3 FATORAÇÃO À ESQUERDA 9

2.1.2.4 ELIMINANDO RECURSÃO À ESQUERDA 10

2.1.3 ANÁLISE SEMÂNTICA 11

2.1.3.1 DEFINIÇÕES DIRIGIDAS PÔR SINTAXE 12

2.1.3.1.1 GRAMÁTICA DE ATRIBUTOS 12

2.1.3.1.1.1 ATRIBUTOS SINTETIZADOS 13

2.1.3.1.1.2 ATRIBUTOS HERDADOS 13

2.1.3.1.13 ELIMINAÇÃO DE RECURSÃO À ESQUERDA DE UM ESQUEMA DE TRADUÇÃO 14

2.2 MAPEAMENTO FINITO (*ARRAY'S*) 17

2.2.1 *ARRAY'S* E ÍNDICES....17

2.2.2 CATEGORIAS DE *ARRAY'S*....18

2.2.2.1 EXEMPLOS DE UTILIZAÇÃO DE *ARRAY'S* NAS LINGUAGENS DE PROGRAMAÇÃO....19

2.2.3 NÚMERO DE SUBSCRITOS EM *ARRAY'S*....21

2.2.4 INICIALIZAÇÃO DE *ARRAY'S*....22

2.2.5 IMPLEMENTAÇÃO DO TIPO *ARRAY*....23

2.3 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO....27

2.3.1 CÓDIGO DE TRÊS ENDEREÇOS....28

2.3.2 TRADUÇÃO DIRIGIDA PELA SINTAXE EM CÓDIGO DE TRÊS ENDEREÇOS....28

2.4 DEFINIÇÃO DE ESCOPOS....30

2.5 GERAÇÃO DE CÓDIGO....30

2.5.1 PROGRAMA-ALVO....31

2.5.2 GERENCIAMENTO DE MEMÓRIA....31

2.5.3 SELEÇÃO DE INSTRUÇÕES....32

2.5.4 ALOCAÇÃO DE REGISTRADORES....34

2.5.5 MÁQUINA-ALVO....34

2.5.5.1 ARQUITETURA DOS MICROPROCESSADORES 8088....35

2.5.5.1.1 REGISTRADORES DE PROPÓSITO GERAL....35

2.5.5.1.2 REGISTRADORES PONTEIROS E DE ÍNDICE....36

2.5.5.1.3 REGISTRADORES DE SEGMENTO....37

2.5.5.1.4 REGISTRADOR DE SINALIZADORES....37

2.5.5.1.5 MODOS DE ENDEREÇAMENTO....37

2.6 LINGUAGEM *ASSEMBLY*....38

2.6.1 CARACTERÍSTICAS GERAIS DO *ASSEMBLY*....39

2.6.2 INSTRUÇÕES DA LINGUAGEM *ASSEMBLY* 39
2.6.3 INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS 40
2.6.4 INSTRUÇÕES ARITMÉTICAS 41
2.6.5 INSTRUÇÕES LÓGICAS 42
2.6.6 INSTRUÇÕES DE CONTROLE DE FLUXO 42
2.6.7 MANIPULAÇÃO DE *STRINGS* 43
2.6.8 SUBPROGRAMAS 44
2.6.9 ESTRUTURA DOS ARQUIVOS *.COM* 45
2.6.10 PREFIXO DE SEGMENTO DE PROGRAMA (PSP) 48
2.7 AMBIENTE FURBOL 48
2.7.1 PRINCIPAIS CARACTERÍSTICAS DA LINGUAGEM FURBOL 48
2.7.2 INTERFACE DO AMBIENTE FURBOL 49
3 DESENVOLVIMENTO DO PROTÓTIPO 52
3.1 DEFINIÇÃO DE ESCOPO IMPLEMENTADA 52
3.2 ESPECIFICAÇÃO DA LINGUAGEM FURBOL 55
3.2.1 PROGRAMAS E BLOCOS 56
3.2.2 ESTRUTURAS DE DADOS 57
3.2.3 ESTRUTURA DE SUBROTINAS 60
3.2.4 ESTRUTURA DE COMANDOS 64
3.2.5 ESTRUTURA DE CONTROLE DE EXPRESSÕES 68
4 APRESENTAÇÃO DO PROTÓTIPO 74
4.1 CARACTERÍSTICAS DO AMBIENTE FURBOL APÓS EXTENSÃO 74
5 CONCLUSÃO 78
5.1 EXTENSÕES 78
REFERÊNCIAS BIBLIOGRÁFICAS 79

LISTA DE FIGURAS

FIGURA 1 – PROCESSO DE COMPILAÇÃO.....4

FIGURA 2 – TELA PRINCIAL DO AMBIENTE FURBOL.....49

FIGURA 3 – DETECÇÃO DE ERROS NO PROTÓTIPODESENVOLVIMENTO DO PROTÓTIPO 50

FIGURA 4 – JANELA PRINCIPAL COM CÓDIGO INTERMEDIÁRIO 51

FIGURA 5 – JANELA DO PROTÓTIPO COM CÓDIGO ASSEMBLY 51

LISTA DE QUADROS

QUADRO 1 – PRODUÇÃO REPRESENTANDO ENUNCIADO CONDICIONAL.....7

QUADRO 2 – UMA CONSTRUÇÃO DE CONTROLE DE FLUXO 9

QUADRO 3 – EXEMPLO DE RECURSÃO À ESQUERDA 9

QUADRO 4 – PRODUÇÃO GRAMATICAL DE CONTROLE DE FLUXO 10

QUADRO 5 – EXEMPLO DE FATORAÇÃO À ESQUERDA 10

QUADRO 6 – PRODUÇÃO RECURSIVA À ESQUERDA 10

QUADRO 7 – PRODUÇÕES NÃO-RECURSIVAS 11

QUADRO 8 – PRODUÇÕES AGRUPADAS.....11

QUADRO 9 – PRODUÇÕES SUBSTITUÍDAS.....11

QUADRO 10 – DEFINIÇÃO DE UMA CALCULADORA DE MESA SIMPLES.....13

QUADRO 11 – DEFINIÇÃO TENDO *L.IN* COMO ATRIBUTO HERDADO.....14

QUADRO 12 – DEFINIÇÃO DE TRADUÇÃO COM UMA GRAMÁTICA RECURSIVA À ESQUERDA.....15

QUADRO 13 - ESQUEMA DE TRADUÇÃO GENÉRICO.....15

QUADRO 14 - ESQUEMA DE TRADUÇÃO TRANSFORMADO COM GRAMÁTICA RECURSIVA À DIREITA.....16

QUADRO 15 – GRAMÁTICA CONSTRUÍDA A PARTIR DO QUADRO 13.....16

QUADRO 16 - ESQUEMA TRANSFORMADO COM AÇÕES SEMÂNTICAS.....16

QUADRO 17 – EXEMPLO SIMBÓLICO DE *ARRAY*.....17

QUADRO 18 – EXEMPLO DE ARRAY *STACK*-DINÂMICO.....19

QUADRO 19 – DECLARAÇÃO DE *ARRAY* DINÂMICO.....19

QUADRO 20 – ALOCAÇÃO DE *ARRAY* DINÂMICO.....19

QUADRO 21 – DESALOCAÇÃO DE *ARRAY* DINÂMICO.....20

QUADRO 22 – EXEMPLO DE *ARRAY'S* NO PASCAL.....21

QUADRO 23 – EXEMPLO DE CHAMADA DE *ARRAY'S* NO PASCAL.....21

QUADRO 24 – DECLARAÇÃO DE ARRAY EM LINGUAGEM C.....22

QUADRO 25 – EXEMPLO DE INICIALIZAÇÃO DE *ARRAY'S* NO FORTRAN 77.....22

QUADRO 26 – EXEMPLO DE INICIALIZAÇÃO DE *ARRAY'S* NA LINGUAGEM C.....22

QUADRO 27 – *ARRAY'S* DE CHAR NA LINGUAGEM C.....23

QUADRO 28 – *ARRAY'S* DE STRINGS NA LINGUAGEM C.....23

QUADRO 29 – DECLARAÇÃO E INICIALIZAÇÃO DE *ARRAY'S* NA LINGUAGEM ADA.....23

QUADRO 30 – LOCALIZAÇÃO DE *ARRAY'S*.....24

QUADRO 31 – LOCALIZAÇÃO DE *ARRAY'S* PARA AVALIAÇÃO EM TEMPO DE COMPILAÇÃO.....24

QUADRO 32 – ORGANIZAÇÃO DE *ARRAY'S*.....25

QUADRO 33 – FÓRMULA PARA *ARRAY* BIDIMENSIONAL.....25

QUADRO 34 – REESCREVENDO A FÓRMULA PARA *ARRAY* BIDIMENSIONAL.....25

QUADRO 35 - GENERALIZANDO A FÓRMULA PARA *ARRAY* DE *K* DIMENSÕES.....26

QUADRO 36 – GRAMÁTICA PARA REFERENCIAR *ARRAY'S*.....26

QUADRO 37 – ÍNDICES A SEREM COMPUTADOS.....27

QUADRO 38 – FÓRMULA DE RECORRÊNCIA.....27

QUADRO 39 – ENUNCIADOS DE TRÊS ENDEREÇOS.....28

QUADRO 40 – TRADUÇÃO DO QUADRO 39.....28

QUADRO 41 – DEFINIÇÃO DIRIGIDA PELA SINTAXE.....29

QUADRO 42 – ENUNCIADO DE ATRIBUIÇÃO EM LINGUAGEM-FONTE.....29

QUADRO 43 – CÓDIGO GERADO A PARTIR DA EXPRESSÃO DO QUADRO 42.....29

QUADRO 44 – DECLARAÇÃO DE VARIÁVEIS.....32

QUADRO 45 – SEQÜÊNCIA DE CÓDIGO PARA ENUNCIADOS DE TRÊS ENDEREÇOS 33
QUADRO 46 – SEQÜÊNCIA DE ENUNCIADOS DE TRÊS ENDEREÇOS 33
QUADRO 47 – TRADUÇÃO DO QUADRO 46 33
QUADRO 48 – SEQÜÊNCIA DE CÓDIGO INEFICIENTE 34
QUADRO 49 – UNIDADE CENTRAL DE PROCESSAMENTO 8088 35
QUADRO 50 – CONJUNTO BASE DE REGISTRADORES DO 8088 36
QUADRO 51 – CÁLCULO DO ENDEREÇO DE INÍCIO DA PRÓXIMA INSTRUÇÃO. . 37
QUADRO 52 – REGISTRADOR DE SINALIZADORES 37
QUADRO 53 – EXEMPLO DE ENDEREÇAMENTO INDEXADO 38
QUADRO 54 – CAMPOS DE UMA LINHA EM LINGUAGEM *ASSEMBLY* 39
QUADRO 55 – EXEMPLO DE PROGRAMA-FONTE EM LINGUAGEM *ASSEMBLY* 40
QUADRO 56 – INSTRUÇÕES BÁSICAS DA LINGUAGEM ASSEMBLY 40
QUADRO 57 – INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS 41
QUADRO 58 – INSTRUÇÕES ARITMÉTICAS 41
QUADRO 59 – INSTRUÇÕES PARA OPERAÇÕES LÓGICAS 42
QUADRO 60 – INSTRUÇÕES DE CONTROLE DE FLUXO 43
QUADRO 61 – INSTRUÇÕES DE MANIPULAÇÃO DE *STRINGS* 44
QUADRO 62 – PROCEDIMENTOS EM LINGUAGEM *ASSEMBLY* 44
QUADRO 63 – ESTRUTURA DE UM ARQUIVO *.COM* 46
QUADRO 64 – PROGRAMA PASCAL COM PROCEDIMENTOS ANINHADOS 52
QUADRO 65 – TABELAS DE SÍMBOLOS PARA PROCEDIMENTOS ANINHADOS ... 53
QUADRO 66 – DEFINIÇÃO DIRIGIDA PELA SINTAXE PARA DECLARAÇÕES DE PROCEDIMENTOS ANINHADOS 54
QUADRO 67 – DEFINIÇÃO DE PROGRAMAS E BLOCOS 57
QUADRO 68 - EXEMPLO DE DECLARAÇÕES DE *ARRAY'S* 58

QUADRO 69 - TABELA DE SÍMBOLOS PARA DECLARAÇÕES DE *ARRAY'S*.....58
QUADRO 70 – DEFINIÇÃO DAS ESTRUTURAS DE DADOS.....58
QUADRO 71 – DEFINIÇÃO DAS ESTRUTURAS DE DADOS (CONTINUAÇÃO).....60
QUADRO 72 – DEFINIÇÃO DA ESTRUTURA DE SUBROTINAS.....61
QUADRO 73 – DEFINIÇÃO DA ESTRUTURA DE SUBROTINAS (CONTINUAÇÃO)..62
QUADRO 74 – EXEMPLO DE DECLARAÇÃO DE PROCEDIMENTO NO FURBOL 62
QUADRO 75 – TRADUÇÃO EM *ASSEMBLY* DO QUADRO 71.....63
QUADRO 76 – ESTADOS DA PILHA NA CHAMADA DE UM PROCEDIMENTO.....63
QUADRO 77 – DEFINIÇÃO DA ESTRUTURA DE COMANDOS.....65
QUADRO 78 – DEFINIÇÃO DA ESTRUTURA DE ATRIBUIÇÃO.....65
QUADRO 79 – DEFINIÇÃO DE CHAMADAS DE PROCEDIMENTOS 66
QUADRO 80 – DEFINIÇÃO DA ESTRUTURA DE COMANDOS DE REPETIÇÕES.....66
QUADRO 81 – DEFINIÇÃO DA ESTRUTURA DE COMANDOS CONDICIONAIS.....67
QUADRO 82 – DEFINIÇÃO DA ESTRUTURA DO COMANDO DE ENTRADA 67
QUADRO 83 – DEFINIÇÃO DA ESTRUTURA DO COMANDO DE SAÍDA 67
QUADRO 84 – DEFINIÇÃO DOS COMANDOS DE INCREMENTO E DECREMENTO.68
QUADRO 85 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES 68
QUADRO 86 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO) 69
QUADRO 87 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO) 70
QUADRO 88 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO) 71
QUADRO 89 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO) 72

QUADRO 90 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO) 73

QUADRO 91 – EXEMPLO DE DECLARAÇÃO DE *ARRAY'S* NO AMBIENTE FURBOL 75

QUADRO 92 - TRADUÇÃO EM CÓDIGO ASSEMBLER DO QUADRO 89 77

QUADRO 93 - TRADUÇÃO EM CÓDIGO ASSEMBLER DO QUADRO 89 (CONTINUAÇÃO 77

QUADRO 94 – EXPRESSÃO GERADA PELO *ARRAY* A 77

QUADRO 95 – EXPRESSÃO GERADA PELO *ARRAY* B 77

RESUMO

Este trabalho descreve o desenvolvimento do protótipo de um ambiente de programação em uma linguagem bloco-estruturada com vocabulário na língua portuguesa. O referido trabalho é baseado no trabalho de conclusão de curso de Geovânio Batista André (André, 2000), estendendo-se através da inclusão de novas construções para implementação de mapeamento finito (*array's*) e geração do código executável (programas *.COM*) para microprocessadores da família iAPX 86/88. Para definição formal da sintaxe da linguagem utilizou-se o método BNF (*Backus Normal Form*) e para definição de análise semântica foi utilizado o método de gramáticas de atributos.

ABSTRACT

This assignment describes the development of a prototype of a programming environment in a language block-structured with vocabulary in the Portuguese language. The assignment is based on the work of conclusion of course of Geovânio Batista André (André, 2000), extending through the inclusion of new constructions for implementation of finit mapping (*array's*) and executable code (programs *.COM*), for microprocessors iAPX 86/88. For formal definition of the syntax of the language the method BNF (*Backus Normal Form*) was used, and for definition of semantic analysis the method of grammars of attributes was used.

1 INTRODUÇÃO

A concepção inicial da criação do ambiente FURBOL teve início em 1987, através da experiência relatada no artigo Silva (1987), apresentado no I Simpósio de Engenharia de Software.

Em 1992, houve uma continuidade do trabalho Editor Dirigido por Sintaxe, financiado pela Universidade Regional de Blumenau, através do Programa de Iniciação a Pesquisa (PIPe). O referido trabalho teve como orientador o professor José Roque Voltolini da Silva e como bolsista o acadêmico Douglas Nazareno Vargas (Vargas, 1992).

Após, houve uma continuidade do trabalho pelo acadêmico Joilson Marcos da Silva, através de um Trabalho de Conclusão de Curso (TCC) apresentado no primeiro semestre do ano de 1993, com o título "Desenvolvimento de um Ambiente de Programação para a Linguagem Portugol" (Silva, 1993).

Em seguida, houve uma continuidade do referido trabalho pelo acadêmico Douglas Nazareno Vargas, também através de um TCC apresentado no segundo semestre do ano de 1993, com o título "Definição e Implementação no Ambiente Windows de uma Ferramenta para o Auxílio no Desenvolvimento de Programas" (Vargas,1993).

No segundo semestre do ano de 1996, através do TCC cujo título é "Definição de um Interpretador para a Linguagem "PORTUGOL" utilizando Gramática de Atributos" (Bruxel, 1996), houve uma nova definição do ambiente, utilizando Gramática de Atributos (Knuth, 1968).

Após, houve uma extensão do referido trabalho através de um TCC apresentado no segundo semestre de 1997, com o título "Protótipo de um Ambiente para Programação em uma Linguagem Bloco Estruturada com Vocabulário na Língua Portuguesa" (Radloff, 1997). Este novo ambiente implementou novas construções, como chamadas de procedimentos e recursividade, entre outras. Ainda, melhorou a interface com o usuário, utilizando para a implementação do ambiente, a linguagem de programação Visual Basic (Orvis, 1994). Também gerou o código objeto para a Máquina de Execução para Pascal (MEPA) proposta por Kowaltowiski (1983). Um interpretador para a MEPA também foi criado. Para depuração, foi dada a opção para visualização da execução passo a passo com acompanhamento do

comando no programa fonte associado com as instruções em código de máquina. A visualização da pilha de execução também foi disponibilizada para ser mostrada.

No primeiro semestre de 1999, o TCC com o título “Implementação de Registros e Métodos de Passagem de Parâmetros no Ambiente FURBOL” (Schimt, 1999), o qual foi uma continuidade no trabalho de Radloff (1997), estendendo-o com a implementação de novas construções, tais como produto cartesiano (registros) e métodos de passagem de parâmetros (cópia valor e referência). Ainda, alguns ajustes na definição formal foram realizadas. A BNF (*Backus Normal Form*) e gramática de atributos foram os métodos usados para a especificação formal da linguagem. O referido software foi implementado no ambiente DELPHI 3.0. Também, a nível de depuração, além das opções existentes no ambiente desenvolvido por Radloff (1997), foi disponibilizada a opção para visualização do vetor de registradores de base.

No segundo semestre de 2000, houve uma extensão no trabalho de Schimt (1999), com o título “Protótipo de Gerador de Código Executável a partir do Ambiente FURBOL” (André, 2000). Esta nova extensão inclui uma visualização da tradução do código fonte para o código intermediário que é composto de enunciados de três endereços. Também como saída o protótipo gera o código equivalente em linguagem *Assembly*, gerando a partir deste código um arquivo executável com extensão *.COM*, através do montador *Turbo Assembler* (Swan, 1989).

A especificação da linguagem FURBOL apresentada em André (2000) será utilizada, estendendo-a através da introdução de mapeamento finito *(array’s).*

A implementação será feita no ambiente Delphi 5.0, que oferece uma grande gama de objetos já prontos para facilitar a confecção da interface com o usuário.

A escolha do assunto do trabalho foi motivada por ser uma continuação de um projeto iniciado em 1987, o qual vem sendo aprimorado através de trabalhos de conclusão de curso.

1.1 OBJETIVOS

O trabalho proposto tem como objetivo principal ampliar o ambiente de programação FURBOL, apresentado em André (2000). Esta ampliação consiste na extensão para suportar o

uso de mapeamento finito. Para tanto, especificações serão incluídas para dar suporte a definição de estruturas e comando para fazer o manuseio de mapeamentos finitos. O código de máquina para as referidas construções também será implementado.

1.2 ORGANIZAÇÃO DO TEXTO

O capítulo 1 apresenta a introdução do trabalho contendo alguns conceitos fundamentais sobre linguagens de programação, bem como a apresentação dos objetivos e a organização do texto.

No capítulo 2 é apresentada a fundamentação teórica com uma breve descrição dos conceitos relacionados a compiladores e as técnicas utilizadas para implementação dos três principais módulos que o compõem. É descrito ainda neste capítulo a estrutura básica dos microprocessadores 8088/8086, comandos básicos da linguagem de programação *assembly*. Também é feita a apresentação do protótipo FURBOL desenvolvido por André (2000).

No capítulo 3 é apresentado o desenvolvimento do protótipo com a descrição das técnicas utilizadas para implementação da definição de escopo de variáveis e declarações e chamadas de procedimentos. É apresentada ainda neste capítulo a especificação do protótipo com as definições da linguagem FURBOL e apresenta-se também o protótipo a nível de usuário.

O capítulo 4 apresenta a conclusão do trabalho e sugestões de trabalhos futuros.

2 CONCEITOS BÁSICOS

Neste capítulo serão apresentados relatos sobre compiladores, técnicas utilizadas para implementação, fundamentos para implementação de mapeamento finito (*array's*) e também sobre geração de código executável (código de máquina).

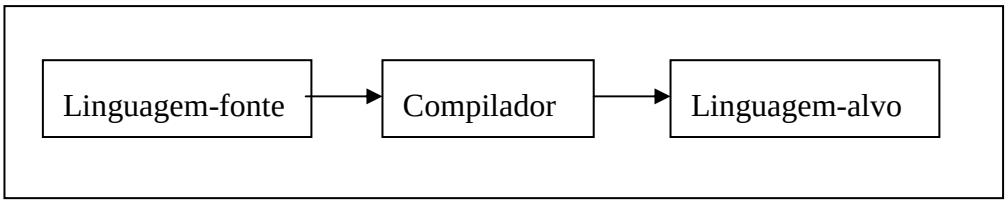
2.1 COMPILADORES

Compilador conforme descrito em José (1987), trata-se de um dos módulos do software básico de um computador, cuja função é a de efetuar automaticamente a tradução de textos, redigidos em uma determinada linguagem de programação, para alguma outra forma que viabilize sua execução. Em geral esta forma é a de uma linguagem de máquina.

Segundo Aho (1995), posto de forma simples, um compilador é um programa que lê um programa escrito em uma linguagem e o traduz num programa equivalente numa outra linguagem chamada de *linguagem-alvo*, como mostrado na fig. 1.

Como parte importante desse processo de tradução, o compilador relata ao seu usuário a existência de erros no programa-fonte.

FIGURA 1 – PROCESSO DE COMPILAÇÃO



A linguagem alvo pode ser uma outra linguagem de programação ou a linguagem de máquina. Pode-se construir compiladores para uma ampla variedade de linguagens fonte e linguagens de máquinas alvo, usando as mesmas técnicas básicas para realizarem as atividades de análise léxica, análise sintática e análise semântica.

2.1.1 ANÁLISE LÉXICA

Conforme descrito em José (1987), a análise léxica implementa uma das três grandes atividades desempenhadas pelos compiladores, das quais constitui aquela que faz a interface entre o texto-fonte e os programas encarregados de sua análise e tradução.

Sua missão fundamental é a de, a partir do texto-fonte de entrada, fragmentá-lo em seus componentes básicos (chamados de partículas, átomos ou *tokens*), identificando trechos elementares completos e com identidade própria, porém individuais para efeito de análise por parte dos demais programas do compilador.

Uma vez identificadas estas partículas do texto-fonte, estas devem ser classificadas segundo o tipo a que pertencem, uma vez que para o módulo da análise sintática, que deverá utilizá-las em seguida, a informação mais importante acerca destas partículas é a classe à qual pertencem, e não propriamente o seu valor. Do ponto de vista do módulo de geração do código no entanto, o valor assumido pelos elementos básicos da linguagem é que fornece a informação mais importante para obtenção do código-objeto. Assim sendo, tanto a classe como o valor assumido pelos componentes básicos da linguagem devem ser preservados pela análise léxica (José, 1987).

2.1.2 ANÁLISE SINTÁTICA

O segundo grande bloco componente dos compiladores e que se pode caracterizar como o mais importante, na maioria dos compiladores, por sua característica de controlador das atividades do compilador, é o analisador sintático. A função principal deste módulo é a de promover a análise da seqüência com que os átomos componentes do texto-fonte se apresentam, a partir da qual efetua a síntese da árvore da sintaxe do mesmo, com base na gramática da linguagem fonte (José, 1987).

A análise sintática cuida exclusivamente da forma das sentenças da linguagem, e procura, com base na gramática, levantar a estrutura das mesmas. Como centralizador das atividades da compilação, o analisador sintático opera, em compiladores dirigidos por sintaxe, como elemento de comando da ativação dos demais módulos do compilador, efetuando decisões acerca de qual módulo deve ser ativado em cada situação da análise do texto-fonte.

A análise sintática segundo José (1987) engloba as seguintes funções principais:

a) identificação de sentenças;
b) detecção de erros de sintaxe;
c) recuperação de erros;
d) correção de erros;
e) montagem da árvore abstrata da sentença;
f) comando da ativação do analisador léxico;
g) comando do modo de operação do analisador léxico;
h) ativação de rotinas da análise referente às dependências de contexto da linguagem;
i) ativação de rotinas da análise semântica;
j) ativação de rotinas de síntese do código objeto.

Um meio comumente usado para representar todas as derivações que indicam a mesma estrutura são as árvores de derivação, ou árvores sintáticas. Segundo José (1987), ao menos conceitualmente, o analisador sintático deve, com base na gramática, levantar, para a cadeia de entrada, a seqüência de derivação da mesma, ou seja, construir a árvore abstrata da sintaxe da sentença.

Em compiladores onde os componentes básicos não se apresentam com formato uniforme em todo o texto do programa, exigindo regras diferentes para sua identificação, cabe em geral ao analisador sintático decidir sobre o modo de operação da analisador léxico, de modo que, conforme o contexto, a rotina adequada de extração de componentes básicos seja utilizada.

O analisador sintático encarrega-se também da ativação de rotinas externas para verificação do escopo das variáveis, da coerência de tipos de dados em expressões, do relacionamento entre as declarações e os comandos executáveis, e outras verificações semelhantes.

Ainda de responsabilidade do analisador sintático, faz parte a ativação de rotinas de síntese do código objeto. Estas rotinas encarregam-se de produzir o código objeto correspondente ao texto-fonte.

A maioria das linguagens podem ter suas sintaxes descritas através de gramáticas livres de contexto.

2.1.2.1 GRAMÁTICAS LIVRES DE CONTEXTO

Segundo José (1987), gramáticas livres de contexto são aquelas em que é levantado o condicionamento das substituições impostas pelas regras definidas pelas produções. Este condicionamento é eliminado impondo às produções uma restrição adicional, que restringe as produções à forma geral $A ::= a$, onde $A \in N$, $a \in V^*$, ou seja, N sendo o conjunto de não terminais e V^* o vocabulário da gramática em questão. O lado esquerdo da produção é um não-terminal isolado e *a* é a cadeia pela qual *A* deve ser substituído ao ser aplicada esta regra de substituição, independente do contexto em que *A* está imerso. Daí o nome "livre de contexto" aplicado às gramáticas que obedecem a esta restrição.

Grande parte das construções de linguagens de programação possuem uma estrutura recursiva que pode ser representada por gramáticas livres de contexto (Aho, 1995). Para exemplificar, será utilizado o enunciado condicional definido pela seguinte regra: se *S1* e *S2* são enunciados e E é uma expressão, então "if *E* then *S1* else *S2*" é um enunciado.

Usando-se a variável sintática *cmd* para denotar a classe de comandos e *expr* para a classe de expressões, pode-se representar o enunciado condicional usando a produção gramatical descrita no quadro 1.

QUADRO 1 – PRODUÇÃO REPRESENTANDO ENUNCIADO CONDICIONAL

cmd ::= **if** *expr* **then** *cmd* **else** *cmd*

Fonte: José (1987).

Uma gramática livre de contexto é formada por *G=(T, N, S, P)* onde:

a) *T* representa os terminais que são os símbolos básicos a partir dos quais as cadeias são formadas, tais como as palavras-chave *if, then, else* da produção gramatical do quadro 1;
b) *N* são os não-terminais que constituem as variáveis sintáticas que denotam cadeias de caracteres, tais como o *cmd* e *expr*, além de impor uma estrutura hierárquica na linguagem que é útil tanto para a análise sintática quanto para a tradução;

c) *S* é o símbolo de partida, onde um não terminal é distinguido dos demais e o conjunto de cadeias que o mesmo denota é a linguagem definida pela gramática;
d) *P* são as produções de uma gramática, responsável em especificar a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Cada produção consiste em um não terminal, seguido por uma seta ou pelo símbolo "::=", seguido por uma cadeia de não terminais e terminais.

Segundo Aho (1995), cada método de análise pode tratar de gramáticas que tenham uma certa conformação. A gramática inicial pode ter que ser reescrita a fim de se tornar analisável pelo método escolhido. Por exemplo, os métodos de análise *top-down* não podem processar recursividade à esquerda. A gramática deverá passar por uma transformação para eliminar a recursão à esquerda. Outras transformações da gramática podem ser necessárias para deixar a gramática analisável pelo método *top-down.*

O método de análise *top-down* será utilizado na especificação do trabalho, visto que o mesmo já foi usado em versões anteriores. Ainda, a especificação descrita em André (2000), será estendida para suportar o uso de mapeamentos finitos. Para tanto, especificações serão incluídas para dar suporte a definição de estruturas e comando para fazer o manuseio de mapeamentos finitos. O código de máquina para as referidas construções também será implementado.

2.1.2.2 ANÁLISE SINTÁTICA *TOP-DOWN*

A análise sintática *top-down* pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada. Equivalentemente, pode ser vista como uma tentativa de se construir uma árvore gramatical, para a cadeia de entrada, a partir da raíz, criando os nós da árvore gramatical em pré-ordem (Aho, 1995).

Para se construir um analisador sintático preditivo (analisador sintático de descendência recursiva que não necessita de retrocesso), precisa-se conhecer, dado o símbolo corrente de entrada *a* e o não-terminal *A* a ser expandido, qual das alternativas da produção *A* ::= *a1*|*a2*|...|*an* é a única que deriva uma cadeia começando por *a*. Ou seja, a alternativa adequada precisa ser detectável examinando-se apenas o primeiro símbolo da cadeia que a mesma deriva. As construções de controle de fluxo na maioria das linguagens de

programação, como suas palavras-chave distintas, são usualmente detectáveis dessa forma. Por exemplo as construções do quadro 2, as palavras-chave *if*, *while* e *begin* informam qual alternativa é a única que possivelmente teria sucesso, caso queira-se encontrar um comando (Aho, 1995).

QUADRO 2 – UMA CONSTRUÇÃO DE CONTROLE DE FLUXO

cmd ::= **if** *expr* **then** *cmd* **else** *cmd*
| **while** *expr* **do** *cmd*
| **begin** *lista_de_comandos* **end**

Fonte: Aho (1995).

É possível um analisador gramatical descendente recursivo rodar para sempre. O problema emerge em produções recursivas à esquerda, tais como mostrado no quadro 3, onde, o símbolo mais à esquerda do lado direito é o mesmo que o não-terminal do lado esquerdo da produção. O lado direito da produção começa com *expr*, de tal forma que o procedimento *expr* é chamado recursivamente e o analisador roda para sempre.

QUADRO 3 – EXEMPLO DE RECURSÃO À ESQUERDA

expr ::= *expr* + *termo*

Fonte: Aho (1995).

Para eliminar situações de conflito (como por exemplo a recursão à esquerda), existem procedimentos específicos, os quais são descritos a seguir.

2.1.2.3 FATORAÇÃO À ESQUERDA

A fatoração à esquerda é uma transformação gramatical útil para a criação de uma gramática adequada à análise sintática preditiva. A idéia básica está em, quando não estiver claro qual das produções alternativas usar para expandir um não-terminal *A*, reescreve-se as produções *A* e posterga-se a decisão até que se tenha visto o suficiente da entrada para realizar a escolha certa. Por exemplo, a construção do quadro 4, ao enxergar o *token* de entrada **if**, não se pode imediatamente dizer qual produção escolher a fim de expandir *cmd*.

QUADRO 4 – PRODUÇÃO GRAMATICAL DE CONTROLE DE FLUXO

cmd ::= **if** *expr* **then** *cmd* **else** *cmd* | **if** *expr* **then** *cmd*

Fonte: Aho (1995).

Em geral, se $A ::= \alpha\beta_1|\alpha\beta_2$ são duas produções *A*, e a entrada começa por uma cadeia não vazia derivada a partir de α, não se sabe se vai expandir *A* em $\alpha\beta_1$ ou em $\alpha\beta_2$. Entretanto, pode-se postergar a decisão expandindo *A* para $\alpha A'$. Então, após enxergar a entrada derivada a partir de α, expandir A' em β_1 ou em β_2. Isto é, as produções originais, fatoradas à esquerda tornam-se como mostrado no quadro 5.

QUADRO 5 – EXEMPLO DE FATORAÇÃO À ESQUERDA

$A ::= \alpha\beta_1 \mid \alpha\beta_2$	não fatorado
$A ::= \alpha A`$ $A' ::= \beta_1\|\beta_2$	Fatorado

Fonte: Aho (1995).

2.1.2.4 ELIMINANDO A RECURSÃO À ESQUERDA

Uma gramática é recursiva à esquerda se possui um não-teminal *A* tal que exista uma derivação $A ::= A\alpha$ para alguma cadeia *a*. Os métodos de análise sintática *top-down* não podem processar gramáticas recursivas à esquerda e, consequentemente, uma transformação que elimine a recursão à esquerda é necessária.

A produção recursiva mostrada no quadro 6 pode ser substituída pelas construções não-recursivas mostradas no quadro 7, sem mudar o conjunto de cadeias de caracteres deriváveis a partir *A*. Esta regra por si mesma é suficiente para muitas gramáticas.

QUADRO 6 – PRODUÇÃO RECURSIVA À ESQUERDA

$A ::= A\alpha\|\beta$

Fonte: Aho (1995).

QUADRO 7 – PRODUÇÕES NÃO-RECURSIVAS

$$A ::= \beta A' ,$$

$$A' ::= \alpha A' \mid \wedge$$

Fonte: Aho (1995).

Não importa quantas produções-*A* existam, pode-se eliminar a recursão imediata das mesmas: primeiro agrupa-se as produções-*A* como no quadro 8, onde nenhum β_i começa por um *A*; em seguida, substitui-se as produções-*A* conforme mostrado no quadro 9.

QUADRO 8 – PRODUÇÕES AGRUPADAS

$$A ::= A\alpha_1 | A\alpha_2 | ... | A\alpha_m | \beta_1 | \beta_2 | ... | \beta_n$$

Fonte: Aho (1995).

QUADRO 9 – PRODUÇÕES SUBSTITUÍDAS

$$A ::= \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$$

$$A' ::= \alpha 1 A' \mid \alpha 2 A' \mid ... \mid \alpha_m A' \mid \wedge$$

Fonte: Aho (1995).

2.1.3 ANÁLISE SEMÂNTICA

A terceira grande tarefa do compilador refere-se à tradução propriamente dita do programa-fonte para a forma do código-objeto. Segundo José (1987), a geração do código vem acompanhada das atividades de análise semântica, responsáveis pela captação do sentido do texto-fonte, operação essencial à realização da tradução do mesmo, por parte das rotinas de geração de código.

Não é uma tarefa simples descrever completamente uma linguagem de tal modo que tanto sua sintaxe como sua semântica sejam descritas de maneira completa e precisa. As atividades de tradução, exercidas pelos compiladores, baseiam-se fundamentalmente em uma perfeita compreensão da semântica de linguagem a ser compilada, uma vez que é disto que

depende a criação das rotinas de geração de código, responsáveis pela obtenção do código-objeto a partir do programa-fonte (José, 1987).

Algumas das funções das ações semânticas do compilador segundo José (1987) são:

a) criação e manutenção de tabelas de símbolos;
b) associar aos elementos da tabela de símbolos seus respectivos atributos;
c) manter informações sobre o escopo dos identificadores;
d) representar tipos de dados;
e) analisar restrições quanto a utilização dos identificadores;
f) verificar o escopo dos identificadores;
g) verificar a compatibilidade de tipos;
h) efetuar a tradução do programa;
i) geração de código.

Existem notações para associar regras semânticas às produções, tais como definições dirigidas pela sintaxe e esquemas de tradução. Mais informações sobre esquemas de tradução podem ser encontradas em Aho (1995). Neste trabalho serão utilizadas definições dirigidas por sintaxe.

2.1.3.1 DEFINIÇÕES DIRIGIDAS POR SINTAXE

Uma definição dirigida pela sintaxe é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto de atributos particionados em dois subconjuntos, chamados de atributos sintetizados e atributos herdados daquele símbolo gramatical (Aho, 1995).

Um atributo pode representar qualquer coisa que se escolha: uma cadeia, um número, um tipo, uma localização de memória etc. O valor para um atributo em um nó da árvore é definido por uma regra semântica associada à produção usada naquele nó (Aho, 1995).

2.1.3.1.1 GRAMÁTICA DE ATRIBUTOS

Uma gramática de atributos é uma definição dirigida pela sintaxe, na qual as funções nas regras semânticas não têm efeitos colaterais, ou seja, estas funções não alteram seus parâmetros ou variável não local (Knuth, 1968).

Conforme descrito em Aho (1995), numa gramática de atributos, um atributo pode ser sintetizado ou herdado.

2.1.3.1.1.1 ATRIBUTOS SINTETIZADOS

Um atributo é dito sintetizado se seu valor em um nó da árvore gramatical é determinado a partir dos valores dos atributos dos filhos daquele nó. Atributos sintetizados são usados extensivamente na prática. Uma definição dirigida pela sintaxe que use exclusivamente atributos sintetizados é dita uma *definição S-atribuída* (Aho, 1995).

Por exemplo a definição *S-atribuída* no quadro 10, especifica uma calculadora de mesa que lê uma linha de entrada, contendo uma expressão aritmética, envolvendo dígitos, parênteses, operadores + e * e um caractere de avanço de linha **n** ao fim, e imprime o valor da expressão.

QUADRO 10 – DEFINIÇÃO DE UMA CALCULADORA DE MESA SIMPLES.

Produção	**Regras Semânticas**
L ::= E **n**	*Imprimir(E. val)*
$E ::= E_1+T$	$E.val := E_1.val + T.val$
E ::= T	*E.val := T.val*
$T ::= T_1 * F$	$T.val := T_1.val \times F.val$
T ::= F	*T.val := F.val*
F ::= (E)	*F.val := E.val*
F ::= **dígito**	*F.val :=* **dígito**.*lexval*

Fonte: Aho (1995).

2.1.3.1.1.2 ATRIBUTOS HERDADOS

Um atributo herdado, segundo Aho (1995), é aquele cujo valor a um nó de uma árvore gramatical é definido em termos do pai e/ou irmãos daquele nó.

Os atributos herdados são convenientes para expressar a dependência de uma construção de linguagem de programação no contexto em que a mesma figurar. Por exemplo, uma declaração gerada pelo não-terminal *D* numa definição dirigida pela sintaxe como mostrado no quadro 11, consiste na palavra-chave *int* ou *real*, seguida por uma lista de identificadores.

QUADRO 11 – DEFINIÇÃO TENDO *L.IN* COMO ATRIBUTO HERDADO.

Produção	**Regras Semânticas**
D ::= ***TL***	*L.in : = T.tipo*
T ::= **int**	*T.tipo : = inteiro*
T ::= **real**	*T.tipo : = real*
L ::= L_1**,id**	L_1*.in : = L.in*
	incluir_tipo(**id**.*entrada, L.in)*
L ::= **id**	*Incluir_tipo*(**id**.*entrada, L.in)*

Fonte: Aho (1995).

O não-terminal *T* possui um atributo sintetizado *tipo*, cujo valor é determinado pela palavra-chave na declaração. A regra semântica *L.in := T.tipo*, associada à produção *D ::= TL*, faz o atributo herdado *L.in* igual ao tipo na declaração.

As regras, então, propagam esse tipo pela árvore gramatical abaixo, usando o atributo herdado *L.in*. As regras associadas às produções para *L* chamam o procedimento *incluir_tipo* para incluir o tipo de cada identificador na sua entrada respectiva na tabela símbolos.

2.1.3.1.1.3 ELIMINAÇÃO DE RECURSÃO À ESQUERDA DE UM ESQUEMA DE TRADUÇÃO

Uma vez que a maioria dos operadores aritméticos é associativa à esquerda, é natural usar de gramáticas recursivas à esquerda para expressões. Estendem-se o algoritmo para eliminação de recursão à esquerda, como visto na seção 2.1.2.4, de forma a permitir atributos quando a gramática subjacente de um esquema de tradução for transformada. A transformação aplica-se a esquemas de tradução com atributos sintetizados. Permite que muitas definições dirigidas pela sintaxe sejam implementadas utilizando a análise sintática preditiva. O exemplo apresentado no quadro 12 motiva a transformação.

QUADRO 12 - ESQUEMA DE TRADUÇÃO COM UMA GRAMÁTICA RECURSIVA À ESQUERDA

$E \rightarrow E_1 + T$		$\{E.val := E_1.val + T.val\}$
$E \rightarrow E_1 - T$		$\{E.val := E_1.val + T.val\}$
$E \rightarrow$	T	$\{E.val := T.val\}$
$T \rightarrow$	(E)	$\{T.val := E.val\}$
$T \rightarrow$	***num***	$\{T.val:=$ ***num***$.val\}$

Fonte: Aho (1995).

Para análise sintática *top-down*, pode-se assumir que uma ação seja executada no tempo em que um símbolo na mesma posição é expandido. Por conseguinte, na segunda produção do quadro 14 a primeira ação (atribuição a R.i) é realizada após T ter sido completamente expandido em terminais e a segunda após R_1 ter sido completamente expandido.

Para adaptar a análise sintática preditiva a outros esquemas de tradução recursivos à esquerda, expressam-se o uso de atributos *R.i* e *R.s* do quadro 14 mais abstratamente. Supondo ter o esquema de tradução do quadro 13, cada símbolo gramatical possui um atributo sintetizado, escrito usando a letra minúscula correspondente, e *f* e *g* são funções arbitrárias.

QUADRO 13 - ESQUEMA DE TRADUÇÃO GENÉRICO

$A \rightarrow A_1 Y$	$\{A.a := g(A_1.a_1.Y.y\}$
$A \rightarrow X$	$\{A.a := f(X.x\}$

Fonte: Aho (1995).

A generalização para produções–*A* adicionais e para produções com cadeias em lugar dos símbolos *X* e *Y* pode ser feita como no quadro 16.

QUADRO 14 - ESQUEMA DE TRADUÇÃO TRANSFORMADO COM GRAMÁTICA RECURSIVA À DIREITA

$E \rightarrow$	T	$\{R.i := T.val\}$
	R	$\{E.val := R.s\}$
$R \rightarrow$	$+$	
	T	$\{R_1.i := R.i + T.val\}$
	R_1	$\{R.s := R_1.s\}$
$R \rightarrow$	$-$	
	T	$\{R_1.i := R.i - T.val\}$
	R_1	$\{R.s := R_1.s\}$
$R \rightarrow$	$\wedge$	$\{R.s := R.i\}$
$T \rightarrow$	$($	
	E	
	$)$	$\{T.val := E.val\}$
$T \rightarrow$	**num**	$\{T.val :=$ **num**$.val\}$

Fonte: Aho (1995).

O algoritmo para eliminar recursão à esquerda na seção 2.1.2.4 constrói a gramática descrita no quadro 15 a partir do quadro 13.

QUADRO 15 – GRAMÁTICA CONSTRUÍDA A PARTIR DO QUADRO 13

$A \rightarrow$	$X\ R$
$R \rightarrow$	$Y\ R \mid \wedge$

Fonte: Aho (1995).

Levando em conta as ações semânticas, o esquema transformado se torna como mostrado no quadro 16.

QUADRO 16 - ESQUEMA TRASFORMADO COM AÇÕES SEMÂNTICAS

$A \rightarrow$	X	$\{R.i := f(X.x)\}$
	R	$\{A.a := R.s\}$
$R \rightarrow$	Y	$\{R_1.i := g(R.i,Y.y)\}$
	R_1	$\{R.s := R_1.s\}$
$R \rightarrow$	$\wedge$	$\{R.s := R.i\}$

Fonte: Aho (1995).

2.2 MAPEAMENTO FINITO (*ARRAY'S*)

Um mapeamento finito também é conhecido como correspondência finita. Nas linguagens de programação normalmente conhecido como definições de matrizes (quando de uma dimensão, normalmente é chamada de vetor). Na linguagem de programação Pascal é conhecido como *array*. Um mapeamento finito (*array*) é um agregado homogêneo de elementos de dados cujo elemento individual é identificado por sua posição no agregado em relação ao primeiro. Uma referência a um elemento *array* em um programa freqüentemente inclui um ou mais subscritos (também conhecido como índice) não constantes (Sebesta, 2000).

2.2.1 *ARRAY'S* E ÍNDICES

Elementos de um *array* são referenciados por meio de um mecanismo sintático de dois níveis, cuja a primeira parte é o nome do agregado e a segunda é um seletor que consiste em um ou mais itens conhecidos como subscritos ou índices. Se todos os índices em uma referência forem constantes, o seletor será estático, caso contrário dinâmico. A operação de seleção pode ser imaginada como uma correspondência do nome do *array* e o conjunto de valores de índice com um elemento do agregado. De fato, os *array's*, as vezes são chamados de correspondência finita. Simbolicamente, esta pode ser mostrada como no quadro 17.

QUADRO 17 – EXEMPLO SIMBÓLICO DE *ARRAY*

nome_do_array(lista_de_valores_indice) → elemento

Fonte: Sebesta (2000).

A sintaxe das referências a *array's* é mais ou menos universal: ao nome do *array* segue-se o da lista de índices, colocada entre parênteses ou entre colchetes. Um problema com os parênteses é que eles freqüentemente são usados para conter os parâmetros de chamadas a subprogramas, isso faz as referências a *array's* parecerem-se exatamente como essas chamadas.

Dois tipos distintos estão envolvidos em um tipo de *array*: o do elemento e o dos subscritos. Este último freqüentemente é uma subfaixa de números inteiros, mas o Pascal, o Modula-2 e a Ada permitem que alguns outros tipos sejam usados como subscritos, como por

exemplo, booleano, caracter e enumeração. Assim exigir a sua verificação é um fator importante na confiabilidade das linguagens. Entre as linguagens contemporâneas, o C, o C++ e o FORTRAN não especificam a verificação de subscritos, mas o Pascal, a Ada e o Java especificam.

2.2.2 CATEGORIAS DE *ARRAY*

Segundo Sebesta (2000) os *array's* ocorrem em quatro categorias:

a) um *array* estático é aquele em que as faixas de subscrito estão estaticamente vinculadas e a alocação de armazenamento é estática (feita antes da execução). A vantagem dos *array's* estáticos é a eficiência. Nenhuma alocação dinâmica é exigida;

b) um *array* stack-dinâmico fixo (semi-estático) é aquele que as faixas de subscrito estão estaticamente vinculadas, mas a alocação é feita no momento de elaboração da declaração durante a execução. A sua vantagem sobre os *array's* estáticos é a eficiência de espaço. Um *array* grande em um procedimento pode usar o mesmo espaço que um grande em um procedimento diferente, contanto que ambos os procedimentos não estejam ativos ao mesmo tempo;

c) um *array stack*-dinâmicos (semi-dinâmico) é aquele que as faixas de subscrito estão dinamicamente vinculadas e a alocação de armazenamento é dinâmica (feita durante a execução). Porém, assim que as faixas de subscrito são vinculadas e o armazenamento é alocado, eles permanecem fixos durante o tempo de vida da variável. A vantagem sobre os estáticos e sobre os *stack*-dinâmicos fixos é a flexibilidade. O tamanho de um *array* não precisa ser conhecido até que ele esteja prestes a ser usado;

d) um *array heap*-dinâmico (dinâmico) é aquele em que a vinculação das faixas de subscrito e de alocação de armazenamento é dinâmica e pode mudar qualquer número de vezes durante o seu tempo de vida. A sua vantagem sobre os outros é a flexibilidade: os *array's* podem crescer e reduzir-se durante a execução do programa conforme mudar a necessidade de espaço.

2.2.2.1 EXEMPLOS DE UTILIZAÇÃO DE *ARRAY'S* NAS LINGUAGENS DE PROGRAMAÇÃO

No FORTRAM 77, o tipo subscrito é vinculado a um *array* no tempo de projeto da linguagem; todos os subscritos são do tipo inteiro. As faixas de subscrito são estaticamente vinculadas e todo o armazenamento é declarado estaticamente, portanto, os *array's* do FORTRAN 77 são estáticos.

Os *array's* ADA podem ser *stack*-dinâmicos, como mostrado no quadro 18.

QUADRO 18 – EXEMPLO DE ARRAY *STACK*-DINÂMICO

```
GET(LIST_LEN);
Declare
   LIST : array (1..LIST_LEN) of  INTEGER;
Begin
. . .
end;
```

Fonte: Sebesta (2000).

No quadro 18, o usuário introduz o número de elementos na lista de *array's*, os quais são, então, dinamicamente alocados quando a execução atinge o bloco **declare**.

O FORTRAN 90 oferece *array's* dinâmicos. Eles podem ser alocados e desalocados mediante solicitação. Suas faixas de subscrito podem ser mudadas por qualquer processo de salvar, de desalocar ou de realocar.

Por exemplo, no FORTRAN 90, declara-se um *array* para que seja dinâmico, como mostrado no quadro 19, o qual declara-se que MAT é uma matriz de elementos do tipo INTEGER que pode ser dinamicamente alocada e cuja especificação é feita com uma instrução ALLOCATABLE, como no quadro 20.

QUADRO 19 – DECLARAÇÃO DE *ARRAY* DINÂMICO

```
INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT
```

Fonte: Sebesta (2000).

QUADRO 20 – ALOCAÇÃO DE *ARRAY* DINÂMICO

```
ALLOCATABLE ( MAT ( 10, NUMERO_DE_COLS ) )
```

Fonte: Sebesta (2000).

As faixas de subscrito podem ser especificadas por variáveis de programa, bem como por literais. Os limites inferiores das faixas de subscrito têm como padrão 1.

Array's dinâmicos podem ser destruídos pela instrução DEALLOCATABLE, como no quadro 21.

QUADRO 21 – DESALOCAÇÃO DE *ARRAY* DINÂMICO

DEALLOCATABLE (MAT)

Fonte: Sebesta (2000).

Para tornar um *array* dinâmico maior ou menor, seus elementos devem ser salvos temporariamente em outro *array*, e ele deve ser desalocado e, depois, realocado no novo tamanho.

O C e o C++ também oferecem *array's* dinâmicos. As funções de biblioteca padrão, *malloc* e *free*, ambas operações de alocação e desalocação gerais do *heap*, respectivamente, podem ser usadas por *array's* C. O C++ usa os operadores **new** e **delete** para gerenciar o armazenamento do *heap*. Uma vez que não há nenhuma verificação de índices no C e no C++, o tamanho de um *array* não interessa ao sistema em tempo de execução, assim, estender ou encolher um *array* é fácil. Eles são tratados como ponteiros em algumas coleções de céluas de armazenagem em que o ponteiro pode ser indexado.

Segundo Sebesta (2000) a linguagem de programação Perl tem outro tipo de *array* dinâmico. Eles crescem implicitamente quando feitas atribuições além do último atual. Pode-se fazer com que eles encolham atribuindo um agregado vazio, especificado com '(' e ')'.

Na versão original do Pascal, a faixa ou as faixas de índice de um *array* faziam parte de seu tipo. Isso juntamente com o uso da equivalência de nomes para garantir a compatibilidade, desaprovava a existência de um subprograma que processava *array's* de tamanhos diferentes. Um procedimento que classificava-os como inteiros, por exemplo, somente podia ser escrito para aqueles com uma única faixa de subscrito fixa. O ISO *Standard Pascal* (ISO, 1982) oferece uma saída para o problema: parâmetros formais que incluem a definição de tipo do *array*, conforme o quadro 22.

QUADRO 22 – EXEMPLO DE *ARRAY'S* NO PASCAL

```
Procedure  sumlist  ( var soma : integer;
                              Lista array  [ inferior .. superior  :
                              Integer ] of integer );
   Var indice  :  integer;
   Begin
   Soma  :=  0;
   For indice  :=  inferior  to  superior  do
      soma  :=  soma  +  lista [ indice ]
   End;
```

Fonte: Sebesta (2000).

Um exemplo de chamada para esse procedimento poderia ser como mostrado no quadro 23.

QUADRO 23 – EXEMPLO DE CHAMADA DE *ARRAY'S* NO PASCAL

```
   Var escores :  array [ 1..100 ]  of  integer;
   . . .
   sumlist ( soma, escores );
```

Fonte: Sebesta (2000).

2.2.3 NÚMERO DE SUBSCRITOS EM *ARRAY'S*

O FORTRAN I limitou a três o número de subscritos de um *array*, porque na época da execução do projeto, a eficiência era uma preocupação fundamental. Os projetistas do FORTRAN I desenvolveram um método muito rápido para acessar elementos de *array* de até três dimensões, mas não mais do que três. Do FORTRAN IV em diante, permitiu-se que o número de dimensões chegasse a sete, mas a maioria das linguagens contemporâneas não impõe esse limite. Não existe nenhuma justificativa para a limitação do FORTRAN. Um programador que deseje usar uma variável com 10 dimensões e esteja disposto a pagar pelo custo das referências aos elementos desse *array* deve ter permissão para fazer.

Os *array's* em C podem ter somente um subscrito, mas estes podem ter outros como elementos, suportando assim, os multidimensionais. Conforme descrito no quadro 24, ele cria uma variável inteira, **mat**, que é um *array* de cinco elementos sendo que, cada um dos quais é

outro *array* de quatro elementos. A diferença entre ele e uma matriz em outra linguagem, diga-se no FORTRAN, é mínima. O usuário quase sempre pode ignorar o fato de que **mat** não é uma matriz de fato, exceto que a sintaxe para referências exige um conjunto de colchetes para cada subscrito.

QUADRO 24 – DECLARAÇÃO DE ARRAY EM LINGUAGEM C

```
Int  mat [5] [4] ;
```

Fonte: Sebesta (2000).

2.2.4 INICIALIZAÇÃO DE *ARRAY'S*

Algumas linguagens fornecem o meio de inicializar *array's* no momento em que o armazenamento é alocado. No FORTRAN 77, todo o armazenamento de dados é alocado estaticamente, assim, a inicialização no momento de carregar (*load-time*) usando-se a instrução DATA é permitida. Conforme o quadro 25 LISTA é inicializada para os valores da lista delimitados por barras diagonais.

QUADRO 25 – EXEMPLO DE INICIALIZAÇÃO DE *ARRAY'S* NO FORTRAN 77

```
INTEGER LISTA (3)
DATA LISTA / 0, 5, 5 /
```

Fonte: Sebesta (2000).

O ANSI C e o C++ também permitem inicialização de seus *array's*, mas com uma nova mudança na declaração, conforme descrita no quadro 26. O compilador define o tamanho do *array*. Essa pretende ser uma conveniência, mas tem seu custo. Ela efetivamente remove a possibilidade do sistema poder detectar alguns erros do programador, como, por exemplo, deixar erroneamente uma valor fora da lista.

QUADRO 26 – EXEMPLO DE INICIALIZAÇÃO DE *ARRAY'S* NA LINGUAGEM C

```
Int  lista  [ ]  =  { 4,  5,  7,  83 };
```

Fonte: Sebesta (2000).

As *strings* de caracteres no C e no C++ são implementadas como *array's* de **char**. Estes, por sua vez, podem ser inicializados para constantes de *string*. Conforme apresentado no quadro 27, o *array* terá oito elementos, porque todas as *strings* são encerradas com um

caractere nulo (zero), o qual é implicitamente suportado pelo sistema para constantes de *string*.

QUADRO 27 – *ARRAY'S* DE CHAR NA LINGUAGEM C

```
char nome [ ] = "freddie";
```

Fonte: Sebesta (2000).

Array's de strings no C e no C++ também podem ser inicializados com literais de *strings*. Nesse caso, o *array* é de ponteiros para caracteres. O quadro 28 ilustra a natureza dos literais de caracteres em C e em C++. No quadro 27, de literal de *string* usada para inicializar o *array* **char** nome, a literal é considerada um **char**. Mas no quadro 28, as literais são consideradas ponteiros para os caracteres, de modo que o *array* é de ponteiros para caracteres.

QUADRO 28 – *ARRAY'S* DE STRINGS NA LINGUAGEM C

```
char *nomes = {"Bob", "Jake", "Darcie"}
```

Fonte: Sebesta (2000).

A Ada fornece dois mecanismos para inicializar *array's* na instrução de declaração: listando-se na ordem em que são armazenados, ou atribuindo-se diretamente a uma posição do índice usando o operador =>, que é chamado de seta na Ada. Considerando o quadro 29, na primeira instrução, todos os elementos de LISTA têm valores de inicialização, os quais são atribuídos às localizações do elemento de *array* na ordem em que eles aparecem. Na segunda, o primeiro e o terceiro elemento deste são inicializados, usando-se a atribuição direta e a cláusula **others** é usada para inicializar os elementos restantes. Essas coleções de valores, definidas pelos parênteses, são chamadas de **valores agregados**.

QUADRO 29 – DECLARAÇÃO E INICIALIZAÇÃO DE *ARRAY'S* NA ADA

```
LISTA : array ( 1..5 ) of INTEGER := ( 1, 3, 5, 7, 9):
GRUPO : array ( 1..5 ) of INTEGER := ( 1 => 3, 3 => 4, others => 0):
```

Fonte: Sebesta (2000).

2.2.5 IMPLEMENTAÇÃO DO TIPO *ARRAY*

Os elementos de um *array* podem receber acesso mais rapidamente se os elementos forem armazenados num bloco de localizações consecutivas. Se a largura de cada elemento do

array é w, o *i*ésimo elemento do *array* A começa na localização calculada conforme mostrada no quadro 30.

QUADRO 30 – LOCALIZAÇÃO DE *ARRAY'S*

$base + (i - linf) \times w$

Fonte: Aho (1995).

O limite *linf* é o limite inferior do intervalo de subscritos e *base* é o endereço relativo da memória alocada para o *array*. Isto é, base é o endereço relativo de A [linf].

A expressão do quadro 30 pode ser primeiramente avaliada em tempo de compilação se for reescrita como no quadro 31.

QUADRO 31 – LOCALIZAÇÃO DE *ARRAY'S* PARA AVALIAÇÃO EM TEMPO DE COMPILAÇÃO

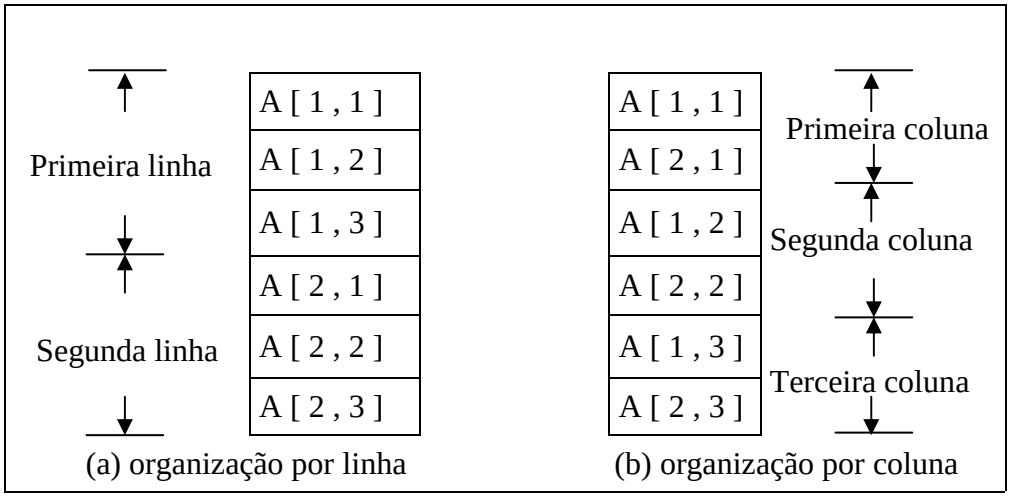
$i \times w + (base - linf \times w)$

Fonte: Aho (1995).

A subexpressão c = *base - linf* × *w* pode ser avaliada quando a declaração do *array* for enxergada. Assume-se que *c* é salvo na entrada da tabela de símbolos para A, de forma que o endereço relativo de A [i] é obtido simplesmente adicionando-se $i \times w$ a *c*.

A pré-computação em tempo de compilação também pode ser aplicada a cálculos de endereços de elementos de *array's* multidimensionais. Um *array* bidimensional é normalmente armazenado de uma ou duas formas, ou por *linha* (linha a linha) ou por *coluna* (coluna a coluna). O quadro 32 mostra a disposição de um *array* A 2×3 (a) organizado por linha, e (b) por colunas. Fortran usa a organização por coluna, Pascal por linha, porque A [i , j] é equivalente a A [i] [j] e os elementos do *array* A [i] são armazenados consecutivamente.

QUADRO 32 – ORGANIZAÇÃO DE *ARRAY'S*



Fonte: Aho (1995).

No caso de um *array* bidimensional armazenado na ordem por linha, o endereço relativo de A [i_1 , i_2] pode ser calculado pela fórmula descrita no quadro 33.

QUADRO 33 – FÓRMULA PARA *ARRAY* BIDIMENSIONAL

$$base + ((i_1 - linf_1) \times n_2 + i_2 - linf_2) \times w$$

Fonte: Aho (1995).

No quadro 33, *linf1* e *linf2* são os limites inferiores sob os valores de i_1 e i_2 respectivamente, e n_2 é o número de valores que i_2 pode assumir. Isto é, se $lsup_2$ é o limite superior sobre os valores de i_2, então $n_2 = lsup_2 + 1$. Assumindo que i_1 e i_2 são os únicos valores que não são conhecidos em tempo de compilação, pode-se reescrever o quadro 33 como no quadro 34.

QUADRO 34 – REESCREVENDO A FÓRMULA PARA *ARRAY* BIDIMENSIONAL

$$((i_1 \times n_2) + i_2) \times w + (base - ((linf_1) \times n_2) + linf_2) \times w)$$

Fonte: Aho (1995).

O último termo da expressão do quadro 34 pode ser determinado em tempo de compilação.

Pode-se generalizar a organização por linha ou por coluna para várias dimensões. A generalização da organização por linha significa armazenar os elementos de tal forma que, à medida que esquadrinha-se um bloco de memória, os subscritos mais à direita parecem variar

mais rapidamente, como os números num odômetro. A expressão do quadro 34 se generaliza para a expressão do quadro 35, para o endereço de A[i_1 , i_2 ,... , i_k].

QUADRO 35 - GENERALIZANDO A FÓRMULA PARA *ARRAY* DE *K* DIMENSÕES

$$((\ldots((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_K + i_K) \times w$$
$$+ base - ((\ldots((linf_1 n_2 + linf_2) n_3 + linf_3) \ldots) n_K + linf_K) \times w$$

Fonte: Aho (1995).

Como para todo j,$n_j = lsup_j - linf_j + 1$ é assumido fixo, o termo na segunda linha do quadro 35 pode ser computado pelo compilador e salvo na entrada da tabela de símbolos para A. A organização por coluna se generaliza num arranjo oposto, no qual os subscritos mais à esquerda variam mais rapidamente.

Algumas linguagens permitem que os comprimentos dos *array's* sejam especificados dinamicamente, quando o procedimento for chamado em tempo de execução. As fórmulas para o acesso aos elementos de tais *array's* são as mesmas que aquelas para *array's* de tamanho fixo, mas os limites superior e inferior não são conhecidos em tempo de compilação.

O problema principal na geração de código para referências a *array's* é o de relacionar o cômputo do quadro 35 a uma gramática para referenciar *array's*. As referências a *array's* poderm ser permitidas nas atribuições se o não terminal *L*, for permitido figurar onde o *id* aparecer. O quadro 36 tem o objetivo de tornar disponíveis os vários limites dimensionais n_j do *array*, à medida que agrupam-se expressões de índices numa *lista_E*.

QUADRO 36 – GRAMÁTICA PARA REFERENCIAR *ARRAY'S*

$$L \rightarrow lista\_E \mid \textbf{id}$$
$$Lista\_E \rightarrow lista\_E\ ,\ E \mid \textbf{id}\ [\ E$$

Fonte: Aho (1995).

O nome do *array* é atrelado à expressão de índice mais à esquerda, ao ínvés de ficar ligada à *lista_E* quando um *L* for formado. Essas produções permitem que um apontador para a entrada da tabela de símbolos para o nome do *array* seja passado como atributo sintetizado numa *lista_E*.

Também usa-se uma *lista_E.ndim* para registrar o número de dimensões (expressões de índices) na *lista_E*. A função *limite (array,j)* retorna n_j , o número de elementos ao longo

da *j_ésima* dimensão do *array*, cuja entrada da tabela de símbolos é apontada por *array*. Finalmente, *lista_E.local* denota um temporário que abriga um valor computado a partir de expressões de índice em *lista_E*.

Uma *lista_E* que produza os *m* primeiros índices de uma referência a um *array k*-dimensional A[i_1 i_2 ,..., i_k] irá gerar o código de endereços para computar a expressão do quadro 37, usando a fórmula de recorrência do quadro 38

QUADRO 37 – ÍNDICES A SEREM COMPUTADOS

$(\ldots \ ((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_m + i_m$

Fonte: Aho (1995).

QUADRO 38 – FÓRMULA DE RECORRÊNCIA

$e_1 = i_1$ $e_m = e_{m-1} \times n_m + i_m$

Fonte: Aho (1995).

Por conseguinte, quando $m = k$, uma multiplicação pela largura *w* é tudo o que será necessitado para computar o termo da primeira linha do quadro 35. Note-se que os *i's* aqui podem realmente ser valores de expressões e o código para avaliar aquelas expressões será entremeado com o código para computar o termo do quadro 37.

Um valor-*l L* terá dois atributos, *L_local* e *L_deslocamento.* No caso em que *L* for simplesmente um nome simples, *L_local* será um apontador para a entrada da tabela de símbolos para aquele nome, e *L_deslocamento* será **null**, indicando que o valor-*l* é um nome simples ao invés de uma referência a um *array*.

2.3 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Apesar de se poder traduzir o programa fonte diretamente na linguagem-alvo, existem alguns benefícios em se usar uma forma intermediária independente de máquina, descritos em Aho (1995) como segue:

a) a partir da representação intermediária pode-se gerar código para diferentes máquinas alvo, bastando apenas mudar o módulo de geração de código;
b) otimizações independentes de máquina podem ser aplicadas à representação intermediária.

O presente trabalho utilizará como base para geração de código intermediário a representação chamada de código de três endereços.

2.3.1 CÓDIGO DE TRÊS ENDEREÇOS

Código de três endereços é uma seqüência de enunciados como mostrado no quadro 39, onde *x*, *y* e *z* são nomes, constantes ou objetos de dados temporários criados pelo compilador e *op* está no lugar de qualquer operador, tal como um operador de aritmética de ponto fixo ou flutuante ou um operador lógico sobre dados *booleanos* (Aho, 1995).

Na construção de código de três endereços são permitidas expressões aritméticas construídas, na medida em que só há um operador no lado direito de um enunciado. Desta forma, uma expressão de linguagem-fonte como a descrita no quadro 39 poderia ser traduzida na seqüência mostrada no quadro 40, onde t_1 e t_2 são nomes temporários gerados pelo compilador.

QUADRO 39 – ENUNCIADOS DE TRÊS ENDEREÇOS

x := y *op* z

Fonte: Aho (1995).

QUADRO 40 – TRADUÇÃO DO QUADRO 39

t_1:= y * z
t_2:= x + t_1

Fonte: Aho (1995).

2.3.2 TRADUÇÃO DIRIGIDA PELA SINTAXE EM CÓDIGO DE TRÊS ENDEREÇOS

Quando o código de três endereços é gerado, os nomes temporários são construídos para os nós interiores da árvore sintática. O valor do não-terminal *E* ao lado esquerdo de *E := E_1 + E_2* (quadro 41) será computado numa nova variável temporária *t*. Em geral, o código de três endereços para *id := E* consiste em código para avaliar *E* em alguma variável temporária *t*, seguido pela atribuição *id.local := t*. Se uma expressão se constituir em um único identificador, como por exemplo *y*, o próprio *y* abrigará o valor da expressão.

QUADRO 41 – DEFINIÇÃO DIRIGIDA PELA SINTAXE

Produção	Regras Semânticas
S ::= id := E	*S.código* := *E.código* || *gerar(id.local ':=' E.local)*
E ::= E_1 + E_2	*E.local* := *novo_temporário;* *E.código* := *E_1.código* || *E_2.código* || *gerar(E.local ':=' E_1.local '+' E_2.local)*
E ::= E_1 * E_2	*E.local* := *novo_temporário;* *E.código* := *E_1.código* || *E_2.código* || *gerar(E.local ':=' '*' E_2.local)*
E ::= - E_1	*E.local* := *novo_temporário;* *E.código* := *E_1.código* || *gerar(E.local ':=' 'uminus' E_1.local)*
E ::= (E_1)	*E.local* := *E_1.local;* *E.código* := *E_1.código*
E ::= id	*E.local* := *id.local;* *E.código* := ''

Fonte: Aho (1995).

A definição do quadro 41 gera código de três endereços para enunciados de atribuição. A partir de uma entrada como por exemplo a expressão do quadro 42, a definição produz um o código apresentado no quadro 43. O atributo sintetizado *S.código* representa o código de três endereços para a atribuição *S*. O não-terminal *E* possui dois atributos: *E.local* e *E.código* onde o primeiro é o nome que irá abrigar o valor de *E* e o segundo é a seqüência de enunciados de três endereços avaliando *E*.

QUADRO 42 – ENUNCIADO DE ATRIBUIÇÃO EM LINGUAGEM-FONTE

```
a:= b * -c + b * -c
```

Fonte: Aho (1995).

QUADRO 43 – CÓDIGO GERADO A PARTIR DA EXPRESSÃO DO QUADRO 42

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

Fonte: Aho (1995).

2.4 DEFINIÇÃO DE ESCOPOS

Nesta seção será apresentado as técnicas utilizadas para implementação de escopos de declarações de variáveis e procedimentos e chamadas de procedimentos. Um escopo de uma declaração é definido em Aho (1995) como parte do programa à qual esta declaração se aplica. Uma ocorrência de um nome dentro de um procedimento é dita local ao mesmo se estiver no escopo de uma declaração dentro de um procedimento, caso contrário, a ocorrência é não local.

Cada procedimento possui uma tabela que conterá todas as variáveis e procedimentos declarados dentro deste procedimento. Para cada nome local é criada uma entrada na tabela de símbolos, com o tipo e o endereço relativo da memória para aquele nome. O endereço relativo consiste em um deslocamento a partir da base estática de dados ou do campo para os dados locais no registro de ativação. A seção 3.1 descreve como foi implementada a definição de escopo.

Um procedimento pode utilizar objetos definidos internamente no procedimento de programa ou definidos externamente, desde que sejam visíveis no procedimento. O bloco de um procedimento pode conter chamadas para outros procedimentos conforme descrito no quadro 65 (Seção 3.1). Em linguagens bloco estruturadas um procedimento pode conter chamadas para (Silva, 2000):

a) ela própria (chamada recursiva);
b) seus irmãos;
c) seus filhos;
d) seus ancestrais (pais, avós, ...);
e) irmãos de seus ancestrais.

2.5 GERAÇÃO DE CÓDIGO

Na geração de código os detalhes são dependentes da máquina-alvo e do sistema operacional. Assuntos como a gerência de memória, seleção de instruções, alocação de registradores e a ordem de avaliação são inerentes a quase todos os problemas de geração de código (Aho, 1995). Neste capítulo serão apresentados os temas genéricos de um projeto de gerador de código.

2.5.1 PROGRAMA-ALVO

O programa-alvo é o arquivo de saída do gerador de código, que pode assumir uma variedade de formas: linguagem absoluta de máquina; linguagem relocável de máquina ou linguagem de montagem (Aho, 1995).

A produção de um programa em linguagem absoluta de máquina possui a vantagem do mesmo poder ser carregado numa localização de memória e executado imediatamente.

A produção de um programa em linguagem relocável de máquina (módulo objeto) como saída permite que os subprogramas sejam compilados separadamente.

A produção de um programa em linguagem de montagem torna o processo de geração de código um tanto menos complexo. Pode-se gerar instruções simbólicas e usar as facilidades de processamento de macros do montador para auxiliar a geração de código.

Neste trabalho será executado a geração de código através de uma linguagem de montagem.

2.5.2 GERENCIAMENTO DE MEMÓRIA

O mapeamento dos nomes no programa-fonte para os endereços dos objetos de dados em tempo de execução é feito cooperativamente pelo analisador sintático e pelo gerador de código. O tipo numa declaração determina a largura, isto é, a quantidade de memória necessária para o nome declarado, por exemplo, a declaração no quadro 44, diz ao compilador que os nomes *I* e *J* são do tipo *integer* e necessitam da mesma quantidade de memória. A partir das informações na tabela de símbolos, pode-se determinar um endereço relativo para um nome na área de dados (Aho, 1995).

Para se gerar código de máquina, os rótulos nos enunciados de três endereços (quadro 39), devem ser convertidos para endereços de instruções. Suponha que os rótulos refiram-se a números de quádruplas[1] num *array*. A medida que esquadrinham-se cada quádrupla, pode-se

[1] quádrupla é uma estrutura de registro com quatro campos.

deduzir a localização da primeira instrução de máquina gerada para aquela quádrupla, mantendo uma contagem do número de palavras usadas para as instruções geradas até então. A contagem pode ser mantida no *array* de quádruplas (em um campo extra) de forma que se for encontrada uma referência tal como *j:goto i* e *i* for menor do que *j*, o número da quádrupla corrente, pode-se simplesmente gerar uma instrução de desvio com o endererço-alvo igual a localização de máquina da primeira instrução do código para a quádrupla *i*. Se, entretanto, o desvio é para adiante, e, dessa forma, *i* excede *j* precisa-se armazenar numa lista para quádrupla *i* a localização para a primeira instrução gerada para a quádrupla *j*, então, ao processar a quádrupla *i* preenche-se a localização adequada em todas as instruções que sejam desvio adiante para *i* (Aho, 1995).

QUADRO 44 – DECLARAÇÃO DE VARIÁVEIS

```
Var

I , J : integer;
```

Fonte: Aho (1995).

2.5.3 SELEÇÃO DE INSTRUÇÕES

Segundo Aho (1995), a natureza do conjunto de instruções da máquina-alvo determina a dificuldade da seleção de instruções. A uniformidade e completeza do conjunto de instruções são fatores importantes. Se a máquina alvo suporta cada tipo de dado de uma maneira uniforme, cada exceção à regra geral requer um tratamento especial.

A velocidade das instruções e os dialetos de máquina são fatores importantes. Se não importar a eficiência do programa-alvo, a seleção de instruções é um processo direto. Para cada tipo de instrução de três endereços, pode-se projetar um esqueleto de código que delineie o código alvo a ser gerado para aquela construção. Por exemplo, cada enunciado de três endereços da forma *x:=y+z*, onde *x*, *y* e *z* são alocados estaticamente, pode ser traduzido na seqüência de código mostrada no quadro 45.

Infelizmente, esse tipo de geração de código enunciado a enunciado freqüentemente produz um código de baixa qualidade. Por exemplo, a seqüência de enunciados apresentada no quadro 46, seria traduzida para o código mostrado no quadro 47.

QUADRO 45 – SEQÜÊNCIA DE CÓDIGO PARA ENUNCIADOS DE TRÊS ENDEREÇOS

MOV	R0,y	/* carregar y no registrador R0 */
ADD	R0,z	/* adicionar z a R0 */
MOV	x,R0	/* armazenar R0 em x */

Fonte: Adaptado de Aho (1995).

QUADRO 46 – SEQÜÊNCIA DE ENUNCIADOS DE TRÊS ENDEREÇOS

a := b+c
d := a+e

Fonte: Aho (1995).

No quadro 47, o quarto enunciado, que move o valor de *A* para o registrador *R0*, é redundante, porque o registrador já possui o valor de *A*, como será também o terceiro, caso *A* não venha a ser utilizado subseqüentemente.

QUADRO 47 – TRADUÇÃO DO QUADRO 46

MOV	R0,b
ADD	R0,c
MOV	a,R0
MOV	R0,a
ADD	R0,e
MOV	d,R0

Fonte: Adaptado de Aho (1995).

A qualidade do código gerado é determinado por sua velocidade e tamanho. Em uma máquina-alvo com um rico conjunto de instruções, pode-se providenciar várias formas de se implementar uma dada operação, uma vez que as diferenças entre as implementações podem ser significativas. Uma tradução do código intermediário pode levar a um código-alvo correto, porém ineficiente. Por exemplo, se a máquina-alvo possui uma instrução de incremento (*INC*), o enunciado de três endereços *a := a +1* pode ser implementado mais eficientemente pela instrução singela "*INC a*" do que por uma seqüência mais óbvia que carregue o *a* no registrador, adicione um ou mais e, em seguida, armazene o resultado de volta em *a* como mostrado no quadro 48.

QUADRO 48 – SEQÜÊNCIA DE CÓDIGO INEFICIENTE

```
MOV  R0,a
ADD  R0,#1
MOV  a,R0
```

Fonte: Adaptado de Aho (1995).

As velocidades das instruções são necessárias para se projetar boas seqüências de código, mas, infelizmente informações acuradas a respeito da cronometrização das instruções são difíceis de se obter. Decidir que seqüência de código de máquina é a melhor para uma dada construção de três endereços requer, também o conhecimento a respeito do contexto no qual a instrução aparece (Aho, 1995).

2.5.4 ALOCAÇÃO DE REGISTRADORES

As instruções envolvendo operadores do tipo registrador são usualmente mais curtas do que aquelas envolvendo operandos na memória. Por conseguinte, a utilização eficiente dos registradores é particularmente importante na geração de código de boa qualidade. O uso dos registradores descritos em Aho (1995), é freqüentemente subdividido em dois subproblemas:

a) durante a locação de registradores, seleciona-se o conjunto de variáveis que residirão nos registradores a um determinado ponto do programa;
b) durante a fase subseqüente de atribuição de registradores, obtêm-se o registrador específico no qual a variável irá residir.

Conforme descrito em Aho (1995), encontrar uma atribuição ótima para os registradores é difícil ainda que com valores únicos de registradores. O problema é adicionalmente complicado porque o *hardware* e/ou sistema operacional podem exigir que certas convenções de uso dos registradores sejam observadas.

2.5.5 MÁQUINA-ALVO

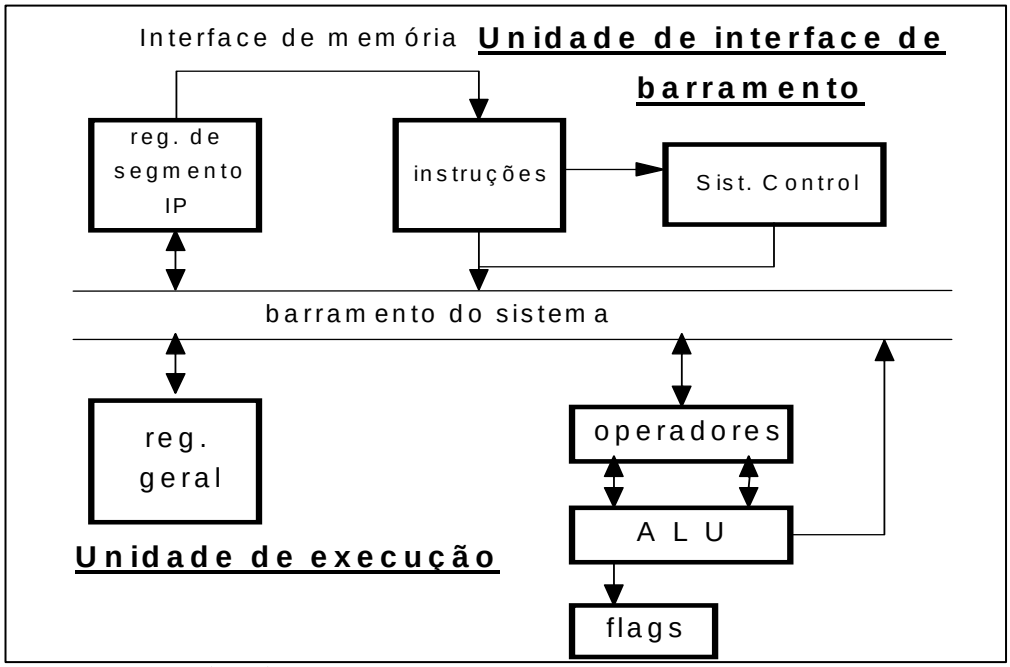
A familiaridade com a máquina-alvo e seu conjunto de instruções, segundo Aho (1995), é um pré-requisito para o projeto de um bom gerador de código.

Neste trabalho será utilizado como máquina-alvo o microprocessador 8088 e compatíveis.

2.5.5.1 ARQUITETURA DOS MICROPROCESSADORES 8088

Conforme descrito em Yeung (1985), as funções internas do processador 8088 são divididas logicamente em duas unidades de processamento, como mostrado no quadro 49.

QUADRO 49 – UNIDADE CENTRAL DE PROCESSAMENTO 8088



Fonte: Yeung (1985).

A unidade de interface de barramento (UIB), possui as funções relacionadas a busca das instruções, acesso a operadores e relocação de endereços.

A unidade de execução (EU), recebe da UIB as instruções a serem executadas, e armazena o resultado na memória.

2.5.5.1.1 REGISTRADORES DE PROPÓSITO GERAL

Segundo Santos (1989), os registradores de propósito geral denominam-se AX, BX, CX, DX, todos de 16 bits. As metades alta e baixa de cada um deles podem ser usadas como registradores de 8 bits, como indicado no quadro 50. Logicamente, o programador pode escolher a conveniência de usar um registrador qualquer de 16 ou 8 bits.

Para muitas operações aritméticas e lógicas, pode-se utilizar qualquer um destes registradores. Com relação ao uso dos registradores de propósito geral, tem-se uma liberdade bem grande na escolha de qual usar para realizar operações aritméticas e lógicas. Entretanto,

existem instruções que usam estes registradores com certas funções especializadas, por exemplo, instruções de multiplicação e divisão concentram-se em AL, AX, BL, BX e DX.

QUADRO 50 – CONJUNTO BASE DE REGISTRADORES DO 8088

Registradores	**16 bits**	**Alta de 8 bits**	**Baixa de 8 bits**
AX	AX	AH	AL
BX	BX	BH	BL
CX	CX	CH	CL
DX	DX	DH	DL

Fonte: Santos (1989).

Segundo Nelson (1991), pode-se endereçar porções selecionadas desses registradores. A parte do registrador que é acessada depende de se estar realizando uma operação em 8 ou 16 *bits*. Cada divisão de um registrador tem um nome separado, AX por exemplo, é o nome de um dos registradores de 16 *bits*, e uma metade do registrador, os 8 *bits* de ordem inferior, é acessível como AL, e a outra, os 8 *bits* de ordem superior, como AH.

Dois registradores adicionais guardam informações de condição sobre o fluxo de instruções em curso. O registrador IP contém o endereço da instrução que está sendo executada, e o registrador FLAGS contém diversos campos, de importância relevante para diferentes instruções (Nelson, 1991).

2.5.5.1.2 REGISTRADORES PONTEIROS E DE ÍNDICE

Conforme descrito em Santos (1989), os registradores BP, SP, SI e DI são usados para armazenar endereços de *offset* dentro de segmentos, onde os dados devem ser acessados. Isto quer dizer que pode-se referenciar um endereço de memória utilizando um destes registradores. O par SP e BP trabalha dentro do segmento da pilha, enquanto o par SI e DI trabalha normalmente no segmento de dados.

O registrador SP é referenciado pelas instruções PUSH e POP para determinar endereço de *offset* do topo da pilha. O BP é utilizado para indicar o endereço de uma área de dados dentro da pilha. E por sua vez os registradores SI e DI também são utilizados para referenciar dados contidos na área de dados do programa.

2.5.5.1.3 REGISTRADORES DE SEGMENTO

Os registradores de segmento CS, DS, SS e ES, segundo Santos (1989), são utilizados para identificar os quatro segmentos endereçáveis naquele momento pelo programa. Cada um deles identifica um bloco de memória, sendo, respectivamente, o código do programa, os dados, a pilha e um segmento extra para dados.

Os códigos de todas as instruções são buscados do segmento de código, onde o registrador IP indica o endereço de *offset*, ou deslocamento dentro daquele segmento, onde está a instrução a ser executada. Por exemplo, um programa onde o registrador CS contenha o valor 113A e IP contendo 210C, o código da próxima instrução a ser executada inicia-se no endereço 134AC, como mostrado no quadro 51.

QUADRO 51 – CÁLCULO DO ENDEREÇO DE INÍCIO DA PRÓXIMA INSTRUÇÃO.

113A0	Endereço do segmento de código x 16
0210C	valor do deslocamento dentro do segmento
134AC	Endereço de memória onde inicia a próxima instrução

Fonte: Santos (1989).

2.5.5.1.4 REGISTRADOR DE SINALIZADORES

Conforme descrito em Santos (1989), o microprocessador 8088 contém em seu interior um total de 9 sinalizadores, também conhecidos como FLAGS (quadro 52). Eles existem para indicar resultados obtidos sempre na última operação lógica ou aritmética executada, ou ainda para definir o comportamento do microprocessador na execução de certas instruções.

QUADRO 52 – REGISTRADOR DE SINALIZADORES

15								7							0
X	X	X	X	Of	Df	If	Tf	Sf	Zf	X	Af	X	Pf	X	Cf

Fonte: Morgan (1988).

2.5.5.1.5 MODOS DE ENDEREÇAMENTO

Conforme descrito em Morgan (1988), os modos de endereçamento ou regras para localizar um operando para uma instrução, para o 8086/8088, podem ser vistos como casos especiais de dois casos gerais: referências de registradores e referências de memórias.

As referências aos registradores são mais simples no sentido de que o operando é simplesmente localizado em um registrador especificado. Entretanto, até quatro quantidades podem ser somadas para determinar o endereço de operando na memória, as quais são (Morgan, 1988): um endereço de segmento; um endereço de base; uma quantidade indexada e um deslocamento.

O endereço de segmento é armazenado no registrador de segmento (DS, ES, SS ou CS), e o conteúdo de um segmento sempre é multiplicado por 16 antes de ser utilizado para formar o endereço efetivo. Para referências de memória, um registrador de segmento é sempre utilizado.

Uma *base* é uma quantidade armazenada em um registrador de base (BX ou BP). Um *índice* é uma quantidade armazenada em um registrador indexado (SI ou DI). Os modos de endereçamento permitem que ambas, uma das duas ou nenhuma destas quantidades seja utilizada na determinação do endereço efetivo.

Em linguagem de montagem (*assembly* por exemplo), uma notação alfanumérica especial em torno do operando, como parênteses ou chaves, indica o modo de endereçamento. O exemplo do quadro 53 possui uma instrução de incremento (INC) e o seu operando está na memória e é acessado utilizando um modo de endereçamento indexado com base e com um deslocamento de 8 bits. Neste caso o endereço de segmento está no segmento de dado, a base está no registrador base (BX), o índice está em DI e o deslocamento é 6.

QUADRO 53 – EXEMPLO DE ENDEREÇAMENTO INDEXADO

```
INC    6[BX][DI]      ;incrementa o conteúdo
                      ;de 'BX + DI + 6'
```

Fonte: Morgan (1988).

2.6 LINGUAGEM *ASSEMBLY*

A linguagem *assembly* segundo Swan (1989) é uma linguagem de computador de aparência singular. No programa-fonte encontram-se palavras com três e quatro caracteres que são os nomes de instruções (mnemônicos) como *JMP*, *MOV*, *ADD*, aparentemente sem haver uma ordem ou relacionamento uma com as outras.

2.6.1 CARACTERÍSTICAS GERAIS DO *ASSEMBLY*

Na linguagem *assembly*, cada linha é composta pelos campos opcionais descritos no quadro 54, onde *nome* é o rótulo dado ao endereço da instrução, e referenciado nas instruções de desvio de fluxo de execução; *operação* é a instrução ou ação sendo tomada; *operandos* são os dados operados pela instrução; e o *comentário* é qualquer texto escrito para elucidar ao leitor do programa, o procedimento ou objetivo daquela instrução.

QUADRO 54 – CAMPOS DE UMA LINHA EM LINGUAGEM *ASSEMBLY*

[nome:] [operação] [operandos] [;comentário]

Fonte: Hozner (1990).

O quadro 55 apresenta um programa-fonte escrito em linguagem *assembly*. Este programa muda um atributo de um arquivo somente para leitura.

Conforme mostrado no quadro 55, as instruções do programa-fonte escrito em linguagem *assembly* podem ser escritas em letras maiúsculas ou minúsculas. Todos os valores numéricos estão por padrão em base decimal, a menos que outra seja especificada e os campos da linha, com exceção do comentário, devem ser separados do anterior por pelo menos um espaço em branco.

A primeira declaração do programa-fonte do quadro 55, é o início de um segmento de código no qual o programa será colocado denominado de *CÓDIGO* e a diretiva *ASSUME* indica que o registrador CS apontará para o segmento *CÓDIGO*. As demais diretivas e instruções serão vistas nas seções seguintes.

2.6.2 INSTRUÇÕES DA LINGUAGEM *ASSEMBLY*

O quadro 56 mostra algumas instruções básicas da linguagem *assembly*, as palavras *op1* e *op2* representam os operandos que especificam os dados. Estas instruções serão utilizadas na geração de código deste trabalho.

QUADRO 55 – EXEMPLO DE PROGRAMA-FONTE EM LINGUAGEM *ASSEMBLY*

```
CODIGO        SEGMENT
              ASSUME  CS:CODIGO
              ORG     100h
PROTEGE       PROC    NEAR
              mov     dx,82h
              mov     di,82h
              mov     al,13
              mov     cx,12
REPNE         SCASB
              mov     byte ptr [di-1],0
              mov     al,1
              mov     cx,1
              mov     ah,43h
              INT     21H
              INT     20H
PROTEGE       ENDP

CODIGO        ENDS
              END     PROTEGE
```

Fonte: Hozner (1990).

QUADRO 56 –INSTRUÇÕES BÁSICAS DA LINGUAGEM ASSEMBLY

MOV	*op1,op2*	move (copia) de *op2* para *op1*
ADD	*op1,op2*	adiciona o valor de *op2* em *op1*
CMPC	*op1,op2*	compara caracteres
SUB	*op1,op2*	subtrai *op2* de *op1* e o resultado fica em *op2*
MUL	*op1*	multiplica *op1* pelo valor de AL e guarda o resultado em AX

Fonte: Morgan (1988).

De forma geral as instruções do 8086/8088 podem ser classificadas em instruções de manipulação de dados, aritméticas, lógicas, de controle de fluxo, de manipulação de *strings*. Na próxima seção serão abordados estes grupos de instruções e também a definição de procedimentos em linguagem *assembly*.

2.6.3 INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

A linguagem *assembly* para 8086/8088 possui uma instrução *MOV* para mover dados de um local para outro. Os operandos podem ter 8 ou 16 bits de tamanho, e eles podem ser registradores de propósito geral, um endereço de memória, ou um dado imediato.

Outras instruções de transferência de dados são *LEA* e *LDS* e são utilizados para carregar o endereço de um nome (*offset*). As instruções *IN*, *INB*, *OUT* e *OUTB* são utilizadas

para entrada e saída de dados a uma fonte ou destino especificado. A forma geral para estas e outras instruções de transferência de dados é mostrada no quadro 57.

QUADRO 57 – INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

Mnemônico/Operandos	Descrição
MOV *op1,op2*	move de *op2* para *op1*
LEA *reg,nome*	carrega em *reg* o endereço de *nome*
LDS *reg,ap*	carrega o apontador *ap* no registrador *reg*
IN *op1*	entrada de *op1* a AX (*word*)
INB *op1*	entrada de *op1* a AL
OUT *op1*	saída de AX (*word*) ao *op1*
OUTB *op1*	saída de AL (*byte*) ao *op1*

Fonte: Morgan (1988).

2.6.4 INSTRUÇÕES ARITMÉTICAS

O 8086/8088 fornece instruções para as quatro operações aritméticas básicas e em operandos de 8 e 16 bits com sinal e sem sinal são fornecidas. A forma geral para as instruções aritméticas é mostrado no quadro 58.

QUADRO 58 – INSTRUÇÕES ARITMÉTICAS

Mnemônico/Operandos	Descrição
ADD *op1,op2*	*Op1:= op1 + op2*
SUB *op1,op2*	*Op1:= op1 – op2*
MUL *op*	AX*:=* AL * *op*
DIV *op*	AL*:=* AX */ op*

Fonte: Morgan (1988).

Na instrução de adição e subtração o resultado é armazenado no primeiro operando, com o estado do registrador de *flags* alterado para refletir o resultado.

As instruções de multiplicação e divisão utilizam registradores específicos para armazenar o resultado da operação.

2.6.5 INSTRUÇÕES LÓGICAS

O 8086/8088 executa quatro operações lógicas NOT, AND, OR e XOR. O resultado da instrução lógica é armazenado no operando de destino com o estado do registrador de *flags* alterado de acordo com o resultado. A forma geral para as instruções lógicas é mostrado no quadro 59 (Swan, 1989).

QUADRO 59 – INSTRUÇÕES PARA OPERAÇÕES LÓGICAS

Mnemônico/Operandos	Descrição
AND *op1,op2*	*Op1*:= *op1* AND *op2*
NOT *op1*	*Op1*:= complemento de um de *op1*
OR *op1,op2*	*Op1*:= *op1* OR *op2*
XOR *op1,op2*	*Op1*:= *op1* XOR *op2*

Fonte: Swan (1989).

2.6.6 INSTRUÇÕES DE CONTROLE DE FLUXO

Conforme descrito em Yeung (1985), as instruções são lidas da memória usando o registrador CS e o registrador IP. Desta forma para desviar a execução para outro bloco de programa altera-se o conteúdo dos registradores CS e IP ou somente IP se o desvio for no mesmo segmento.

O 8086/8088 possui diversas instruções para controle de fluxo que podem ser de desvio condicional, incondicional, de repetição e de interrupções. A forma geral de algumas destas instruções para controle de fluxo é mostrado no quadro 60.

O *JMP* incondicional com um operando (*alvo*) é um pulo direto no mesmo segmento ao endereço fornecido pelo operando. O operando realmente contêm um endereço relativo denominado deslocamento. Para obter o endereço verdadeiro deve-se acrescentar o valor do operando ao valor atual do indicador de instrução (IP).

QUADRO 60 – INSTRUÇÕES DE CONTROLE DE FLUXO

Mnemônico/Operandos		**Descrição**
Instruções de desvio incondicional		
CALL	*Proc*	chamada de procedimento
JMP	*Label*	desvio interno ao segmento; label é definido no mesmo segmento como *near*;
JMP	*Reg16*	desvio indireto interno ao segmento; *offset* é especificado no conteúdo de *reg16*;
RET	*Valor*	retorna do procedimento;
RETN	*valor*	retorna do procedimento definido como *near*;
RETF	*valor*	retorno do procedimento definido como *far;*
Instruções de desvio condicional		
JE	*alvo*	desvia para *alvo* se igual
JZ	*alvo*	desvia para *alvo* se zero
JNZ	*alvo*	desvia para *alvo* se não-zero
Instruções de repetição		
JCXZ	*alvo*	desvia para *alvo* se CX for igual a 0
LOOP	*alvo*	repete a partir de *alvo* enquanto CX for diferente de 0
Instruções de interrupção		
INT	*tipo*	executa a interrupção *tipo*
INTO		executa interrupção em *overflow*
IRET		executa interrupção de retorno

Fonte: Swan (1989).

2.6.7 MANIPULAÇÃO DE *STRINGS*

Segundo Holzner (1990), a linguagem *assembly* se sobressai no manuseio de cadeias de caracteres (*strings*), principalmente quando procura ou compara. As operações com *strings* assumem que, se os dados serão movidos, eles serão movidos do endereço ES:[SI] para o DS:[DI]. As instruções que manipulam *strings* incrementam ou decrementam esses índices automaticamente.

O 8086/8088 manuseia tanto *strings* de bytes como de palavras (*words*). Pelo fato de ser possível manipular apenas um *byte*/*word* de cada vez, as instruções de manipulação de caracteres possuem um prefixo de repetição denominado REP. A forma geral de algumas instruções de manipulação de caracteres é apresentada no quadro 61.

QUADRO 61 – INSTRUÇÕES DE MANIPULAÇÃO DE *STRINGS*

Mnemônico/Operandos		Descrição
LEA	*re1g,msg1*	Carrega o endereço efetivo de *msg1* no registrador *reg1*
MOVS	*msg1,msg2*	move o caracter de *msg2* apontado por SI para o endereço de *msg1* apontado por DI
CMPS	*msg1,msg2*	compara o caracter de *msg1* apontado por DI com o caractere de *msg2* apontado por SI
STOSB		move o conteúdo de AL para a posição indicada por ES:DI
STOSW		move o conteúdo de AX para a posição indicada por ES:DI
REP	*instruçãoString*	repete a instrução de *string* até CX chegar 0
REPE	*instruçãoString*	repete a instrução de *string* enquanto os caracteres comparados forem iguais

Fonte: adaptado de Holzner (1990).

2.6.8 SUBPROGRAMAS

Quando a mesma sucessão de instruções é requerida executando várias vezes em várias partes de um programa, estas instruções devem ser definidas como procedimento.

Em linguagem *assembly* utiliza-se os pseudo-operadores *PROC* e *ENDP* para definir um procedimento, onde o primeiro define o nome e marca o início do procedimento e o segundo o fim do código deste procedimento. O atributo de um procedimento pode ser *FAR* indicando que o procedimento pode ser chamado de outro segmento ou *NEAR* que indica que o procedimento só pode ser chamado dentro do segmento corrente. Um exemplo de definição de um procedimento em linguagem *assembly* é apresentado no quadro 62.

QUADRO 62 – PROCEDIMENTOS EM LINGUAGEM *ASSEMBLY*

```
Teste   PROC    NEAR    ;definição do procedimento de nome teste
        .....           ;instruções
        RET             ;instrução de retorno de procedimento
Teste   ENDP            ;fim do procedimento
```

Fonte: adaptado de Holzner (1990).

Um procedimento só pode ser chamado de um outro segmento em programas com formato *.EXE* (Holzner, 1990). Em programas *.COM*, que será descrito na seção seguinte, todos os procedimentos devem ser definidos com o atributo *NEAR*.

Os arquivos podem ser agrupados em duas categorias: arquivos de múltiplos segmentos (*.EXE*) que são organizados em segmentos individuais e separados e arquivos de segmento único (*.COM*) que contêm somente um segmento, o qual inclui todos os dados e os códigos das instruções e neste caso os registradores CS, DS, SS e ES apontam para o mesmo segmento.

Será abordado na próxima seção somente a estrutura dos arquivos de segmento único, pelo fato de que o presente trabalho possui como objetivo no que diz respeito ao código-alvo gerar um programa executável de estrutura *.COM*.

2.6.9 ESTRUTURA DOS ARQUIVOS *.COM*

Os arquivos executáveis *.COM* contêm apenas um segmento definido, que contêm todos os dados do programa. Isto é feito criando-se um segmento para o código e, através do pseudo-operador *ASSUME* (quadro 63), faz com que os quatro registradores de segmento referenciem o mesmo bloco físico de 64 *Kbytes* (Santos, 1989).

Um programa consiste de um ou mais segmentos, no caso de um arquivo *.COM,* apenas um segmento pode ser definido. O pseudo-operador utilizado para indicar o início de um segmento é o *SEGMENT*, que segue após o nome do segmento, e para indicar o fim do segmento utiliza-se *ENDS* precedido do nome do segmento, como mostrado no quadro 63.

A linha *ORG 100H* (quadro 63), define onde o registrador IP deverá ser posicionado, ou seja as instruções devem iniciar no *byte* 100H do programa. Isto deve ser feito nos programas *.COM*, porque deve-se reservar 256 *bytes* (100H) para o prefixo de segmento do programa (PSP), o cabeçalho do programa providenciado pelo DOS. Este prefixo sempre começa em CS:0000. O código executável para todos os programas *.COM* sempre iniciam logo após o PSP, em CS:0100 (Holzner, 1990).

A instrução após o *ORG 100H* (*JMP COMECO*), será colocada em CS:100H. No exemplo do quadro 63, é uma instrução de desvio, ou seja um pulo sobre os dados que usualmente colocam-se logo no início antes de serem referenciados. Desta forma quando um programa *.COM* é executado o registrador IP apontará para CS:100H, e após a execução da

instrução de desvio pulará os dados para iniciar a execução do procedimento principal do programa.

A instrução *INT 20H* do quadro 63 finaliza o programa retornando o controle ao sistema operacional. Outra maneira de terminar procedimentos é a instrução RET. Quando se tem uma instrução RET no fim do programa o registrador IP passa a valer ‘0000’ que é o endereço do primeiro *byte* de PSP. Os dois primeiros *bytes* de PSP, para qualquer programa é o código de máquina para a instrução INT 20H. Uma instrução RET no fim de um programa *.COM*, portanto, sempre envia o controle para o início do PSP, onde INT 20H é executada e este método é o mesmo que incluir INT 20H no final do programa.

QUADRO 63 - ESTRUTURA DE UM ARQUIVO *.COM*

```
CODIGO   SEGMENT
         ASSUME  CS:CODIGO, SS:CODIGO, DS:CODIGO, ES:CODIGO
         ORG     100H
INICIO:  JMP     COMECO
         |
         definição de variáveis
         |
COMECO   PROC    NEAR
         |
         corpo de instruções
         |
         INT 20H
COMECO   ENDP
CODIGO   ENDS
         END     INICIO
```

Fonte: Swan (1989).

Em um programa *.COM*, por ser de segmento único, e por isso seus procedimentos não podem ser chamados de fora de seu segmento, todas as rotinas inclusive a principal como mostrado no quadro 63 devem ter o atributo *NEAR* (Santos, 1990).

Um programa *.COM* é armazenado em disco na forma de imagem de memória, isto é, o arquivo contém exatamente os *bytes* do programa. O seu tamanho não pode ser superior a 64 *Kbytes* (Quadros, 1988).

Os passos que descrevem a carga de um programa *.COM* segundo Quadros (1988) são:

a) o DOS monta um Prefixo de Segmento de Programa (PSP) para o programa;
b) o programa é carregado logo após a PSP, ou seja, no endereço PSP:100h, através da sua leitura para a memória;
c) os registradores de segmento (CS, DS, ES, SS) são carregados com o segmento da PSP, e o registrador SP fica com 0FFF0h ou aponta para o fim da memória, se a memória livre for inferior a 64 *Kbytes*;
d) os registradores AL e AH são carregados com 0FFh ou 000h, conforme os dois primeiros parâmetros sejam nomes válidos ou inválidos para arquivos;
e) é colocado na pilha um *word* (2 bytes) com zeros, de forma que um *RET NEAR* passe o controle para a *int 20h* da PSP, causando o término do programa;
f) o programa é iniciado pela execução da instrução em 100h.

Um programa *.COM* deve independer do segmento em que foi carregado, visto não sofrer nenhuma relocação quando da carga. Deve também ter sua primeira instrução no primeiro byte do arquivo, que será carregado no *offset* 100h (Quadros, 1988).

Normalmente, um programa em linguagem *assembly .COM* é gerado através de três passos:

a) o programa fonte é convertido em objeto executável pelo assembler;
b) o objeto relocável é convertido em objeto executável *.EXE*;
c) o objeto *.EXE* é convertido em *.COM*.

Quando os programas *.COM* são carregados na memória, eles são colocados no primeiro endereço disponível que seja múltiplo de 16 bytes (Holzner, 1990).

2.6.10 PREFIXO DE SEGMENTO DE PROGRAMA (PSP)

Segundo Holzner (1990), quando o DOS carrega um programa .*COM* na memória, ele monta um prefixo de segmento do programa, ou PSP. Um PSP montado para os programas .*COM* deve estar localizado dentro do segmento comum.

Um programa .*COM* típico na memória tem um PSP de CS:0000 a CS:00FF, seguido pelo programa propriamente dito em CS:0100. Quando um programa .*COM* é carregado, todos os registradores de segmento (CS, DS, SS) são posicionados para o mesmo segmento comum. Até mesmo a pilha, indicada por SS:SP, deve estar no mesmo segmento (Holzner, 1990).

O PSP tem 256 bytes e fornece ao programa informações sobre o que foi digitado na linha de comando e outros itens. As instruções são colocadas após o PSP na memória porque o programa está contido em um único segmento (Holzner, 1990).

2.7 AMBIENTE FURBOL

O ambiente FURBOL foi desenvolvido na Universidade Regional de Blumenau por André (2000), objetivando auxiliar no ensino de introdução a programação de computadores. Com a intenção de facilitar o seu uso, os comandos da linguagem são descritos na língua portuguesa.

Para a especificação da linguagem FURBOL foram utilizados conceitos de BNF. Para montagem da definição de escopo e geração de código foram utilizados ações semânticas, descrito através de gramática de atributos. O código descrito é também gerado em código de máquina tipo iAPX 86/88.

2.7.1 PRINCIPAIS CARACTERÍSTICAS DA LINGUAGEM FURBOL

As características principais da linguagem FURBOL são:

a) utilização de dados do tipo *inteiro, lógico* e *array (matriz)*;
b) comando condicional *se*, *então* e *senão*;

c) comando de repetição *enquanto faça*;

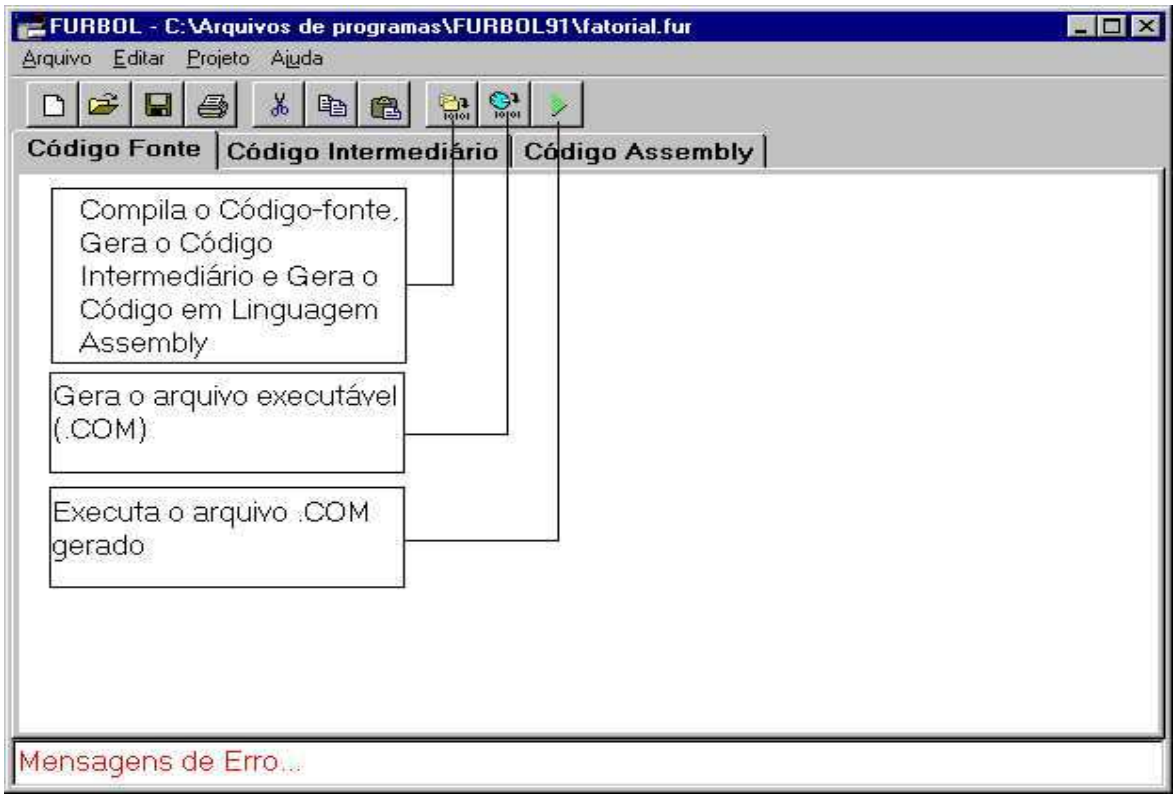
d) comando de entrada e comando de saída;

e) unidade do tipo *procedimento* com passagem de parâmetro por cópia-valor e por referência.

2.7.2 INTERFACE DO AMBIENTE FURBOL

O protótipo desenvolvido possui a janela principal onde são editados os programas fonte. Esta janela possui um menu com todas as funções disponíveis. Possui também uma barra de ferramentas onde podem ser acionadas as principais funções do menu. Esta janela principal é mostrada na figura 2.

FIGURA 2 – TELA PRINCIPAL DO AMBIENTE FURBOL



A opção “Projeto” do menu principal oferece três funções que são:

a) “Compilar”, que efetua a compilação do programa editado informando a existência ou não de erros (figura 3). Caso exista algum erro no programa-fonte, o compilador irá selecionar o *token* com o erro, especificando o erro na barra de *status*, na parte

inferior da tela principal. Se nenhum erro for encontrado será emitida a mensagem “Programa sem erros”. O código intermediário formado por enunciados de três endereços será criado na guia “Código Intermediário” (figura 4), e o código em linguagem *assembly* será criado na guia “Código Assembly”, como mostrado na figura 5;

b) “Gerar Código” gera o código em linguagem de máquina a partir do código em linguagem *assembly*. O montador *Turbo Assembler* é acionado e um programa com a extensão *.COM* é criado a partir do arquivo *.ASM* que contêm o código em linguagem *assembly* gerado pelo compilador;

c) “Executar”: ao se ativar esta função inicia-se a execução do programa *.COM* anteriormente criado.

FIGURA 3 – DETECÇÃO DE ERROS NO PROTÓTIPODESENVOLVIMENTO DO PROTÓTIPO

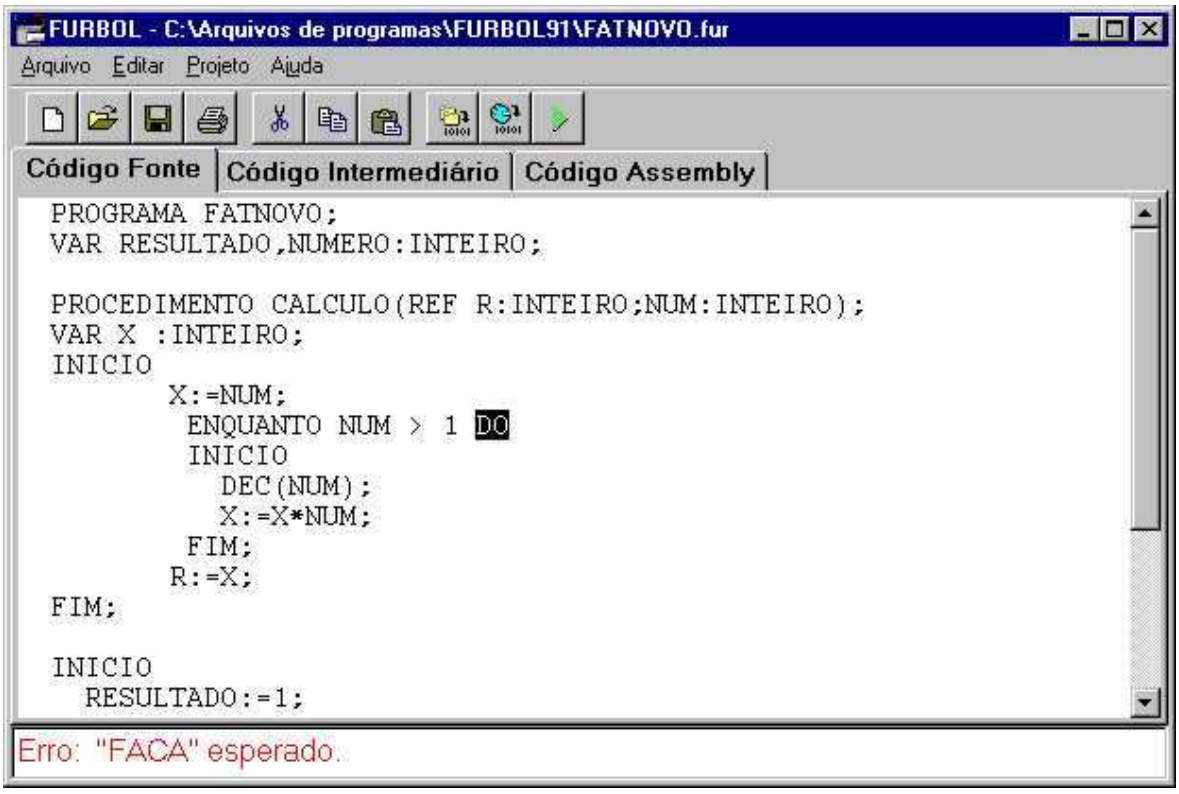


FIGURA 4 - JANELA PRINCIPAL COM CÓDIGO INTERMEDIÁRIO

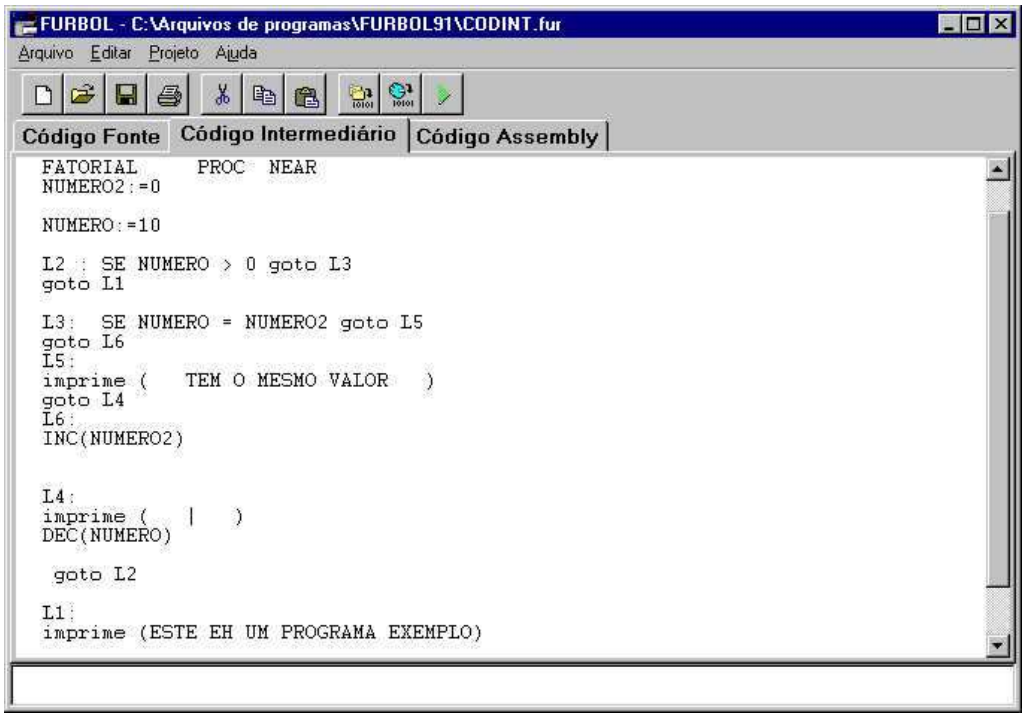
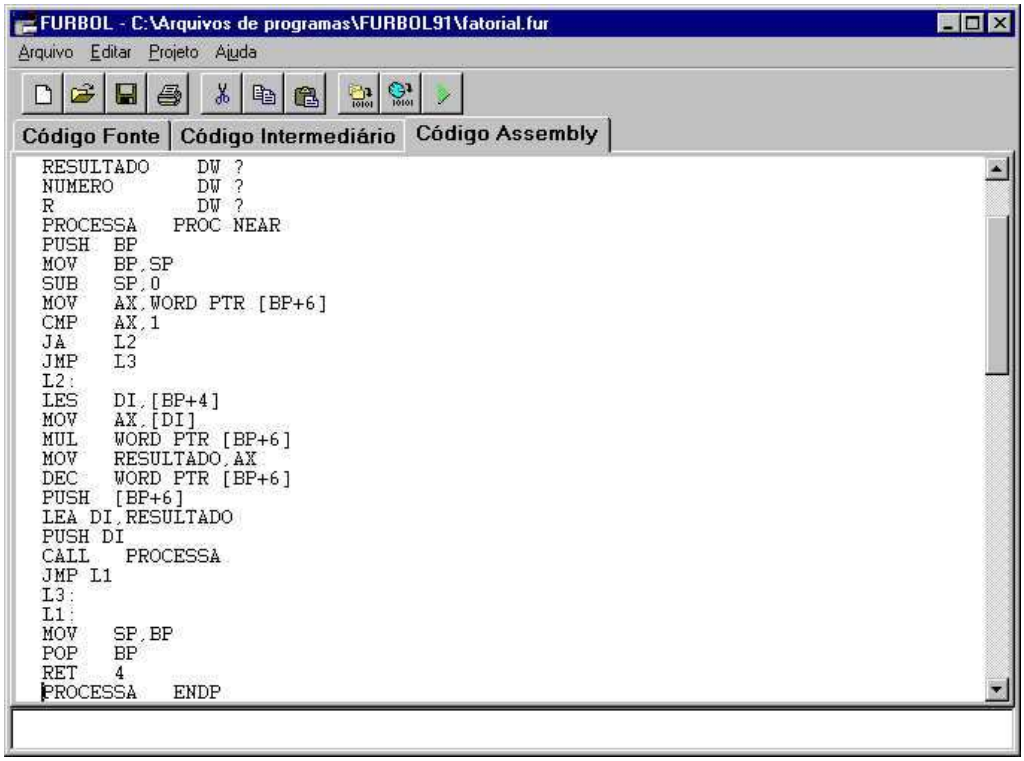


FIGURA 5 – JANELA DO PROTÓTIPO COM CÓDIGO ASSEMBLY



3 DESENVOLVIMENTO DO PROTÓTIPO

Neste capítulo será apresentada a definição da linguagem FURBOL proposta neste trabalho. Esta definição é uma revisão da apresentada por André (2000), com o acréscimo de novas produções para implementação de *array's* semi-estáticos. O nome FURBOL significa a concatenação da sigla que referencia a Universidade Regional de Blumenau (FURB) com ALGOL, que é uma linguagem que utiliza estrutura de blocos para controlar o escopo das variáveis e a divisão do programa em unidades. O desenvolvimento do protótipo apresenta três etapas: definição de escopos; especificação da linguagem e apresentação do protótipo.

3.1 DEFINIÇÃO DE ESCOPO IMPLEMENTADA

Nesta seção será abordada as técnicas utilizadas para implementação de escopos a nível de declarações e chamadas de procedimentos.

A cada chamada de procedimento o tratamento é feito seguindo a regra do aninhamento mais interno. Por exemplo no programa *sort* apresentado no quadro 64, o procedimento *exchange*, chamado por *partition* à linha 17, é não local a *partition*. Aplicando-se a regra, primeiro verifica-se se *exchange* está definido dentro de *quicksort*; como não está procura-se no programa principal *sort*.

QUADRO 64 – PROGRAMA PASCAL COM PROCEDIMENTOS ANINHADOS

```
(1)     program sort;
(2)         var  a : integer;
(3)              x : integer;
(4)     procedure readarray;
(5)         var  i : integer;
(6)     begin . . . a . . . end  {readarray};
(7)     procedure exchange;
(8)     begin . . .
(9)              ......
(10)    end {exchange};
(11)    procedure quicksort;
(12)        var  k, v : integer;
(13)        procedure partition;
(14)        var  i, j : integer;
(15)        begin  . . . a . . .
(16)            . . . v . . .
(17)            . . . exchange; . . .
(18)        end  {partition};
(19)    begin . . . end  {quicksort};
(20)    begin . . . end. {sort}.
```

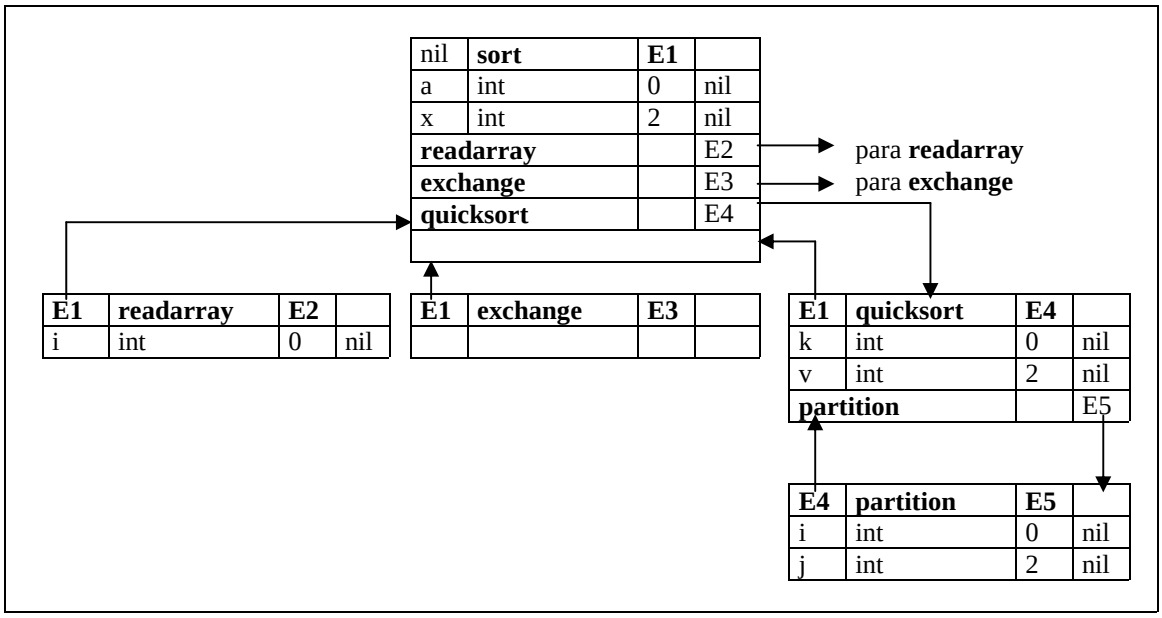
Fonte: Baseado em Aho (1995).

No quadro 64 pode-se ter a noção de profundidade de procedimentos aninhados, onde o programa principal *sort* está com profundidade 1; então adiciona-se 1 à profundidade de aninhamento à medida que encaminha-se de um procedimento envolvente para outro envolvido. Desta forma, o procedimento *quicksort* está com profundidade 2 e por sua vez *partition* está com profundidade 3. A cada ocorrência de um nome associa-se a profundidade de aninhamento do procedimento no qual é declarado (Aho, 1995).

Para cada procedimento deve ser criado uma estrutura que armazenará todas as variáveis e procedimentos declarados dentro deste procedimento. Esta estrutura denomina-se tabela de símbolos.

No quadro 65 são mostradas as tabelas de símbolos para os procedimentos do programa do quadro 64. A estrutura de aninhamento pode ser mais facilmente observada a partir da relação entre as tabelas de símbolos estabelecida pelas entradas em cada tabela. As tabelas de símbolos *readarray*, *exchange* e *quicksort* apontam de volta para a tabela que contém o procedimento principal *sort*. A tabela do procedimento *partition* por ser declarado dentro do procedimento *quicksort* aponta para a tabela de *quicksort.*

QUADRO 65 – TABELAS DE SÍMBOLOS PARA PROCEDIMENTOS ANINHADOS



Fonte: Aho (1995).

Para verificação do escopo de um nome (variável ou procedimento), é necessário realizar uma busca na tabela de símbolos do procedimento atual. Caso não seja encontrada,

deve-se realizar buscas nas tabelas de símbolos dos procedimentos ancestrais apontados pelas relações entre as tabelas até que seja encontrada entrada para o nome.

Para construir a estrutura apresentada no quadro 65 é necessário incorporar ações semânticas nas produções que envolvem procedimentos e variáveis. No quadro 66 está descrito a definição dirigida pela sintaxe para declarações de procedimentos aninhados.

QUADRO 66 – DEFINIÇÃO DIRIGIDA PELA SINTAXE PARA DECLARAÇÕES DE PROCEDIMENTOS ANINHADOS

Produção		**Regras semânticas**	
P	::=	M D;	{registrar_largura(topo(ptr_Tab), topo(deslocamento)); desempilhar(ptrTab); desempilhar(deslocamento)}
M	::=	^;	{t:=criar_tabela(nil); empilhar(t,ptr_tab);empilhar(0,deslocamento)}
D	::=	D_1;D_2	
D	::=	**proc id** ; N D_1; S	{t:=topo(ptr_tab); registrar_largura(t,topo(deslocamento)); desempilhar(ptr_tab); desempilhar(deslocamento); instalar_proc(topo(ptr_tab), id.nome, t)}
D	::=	id: T	{instalar(topo(ptr_tab), id.nome, T.tipo, topo(deslocamento)); topo(deslocamento):=topo(deslocamento) + T.largura }
N	::=	^;	{t:=criar_tabela(topo(ptr_tab)); empilhar(t,ptr_tab); empilhar(0,deslocamento)}

Fonte: Aho (1995).

As regras semânticas do esquema de tradução do quadro 66 são definidas em termos das seguintes operações:

a) *criar_tabela(anterior)* cria uma nova tabela de símbolos retornando um apontador para a mesma. O argumento *anterior* aponta para a tabela anteriormente criada;

b) *instalar(tabela, nome, tipo, deslocamento)* cria uma nova entrada para o nome *nome* na tabela de símbolos apontada por *tabela* e coloca o tipo *tipo* e o endereço relativo *deslocamento* nos campos da entrada criada;

c) *registrar_largura(tabela, largura)* registra a largura acumulada de todas as entradas de *tabela* no cabeçalho associado a esta tabela de símbolos;

d) *instalar_proc(tabela, nome, nova_tabela)* cria uma nova entrada para o nome de procedimento *nome* na tabela de símbolos apontada por *tabela*. O argumento *nova_tabela* aponta para a tabela de símbolos do procedimento *nome*;

e) empilhar(endereço, ptrpilha) empilha o endereço apontado por endereço na pilha ptrpilha;

f) *desempilha(ptrpilha)* desempilha um elemento da pilha *ptrpilha*.

O esquema de tradução do quadro 66 mostra como os dados podem ser dispostos em uma única passagem, usando-se a pilha *ptr_tab* para guardar os apontadores para as tabelas de símbolos dos procedimentos envolventes. Com as tabelas de símbolos do quadro 65, *ptr_tab* irá conter apontadores para as tabelas de *sort*, *quicksort* e de *partition* quando as declarações dentro de *partition* forem consideradas. O apontador para a entrada corrente da tabela de símbolos está ao topo.

A ação para o não-terminal *M* inicializa a pilha *ptr_tab* com a tabela de símbolos para o escopo mais externo, criada pela operação *criar_tabela(nil).* A ação também empilha o endereço relativo 0 (zero) sobre a pilha *deslocamento*. O não-terminal *N* desempenha um papel similar quando uma declaração de procedimento aparece. Sua ação usa a operação *criar_tabela(topo(ptr_tab))* para criar uma nova tabela. Um apontador para a nova tabela é empilhado acima daquele para o escopo envolvente. Novamente, 0 é empilhado sobre a pilha deslocamento.

Para cada declaração de variável *id : T*, uma entrada é criada para *id* na tabela de símbolos. Esta declaração deixa a pilha *ptr_tab* inalterada; o topo da pilha *deslocamento* é incrementado por *T.largura*. Quando a ação ao lado direito de $D ::= proc\ id;\ N\ D_1;\ S$ ocorre, a largura de todas as declarações geradas por D_1 está ao topo da pilha deslocamento; é registrada usando-se *registrar_largura*. As pilhas *ptr_tab* e *deslocamento* têm, então, seus topos removidos e volta-se a examinar as declarações do procedimento envolvente. A esse ponto, o nome do procedimento é introduzido na tabela de símbolos de seu procedimento envolvente.

3.2 ESPECIFICAÇÃO DA LINGUAGEM FURBOL

Nesta seção será apresentada a especificação da linguagem FURBOL implementada no protótipo. Esta especificação é uma extensão da especificação apresentada em André (2000), na qual foram incluídas ações semânticas para geração de código executável para tratar estruturas do tipo mapeamentos finitos.

O símbolo sustenido (#) significa que o identificador (ID) representa um elemento léxico, ou seja, um símbolo terminal da gramática. As palavras entre apóstrofes (‘’) também são consideradas como símbolos terminais, podendo ser palavras reservadas ou literais. As

demais palavras são consideradas elementos não-terminais (como por exemplo, EstruturaDados, ComandoComposto e outros), os quais possuem derivações. O símbolo circunflexo (^) significa a existência de uma palavra vazia.

Os atributos *código* e *codAsm* contêm o código intermediário e o código *assembly* respectivamente. O símbolo "||" é utilizado para representar a concatenação dos enunciados e atributos que armazenam os códigos à medida que são gerados.

3.2.1 PROGRAMAS E BLOCOS

A definição de programas e blocos é mostrada no quadros 67. A função *CriarTabela* cria uma nova tabela de símbolos e retorna o endereço da tabela criada e a função *Empilhar* coloca na pilha *ptrtab* o endereço armazenado em *t*, e na pilha *Deslocamento* coloca o valor do deslocamento inicial.

A ação semântica *GeraAsm* é responsável pela construção do código em linguagem *Assembly*. O procedimento *DeclaraDados* declara os dados em linguagem *Assembly*, sendo que somente são declaradas as variáveis que são globais dentro da definição de escopo implementada.

No quadro 67 o argumento *ID.nome* contêm o nome do programa que será declarado em *assembly* pela instrução *ID.nome PROC NEAR*. A instrução *PUSH BP*, salva (coloca) na pilha o valor de *BP* para que este seja restaurado antes do término do programa com a instrução *POP BP*. A instrução *int 20h* (quadro 67), encerra o programa sendo seguida da instrução *nomeprog ENDP*.

QUADRO 67 - DEFINIÇÃO DE PROGRAMAS E BLOCOS

Programa	::=	‘PROGRAMA’,	GeraAsm('CODIGO SEGMENT'); GeraAsm('ASSUME CS:CODIGO, DS:CODIGO'); GeraAsm('ORG 100H');
		#ID,	ID.nome:=#ID; GeraAsm('ENTRADA: JMP ' || #ID.nome); nivel:=0;
		‘;’,	T:=CriarTabela(nil); Empilhar(t,ptrtab);Empilhar(0,Deslocamento);
		EstruturaDados,	nível:=nivel+1;
		EstruturaSubRotinas,	nivel:=nivel-1;
		CComposto	GeraAsm(EstruturaSubRotinas.codAsm || ID.nome ||' PROC NEAR' || 'PUSH BP' || 'MOV BP,SP' || CComposto.CodAsm || 'POP BP' || 'int 20h' || ID.nome || ENDP);

		‘.’	DeclaraDados; // Variáveis globais em Assembly
			GerarASm(CODIGO ENDS);
			GerarAsm(END ENTRADA);
			RegistrarLarg(Topo(PtrTab),Topo(Deslocamento));
			Desempilha(PtrTab);Desempilha(Deslocamento);

Bloco	::=	EstruturaDados,	
		EstruturaSubRotinas,	
		CComposto;	Bloco.Codigo := EstruturaSubrotina .Codigo || CComposto.Codigo; Bloco.CodAsm:= EstruturaSubRotina.CodAsm || CComposto.CodAsm;
CComposto	::=	‘INICIO’,	
		Comando,	CComposto.Codigo:=Comando.Codigo; CComposto.CodAsm:=Comando.CodAsm;
		‘FIM’;	

3.2.2 ESTRUTURAS DE DADOS

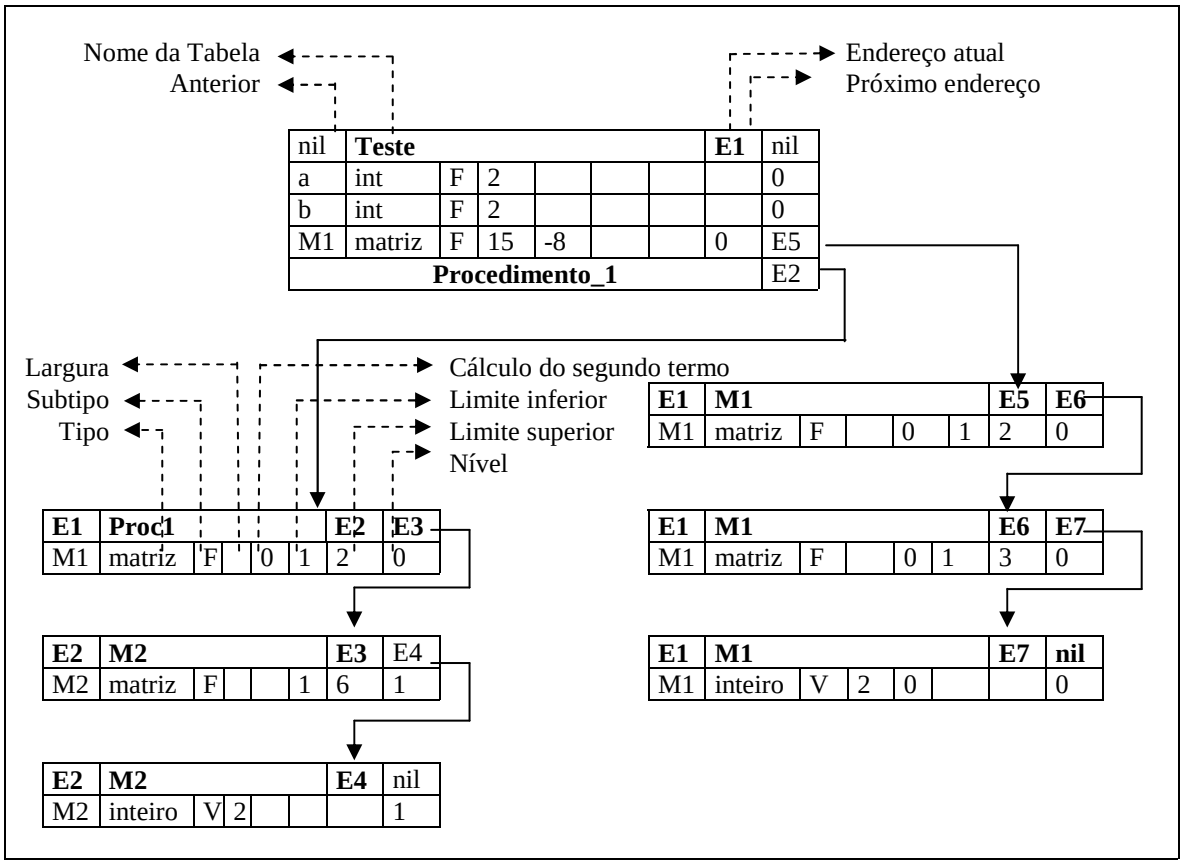
A definição das estruturas de dados é mostrada no quadro 70 e 71. A ação semântica *Instalar* cria uma nova entrada para o *ID.nome* na tabela de símbolos apontada por *TopoPtrTab*. A ação semântica *AtualizarTipo* coloca na tabela apontada por *TopoPtrTab* o tipo das variáveis que está em *T.tipo* e o endereço relativo “*deslocamento”*, que é o valor do tamanho de memória ocupado pelo referido tipo.

Se a variável for do tipo *matriz*, a ação semântica *AtualizarMatriz* atualiza a tabela de símbolos com a largura da matriz e armazena o cálculo do segundo termo descrita na fórmula apresentada no quadro 35, que é avaliada quando a declaração do *array* for enxergada. O quadro 68 mostra um exemplo de declaração de *array's* e o quadro 69 apresenta a tabela de símbolos mostrada para o respectivo exemplo.

QUADRO 68 – EXEMPLO DE DECLARAÇÕES DE *ARRAY'S*

```
programa teste;
Var
   a,b:inteiro;
    M1:matriz[1..2,1..3]:inteiro;
  procedimento proc1;
  var
     M2:matriz[1..6]:inteiro;
 Inicio
  fim;
Inicio
fim.
```

QUADRO 69 – TABELAS DE SÍMBOLOS PARA DECLARAÇÕES DE *ARRAY'S*



Fonte: Baseado em Aho (1995).

QUADRO 70 – DEFINIÇÃO DAS ESTRUTURAS DE DADOS

EstruturaDados	::=	‘VAR’,	
		#ID,	Se não (encontra_var(ID)) então Instalar(TopoPtrTab),ID.Nome);
		ListaID, (Matriz|^)	AtualizarTipo(Topo(PtrTab),NroVar,T.tipo,Topo(Deslocamento),TipoDesloca,LimInf,LimSup,SubTipo); Topo(Deslocamento):=Topo(Deslocamento)+T.Largura;
		‘;’,	
		Declarações	
		| ^;	
Declarações	::=	#ID,	Se não (encontra_var(ID)) então Instalar(TopoPtrTab),ID.Nome,ID.tipo,param);
		ListaID, (Matriz|^)	AtualizarTipo(Topo(PtrTab),NroVar,T.tipo,Topo(Deslocamento),TipoDesloca,LimInf,LimSup,SubTipo); Topo(Deslocamento):=Topo(Deslocamento)+T.Largura;
		‘;’,	
		Declarações	
		|^;	
ListaID	::=	‘,’,	
		ID,	Se não (encontra_var(ID)) então Instalar(TopoPtrTab),ID.Nome,ID.Tipo,param);
		ListaID, (Matriz|^)	AtualizarTipo(Topo(PtrTab),NroVar,T.tipo,Topo(Deslocamento),TipoDesloca,LimInf,LimSup,SubTipo); Topo(Deslocamento):=Topo(Deslocamento)+T.Largura;
		| ‘:’,	
		Tipo;	ListaId.Tipo:=Tipo;
Tipo	::=	‘INTEIRO’	T.Tipo:=2; T.Largura:=2;
		|	
		‘LOGICO’;	T.Tipo:=1;T.Largura:=1;
		|	
		‘MATRIZ’’	T.Tipo:=4;
Matriz	::=	LimitesM, ‘:’, Tipo	AtualizarTipo(Topo(PtrTab),NroVar,T.tipo,Topo(Deslocamento),TipoDesloca,LimInf,LimSup,SubTipo); Topo(Deslocamento):=Topo(Deslocamento)+T.Largura;
		| ^	
LimitesM	::=	‘[‘, Dimensao, ‘]’	
Dimensao	::=	#NUM, ‘..’, #NUM,	T:=CriarTabela(nil); Empilhar(t,ptrtab);Empilhar(0,Deslocamento); Instalar(Mid,Mtipo,Mparam,Lsup,Linf);
		MaisDimensao	

QUADRO 71 – DEFINIÇÃO DAS ESTRUTURAS DE DADOS (CONTINUAÇÃO)

MaisDimensao	::=	‘,’, Dimensao	
	|		
		^	

3.2.3 ESTRUTURA DE SUBROTINAS

A estrutura de subrotinas pode ser vista nos quadros 72 e 73. A função *instalar_Proc* cria uma nova entrada para o nome de procedimento *id.nome* na tabela de símbolos apontada por *TopoPtrTab*. O argumento *t* aponta para a tabela do procedimento *id.nome*.

O argumento *ID.nome* (quadro 71), contêm o nome do procedimento declarado, a instrução *PUSH BP* salva na pilha o valor atual do registrador *BP* e logo em seguida o valor de *SP* que contêm o topo da pilha é movido para *BP*. O valor do argumento *Proc.largura* é a quantidade de *bytes* das variáveis locais do procedimento.

A instrução *SUB SP,Proc.largura* conforme mostrado no quadro 71, reserva no topo da pilha a quantidade de memória necessária para os dados locais do procedimento.

O argumento *NRET* (quadro 71), contêm o número de *bytes* que deverão ser desempilhados quando a instrução *RET* for executada. O valor *NRET* é calculado através do número de parâmetros e registradores que são colocados na pilha antes da chamada do procedimento.

QUADRO 72 – DEFINIÇÃO DA ESTRUTURA DE SUBROTINAS

EstruturaSubRotinas	::=	EstruturaProcedimento	EstruturaSubRotina.Codigo:= EstruturaProcedimento.Codigo EstruturaSubRotina.CodAsm:= EstruturaProcedimento.CodAsm;
		| ^;	
EstruturaProcedimento	::=	'PROCEDIMENTO',	

		#ID,	se encontra_var(id.nome)<>nil entao erro;
			T:=CriarTabela(Topo(PtrTab));
			instalar_Proc(Topo(PtrTab),Id.Nome, t, Desvio);
			Empilhar(T,PtrTab);Empilhar(0,Deslocamento);
			Nret:=0;

		ParamFormais,	CComposto.Nret:=ParaFormais.Nret;
		';',	
		EstruturaDados,	nivel:=nivel+1;
		EstruturaSubRotinas,	nivel:=nivel-1;
		CComposto,	EstruturaProcedimento.CodigoAsm := ID.nome || PROC NEAR || PUSH BP || MOV BP,SP || SUB SP,Proc.largura || CComposto.CodAsm || MOV SP,BP || POP BP || RET Nret || ID.nome ENDP|| EstruturaSuRotinas.CodAsm;
		';',	T:=Topo(PtrTab); Registrar_Largura(t,Topo(Deslocamento)); Desempilhar(PtrTab);Desempilhar(Deslocamento);
		EstruturaSubRotinas;	EstruturaProcedimento.Codigo := EstruturaProcedimento.Codigo ||EstruturaSubRotina.Codigo; EstruturaProcedimento.CodAsm := EstruturaProcedimento.CodAsm ||EstruturaSubRotina.CodAsm;
ParamFormais	::=	'(',	
		(ParamValor |	
		ParamRef),	
		SecaoParam,	
SecaoParam	::=	';'	
		(ParamValor|	
		ParamRef),	
		SecaoParam	
		| ^	

QUADRO 73 – DEFINIÇÃO DA ESTRUTURA DE SUBROTINAS (CONTINUAÇÃO)

		‘)’	ParamFormais.Nret:=Desloc[TopoDesloc];
		| ^;	

ParamValor	::=	#ID,	Se encontraVar(id.nome)<>nil entao erro;
			param:=1;
			intalar(Topo(PtrTab),id.nome, nivel, Param);

		ListaID;	AtualizarTipo(PtrTab[TopoPtrTab],Nvar, ListaID.Tipo,Desloc[TopoDesloc]); Desloc[TopoDesloc]:=Desloc[TopoDesloc]+ListaId. .Largura;
ParamRef	::=	‘REF’,	

		#ID,	Se encontraVar(id.nome)<>nil entao erro;
			param:=2;
			instalar(topo(ptrtab), id.nome,nivel, Param);

		ListaID	AtualizarTipo(PtrTab[TopoPtrTab],Nvar, ListaIDTipo,Desloc[TopoDesloc]); Desloc[TopoDesloc]:=Desloc[TopoDesloc]+ListaId. .Largura;

No exemplo do quadro 74 o procedimento *CALCULO* possui dois parâmetros e uma variável local do tipo inteiro. Neste caso *Proc.lagura* deverá conter o valor 2 e *NRET* o valor 4.

No quadro 76 é mostrado o estado da pilha após cada execução das instruções do quadro 75. O *ESTADO 1* (quadro 76), é o estado em que a pilha se encontra logo após o chamada do procedimento *CALCULO*. Portanto, já estão na pilha o endereço de retorno para o procedimento chamador e os parâmetros *RESULTADO* e *NUMERO.* No *ESTADO 3* o registrador *SP* é decrementado em 2, que são os dois *bytes* necessários para a variável local *X*.

Os comandos deste procedimento são executados entre o *ESTADO 3* e o *ESTADO 4* (quadro 76), que com a instrução *MOV SP,BP* (quadro 75), devolve para SP seu valor original, automaticamente eliminado o espaço na pilha reservado para a variável local *X*. No *ESTADO 5*, *BP* é desempilhado recebendo o valor que possuía antes da chamada do procedimento.

QUADRO 74 – EXEMPLO DE DECLARAÇÃO DE PROCEDIMENTO NO FURBOL

```
PROCEDIMENTO CALCULO(REF R:INTEIRO;N:INTEIRO);
var x : inteiro;
INICIO
   |...
   Comandos
   |...
FIM;
```

QUADRO 75 – TRADUÇÃO EM *ASSEMBLY* DO QUADRO 74

```
CALCULO   PROC NEAR
PUSH  BP
MOV   BP,SP
SUB   SP,2
      |...
      Comandos
      |...

MOV   SP,BP
POP   BP
RET   4
CALCULO   ENDP
```

A instrução *RET 4* (quadro 75), retira da pilha o endereço de retorno que é onde está a próxima instrução a ser executada e o argumento 4 faz com que antes deste retorno sejam retirados da pilha os parâmetros *RESULTADO* e *NUMERO* que foram empilhados antes da chamada do procedimento.

QUADRO 76 – ESTADOS DA PILHA NA CHAMADA DE UM PROCEDIMENTO

ESTADO 1
PUSH BP
BP=42 SP=34

	28
	30
	32
BP=42	34
End. de Retorno	36
End. da var Resultado	38
Valor de NUMERO	40
BP=50	42

ESTADO 2
MOV BP,SP
BP=34 SP=34

	28
	30
	32
BP=42	34
End. de Retorno	36
End. da var Resultado	38
Valor de NUMERO	40
BP=50	42

ESTADO 3
SUB SP,2
BP=34 SP=32

	28
	30
RESERVA P/ X	32
BP=42	34
End. de Retorno	36
End. da var Resultado	38
Valor de NUMERO	40
BP=50	42

ESTADO 4
MOV SP,BP
BP=34 SP=34

	28
	30
RESERVA P/ X	32
BP=42	34
End. de Retorno	36
End. da var Resultado	38
Valor de NUMERO	40
BP=50	42

ESTADO 5
POP BP
BP=42 SP=36

	28
	30
	32
	34
End. de Retorno	36
End. Da var Resultado	38
Valor de NUMERO	40
BP=50	42

ESTADO 6
RET 4
BP=42 SP=42

	28
	30
	32
	34
	36
	38
	40
BP=50	42

3.2.4 ESTRUTURA DE COMANDOS

A definição das estruturas de comando para a linguagem FURBOL podem ser vistas nos quadros 77, 78, 79, 80, 81, 82 ,83 e 84.

QUADRO 77 – DEFINIÇÃO DA ESTRUTURA DE COMANDOS

Comando	::=	#ID,	
		Atribuição	EndVar:=Encontra_var(id.nome); se EndVar = nil entao erro; Atribuição.Tipo:=EndVar^..tipo; Comando.código:=Atribuição.código; Comando.CodAsm:=Atribuição.CodAsm;
		| ChamProc,	ChamProc.nome:=Id.nome; Comando.código:=ChamProc.código || CHAMADA ChamaProc.nome; Comando.códAsm:=ChamProc.códAsm || CALL ChamaProc.nome;
		Virgula	
		|	
		CCondicional,	
		Virgula	Comando.código:=CCondicional.código Comando.códAsm:=CCondicional.códAsm
		|	
		CRepetição,	
		Virgula	Comando.código:=CRepetição.código Comando.códAsm:=CRepetição.códAsm
		|	
		CEntrada,	
		Virgula	Comando.código:=CEntrada.código
		|	
		CSaida,	
		Virgula	Comando.código:=CSaida.código
		|	
		CInc,	Comando.código:=CInc.Código Comando.códAsm:=CInc.CódAsm
		Virgula	
		|	
		CDec,	Comando.código:=CDec.Código Comando.códAsm:=CDec.CódAsm
		Virgula	
		|	
		‘NL’,	
		Virgula	Comando.código:=CNL.código
		|	
		Virgula;	
Virgula	::=	‘;’ ,	
		Comando	Virgula.código:=Comando.código Virgula.códAsm:=Comando.códAsm
		|	
		^;	

A função *ehprocedimento* do quadro 77, verifica se o nome (ID) encontrado é ou não uma chamada de procedimento. Se for encontrado na tabela de símbolos um nome de procedimento igual ao nome procurado, *ehprocedimento* retorna *TRUE* (verdadeiro) caso contrário retorna *FALSE* (falso).

Na definição da estrutura de atribuição gera-se uma atribuição normal se *Atribuição.local* for um nome simples e uma atribuição indexada caso contrário.

QUADRO 78 – DEFINIÇÃO DA ESTRUTURA DE ATRIBUIÇÃO

Atribuição	::=	‘:=’,	Se Atribuição.deslocamento <> ‘’ então RX:=novo_t; IndCodAsm:='MOV RX,’ || Ldesloca || 'PUSH 'RX; PreIdAt:='POP DI';
		Expressão;	E.local := Expressão.local; se Expressao.tipo:<>Atribuicao.tipo entao erro; se Expressão.deslocamento :<> ‘ ’ então Atribuição.código:=gerar(idAT’[‘L.deslocamento ‘]:=’E.local); senão Atribuição.código:=gerar(IdAT ':=' E.Local); Atribuição.Asm:= IndCodAsm Expressao.CodAsm || AtPrelocal 'MOV RX,'aTlocal || PreIdAT || ‘MOV ‘IdAT','RX;

Na definição da chamada de procedimentos do quadro 79, cada instrução *PUSH* coloca na pilha um parâmetro para que este possa ser acessado pelo procedimento chamado, se o parâmetro for por referência (*REF*) será empilhado o endereço da variável caso contrário será empilhado apenas o valor atual da variável.

QUADRO 79 – DEFINIÇÃO DE CHAMADAS DE PROCEDIMENTOS

ChamProc	::=	‘(’	
		ParamAtuais,	chamProc.códAsm:= ParamAtuais.CodAsm;
		‘)’ | ^;	
ParamAtuais		#ID,	EndVar := encontra_var(ID.nome); se EndVar = nil entao erro EndPar:= encontra_Par(ChamaProc.nome); se EndVar^.Tipo <> EndPar^.tipo entao erro; se ParâmetroReferencia entao ParamAtuais.CodAsm := LEA DI,ID.nome || PUSH DI || ParamAtuais.codAsm ; Se ParâmetroValor entao ParamAtuais.CodAsm:= PUSH ID.nomeAsm || ParamAtuais.codAsm;
		|‘,’	
		ParamAtuais;	
		| ^;	

QUADRO 80 – DEFINIÇÃO DA ESTRUTURA DE COMANDOS DE REPETIÇÕES

Crepetição	::=	‘ENQUANTO’,	CRepetição.início:=novo_L; Expressão.v:=novo_L; Expressão.f:=CRepetição.próx;
		Expressao,	
		‘FACA’,	
		ComandoComposto;	CComposto.prox:=CRpetição.inicio; CRepetição.código:= CRepetição.inicio ‘:” || Expressão.código || E.v ‘:’ || CComposto.código || GOTO|| CRepetição.inicio; CRepetição.códiAsm:= CRepetição.inicio ‘:” || Expressão.códAsm ||E.v ‘:’ || CComposto.códAsm || JMP|| CRepetição.inicio;

QUADRO 81 – DEFINIÇÃO DA ESTRUTURA DE COMANDOS CONDICIONAIS

Ccondicional	::=	‘SE’,Expressão,’Então’,	
		CComposto,	$CComposto_1$.próx:=CCondicional.próx; CCondicional.código.:= Expressão.código || E.v ‘:’ || CComposto.código; CCondicional.códAsm.:= Expressão.códAsm || E.v ‘:’ || CComposto.códAsm;
		CCondicional2;	CCondicional.codigo:=CCondicional.Codigo || CCondicional2.codigo; CCondicional.codAsm:=CCondicional.CodAsm || CCondicional2.codAsm;
CCondicional2	::=	‘SENÃO’,	
		ComandoComposto,	fproximo:=proximo; CCondicional2.codigo := Expressao.codigo|| goto || proximo||f ‘:'|| CComposto.Codigo; CCondicional2.CodAsm:=Expressao.codAsm|| JMP || proximo||f ‘:'|| CComposto.CodAsm;
		|	
		^;	Ccondicional2.Codigo:= Ccondicional2.Codigo || f || ':’; Ccondicional2.Codasm:= Ccondicional2.Codasm || f || ':’;

QUADRO 82 – DEFINIÇÃO DA ESTRUTURA DO COMANDO DE ENTRADA

Centrada	::=	‘LEITURA’,	
		‘(‘,	
		#ID,	if (encontra_var(ID)=nil) then erro;
		LeituraListaId,	
		‘)’;	CEntrada.codigo:=‘LEITURA(’ID1’,’LeituraListaI D.codigo;
LeituraListaID	::=	‘,’,	
		#ID,	if encontra_var(ID)=nil then erro; LeituraListaID.codigo:= LeituraListaID.codigo’,’ID;
		LeituraListaId	
		|	
		^;	

QUADRO 83 – DEFINIÇÃO DA ESTRUTURA DO COMANDO DE SAÍDA

Csaida	::=	‘IMPRIME’,	
		‘(’,	
		Expressao,	
		ListaExpressao,	
		‘)’;	CSaida.código:=‘LEITURA(’Expressão.código||’,’|| ListaExpressao.codigo’;
ListaExpressao	::=	‘,’	
		Expressao,	ListaExpressao.codigo:= ListaExpressao.codigo || expressao.codigo;
		ListaExpressao	
		| ^;	

QUADRO 84 – DEFINIÇÃO DOS COMANDOS DE INCREMENTO E DECREMENTO

Cinc	::=	‘(’,	
		#ID,	se encontra_var(id.nome) = nil entao erro; se ID.tipo <> ‘inteiro’ entao erro; CInc.Codigo:= INC || ’(‘ || Id.nome || ’)’; CInc.CodAsm:= INC || id.nome;
		‘)’;	
Cdec	::=	‘(’,	
		#ID,	se encontra_var(id.nome) = nil entao erro; se ID.tipo <> ‘inteiro’ entao erro; CDec.Codigo:= DEC || ’(‘ || Id.nome || ’)’; CDec.CodAsm:= DEC || id.nome;
		‘)’;	

3.2.5 ESTRUTURA DE CONTROLE DE EXPRESSÕES

A estrutura de controle de expressões pode ser vista nos quadros 85, 86, 87, 88, 89 e 90. Esta definição foi construída a partir da definição de precedência de operadores definida em Aho (1995).

QUADRO 85 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES

Expressão	::=	Expressão2,	Relação.v:=Expressão2.v; Relação.f:=Expressão2.f; Relação.local:=Expressão2.local; Relação.codigo:=Expressão2.codigo; Relação.codAsm:=Expressão2.codAsm;
		Relação;	Expressão.local :=Relação.local; Expressão.código:=Relação.código; Expressão.códAsm:=Relação.códAsm; Expressão.v:=Relação.v Expressão.f:=Relação.f

QUADRO 86 - DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO)

Relação	::=	‘=’,	
		Expressão2	Relação.código:= ‘SE’ || Relação.local ‘=’ Expressão2.local ||‘goto’ E.v || ‘goto’ E.f; Relação.códAsm := MOV R0, Relacao.local || CMP R0, local || JE Rv || JMP Rf;
		|	
		‘<>’,	Relação.código:= ‘SE’ || Relação.local ‘<>’ Expressão2.local ||‘goto’ E.v || ‘goto’ E.f; Relação.códAsm := MOV R0, Relacao.local || CMP R0, local || JNE Rv || JMP Rf;
		|	
		‘<’,	Relação.código:= ‘SE’ || Relação.local ‘<’ Expressão2.local ||‘goto’ E.v || ‘goto’ E.f; Relação.códAsm := MOV R0, Relacao.local || CMP R0, local || JB Rv || JMP Rf;
		|	
		‘>’,	Relação.código:= ‘SE’ || Relação.local ‘>’ Expressão2.local ||‘goto’ E.v || ‘goto’ E.f; Relação.códAsm := MOV R0, Relacao.local || CMP R0, local || JA Rv || JMP Rf;
		^;	
Expressão2	::=	TC,	EL_i.v:=TC.v EL_i.f:=TC.f EL_i.local := TC.local EL_i.código := TC.código EL_i.códAsm := TC.códAsm
		EL;	Expressao2.local := EL_S.local Expressao2.código := EL_S.código Expressao2.códAsm := EL_S.códAsm

QUADRO 87 - DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO)

EL	::=	‘+’,	
		TC,	EL_{1i}.local:= novo_t; EL_{1i}.código:= EL.código|| TC.código || EL_{1i}.local ‘:=’ EL.local ‘+’ TC.local EL_{1i}.códigoAsm:= EL.código|| TC.código || MOV RX, EL.local || ADD RX, Tc.Local || MOV EL_{1i}.local , RX;
		EL_1	EL.local := EL_{1S}.local EL.código := EL_{1S}.código EL.códAsm := EL_{1S}.códAsm
		|	
		‘-‘,	
		TC,	EL_{1i}.local:= novo_t; EL_{1i}.código:= EL.código|| TC.código || EL_{1i}.local ‘:=’ EL.local ‘-’ TC.local; EL_{1i}.códigoAsm:= EL.código|| TC.código || MOV RX, EL.local || SUB RX, Tc.Local || MOV EL_{1i}.local ,RX;
		EL_1	EL.local := EL_{1S}.local EL.código := EL_{1S}.código EL.códAsm := EL_{1S}.códAsm
		|	
		^;	EL.v:= EL_S.v EL.f:= EL_S.f EL.local := EL_S.local EL.código := EL_S.código EL.códAsm := EL_S.códAsm
TC	::=	F,	TL_1.v := F.v TL_1.f := F.f TL_1.local := F.local TL_1.código := F.código TL_1.códAsm := F.códAsm
		TL;	T.v := TL_s.v T.f := TL_s.f TC.local := TL_s.local TC.códAsm := TL_s.códAsm TC.código := TL_s.código

QUADRO 88 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO)

TL	::=	‘*’,	
		F,	TL_1.local := novo_t TL_1.código := TL.código || F.código || TL1.local ‘:=’ TL.local ‘*’ F.local; TL_1.códAsm : = TL.código || F.código MOV AX,TLLOCAL || MUL Local || MOV TliLocal,AX;
		TL_1	TL.local := TL_{1s}.local TL.código := TL_{1s}.código TL.códAsm := TL_{1s}.códAsm
		|	
		‘/’,	
		F,	TL_1.local := novo_t TL_1.código := TL.código || F.código || TL1.local ‘:=’ TL.local ‘/’ F.local; TL_1.códigoAsm : = TL.código || F.código MOV AX,TLLOCAL || DIV Local || MOV TliLocal,AX;
		TL_1	TL.local := TL_{1s}.local TL.código := TL_{1s}.código TL.códAsm := TL_{1s}.códAsm
		|	
		‘E’,	
		F,	TL.v := Novo_L TL.f := TL_1.f F.v := TL_1.v F.f :=TL_1.f TL_1.código := TL.código || TL_1.v ‘:’ ||F.código TL_1.códAsm := TL.códAsm || TL_1.v ‘:’ ||F.códAsm
		TL_1	TL.v := TL_{1s}.v TL.f := TL_{1s}.f TL.códAsm := TL_{1s}.códAsm TL.código := TL_{1s}.código
		|	
		^;	TL.v := TL_S.v TL.f := TL_S.f TL.local := TL_S.local TL.código := TL_S.código TL.códAsm := TL_S.códAsm

QUADRO 89 – DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO)

F	::=	‘(‘	Expressão.v := F.v Expressão.f := F.f
		Expressão,	
		‘)’	F.local := Expressão.local; F.código := Expressão.código F.códAsm := Expressão.códAsm
		|	
		‘-‘,	
		Expressão	F.local := novo_t; F.código := Expressão.código || gerar(F.local ‘:=’ ‘ uminus ’ E.local)
		|	
		‘NAO’	Expressão.v := F.f; Expressão.f:= F.v;
		Expressão	
		|	
		L	se encontra_var(ID) então se L.deslocamento := 0 então F.local := L.local senão F.local := novo_t; gerar(F.local ’:=’ L.local ‘[‘ L.deslocamento ‘]’);
		|	
		NUM	F.local := NUM; F.código := ‘’; F.códAsm := ‘’;
		| ^;	
L	:=	Lista_E]	L.local := novo_t; L.deslocamento := novo_t; L.código := gerar(L.local := c(Lista_Earray); L.código := gerar(L.deslocamento ’:=’ Lista_E.local ‘*’ Largura(Lista_E.array)); L.CódAsm:=Lista_E.Códsm+ 'MOV DX,'+Lista_Elocal || 'MOV AL,'+IntToStr(VerificaTipoMatriz(Lista_Earray)) || 'MUL DX' || 'MOV '+LDesloca+',AX' || 'ADD '+LDesloca+','+IntToStr(Lc) || 'ADD '+LDesloca+',2';
		|	
		#ID	L.local := ID.local; L.deslocamento := ‘’;
Lista_E	::=	ID[Expressao,	R_i.matriz:=ID.local; R_i.local:=Expressao.local; R_i.ndim:=1;
		R	Lista_E.matriz:= R_s.matriz; Lista_E.local:= R_s.local; Lista_E.ndim:= R_s.ndim;

QUADRO 90– DEFINIÇÃO DA ESTRUTURA DE CONTROLE DE EXPRESSÕES (CONTINUAÇÃO)

R	::=	Expressao	t:=novo_t;
			m:=Ri..ndim+1;
			gerar(t+':='+Lista_E.local*
			limite(Lista_E.matriz,m);
			R_1i.matriz:=Ri.matriz;
			R_1i.local:=t;
			R_1i.ndim:=m;
			R_1i.CódAsm:=Expressao.CódAsm+
			'MOV CX,'+local ||
			'MOV AL,'+IntToStr(limite(L_Earray,m)) ||
			'MUL CX' ||
			'MOV '+t+',AX' ||
			'ADD '+t+','+Elocal;

		R_1	R_s.matriz:= R_{1s}.matriz; R_s.local:= R_{1s}.local; R_s.ndim:= R_{1s}.ndim; R_s.CódAsm:= R_{1s}.CódAsm;
		|	
		^	R_s.matriz:= R_i.matriz; R_s.local:= R_i.local; R_s.ndim:= R_i.ndim; R_s.CódAsm:= R_i.CódAsm;

Para suportar atribuições do tipo matriz *(array)* toda vez que aparecer um *id* a avaliação é feita através da estrutura L e R.

Geramos uma atribuição normal se *L* for um nome simples e uma atribuição indexada na localização denotada por *L* caso contrário (Aho, 1995).

4 APRESENTAÇÃO DO PROTÓTIPO

O objetivo da implementação deste protótipo foi demonstrar a construção de um compilador utilizando os métodos formais para definição de uma linguagem de programação. A linguagem de programação utilizada foi o FURBOL, anteriormente especificada na seção 3.2, a qual possui características semelhantes ao “Portugol” (Guimarães, 1985). Este ambiente herdou características dos ambientes já implementados por Silva (1993), Bruxel (1996), Radloff (1997), Schimt (1999) e André (2000).

O protótipo foi desenvolvido no ambiente de programação Delphi 5. O código para geração do ambiente foi escrito na linguagem *Object Pascal* do ambiente Delphi. O código escrito em Delphi 5 por André (2000) foi reutilizado.

As características gerais da linguagem FURBOL após a extensão continuaram como apresentado em André (2000).

A principal mudança proporcionadas por este trabalho é a implementação de *array’s* semi-estáticos na linguagem FURBOL.

4.1 CARACTERÍSTICAS DO AMBIENTE FURBOL APÓS EXTENSÃO

O ambiente FURBOL possui uma nova extensão. Esta nova extensão consiste na implementacão de mapeamento finito (*array’s*). Estes *array’s* estão definidos no ambiente FURBOL com o nome de *matriz* que podem ser do tipo inteiro. O quadro 91 mostra um exemplo de programa com declarações de variáveis do tipo *matriz* e o seu respectivo código assembler é descrito no quadro 92 e 93.

QUADRO 91 – EXEMPLO DE DECLARAÇÃO E USO DE *ARRAY'S* NO AMBIENTE FURBOL

```
programa xpto6;
Var
  A:matriz[1..2]:inteiro;
  B:matriz[1..2,1..3,1..5]:inteiro;
  i,j,k:inteiro;

Inicio
 k:=2;
 i:=1;
 j:=1;
 A[k]:=1;
 B[A[k],j*k,4]:=10;
fim.
```

QUADRO 92 – TRADUÇÃO EM CÓDIGO *ASSEMBLER* DO QUADRO 91

```
(1)  CODIGO     SEGMENT
(2)  ASSUME     CS:CODIGO, DS:CODIGO
(3)  ORG        100H
(4)  ENTRADA:     JMP    XPTO6
(5)  K      DW ?
(6)  I      DW ?
(7)  J      DW ?
(8)  B      DW 24 DUP (0)
(9)  A      DW 2 DUP (0)
(10)
(11) XPTO8  PROC NEAR
(12) PUSH  BP
(13) MOV   BP,SP
(14)
(15) MOV   AX,2
(16) MOV   K,AX        ;K=2
(17)
(18) MOV   AX,1
(19) MOV   I,AX        ;I=1;
(20)
(21) MOV   AX,1
(22) MOV   J,AX        ;J=1
(23)                   ;linha 20 à 26 – cálculo do índice do array A
(20) MOV   DI,K
(21) MOV   AX,2        ; linha 20 à 23 – primeiro termo da fórmula
(22) MUL   DI          ; (( ... (( i1 n2 + i2 ) n3 + i3 ) ... )inf1) nK + iK ) X w linfK ) X w
(23) MOV   DI,AX       ; + base -(( ... ((linf1 n2 + linf2 ) n3 + linf3) ... )nK + linfK ) X w
(24) ADD   DI,-2       ; -2 segundo termo da fórmula já calculado em tempo de compilação
(25) MOV   BX,DI
(26) PUSH  BX          ; armazena o cálculo do índice do array A na pilha
(27)
(28) MOV   AX,1
(29) POP   DI          ; retira da pilha o valor já calculado do índice
(30) MOV   A[DI],AX    ; move 1 p/ A[2], em assembler o array começa com índice 0
(31)
(32) MOV   DI,K        ;linha 32 à 57 – cálculo do índice do array B
(33) MOV   AX,2
(34) MUL   DI
(35) MOV   DI,AX
(36) ADD   DI,-2
(37) MOV   CX,A[DI]    ; CX=1
(38) MOV   SI,CX
(39) MOV   DI,K
(40) MOV   AX,J
(41) MUL   DI
(42) MOV   BX,AX       ;BX=2
(43) MOV   DI,SI
(44) MOV   AX,3
(45) MUL   DI
(46) MOV   DI,AX
(47) ADD   DI,BX
(48) MOV   SI,DI       ;SI=5
(49) MOV   DI,SI
(50) MOV   AX,5
(51) MUL   DI
(52) MOV   DI,AX
```

QUADRO 93 – TRADUÇÃO EM CÓDIGO *ASSEMBLER* DO QUADRO 91 (CONTINUAÇÃO)

```
(53)  ADD   DI,4
(54)  MOV   AX,2
(55)  MUL   DI
(56)  MOV   DI,AX
(57)  ADD   DI,-44
(58)  MOV   CX,DI          ; CX = 14
(59)  PUSH  CX             ; empilha CX (cálculo do índice do array B)
(60)  MOV   AX,10
(61)  POP   DI             ; desempilha para DI o cálculo do índice do array B
(62)  MOV   B[DI],AX       ; B[14] = 10
(63)  POP   BP
(64)  Int   20h
(65)  XPTO6   ENDP
(66)  CODIGO   ENDS
(67)  END   ENTRADA
```

Considerando a fórmula para calcular os índices dos *arrays* descrita no quadro 35, o cálculo do *array* A, que possui somente um índice com i_1=2 e w=2, gera a expressão do quadro 94 e o cálculo do array B, que possui 3 índices com i_1=1, n_2=3, i_2=1, n_3=5, i_3=4 e w=2 gera a expressão do quadro 95. Para estes casos (*array´s* A e B) está sendo considerado a base igual a zero (base=0) porque os *array´s* não estão armazenado na pilha. Os *array´s* só serão armazenados na pilha quando estiverem dentro de um procedimento. O uso da pilha para armazenar o cálculo dos índices dos *arrays* faz-se necessário porque pode ocorrer a atribuição de um *array* para outro *array*, o que ocasionaria o uso dos mesmos registradores antes da atribuição ser efetuada, gerando um erro.

QUADRO 94 – EXPRESSÃO GERADA PELO *ARRAY* A

2 X 2 + base -2 = 2

QUADRO 95 – EXPRESSÃO GERADA PELO *ARRAY* B

((1 X 3 + 2) X 5 + 4) X 2 + *base* – ((1 X 3 + 1) X 5 + 2) X 2 = 14

5 CONCLUSÃO

Os objetivos do referido trabalho foram atingidos, os quais eram a definição e implementação de mapeamento finito (*array's*). Estas novas inclusões ampliam a capacidade do compilador FURBOL, tornando-o mais abrangente.

A implementação de *array's* foi baseada em Aho (1995), através de métodos formais como BNF e gramática de atributos que foram essenciais para realização do mesmo.

Grandes dificuldades foram encontradas para inserção dos *array's* nas tabelas de símbolos, já que as referências bibliográficas citadas não concluiam este assunto..

Os array's podem ser usados tanto em expressões numéricas e atribuições, como também dentro do índice de outro *array*, limitando-se somente para estruturas do tipo inteiro.

Para gerar o código *Assembly*, além do método apresentado em AHO (1995), baseou-se também em programas compilados no Turbo Pascal 5.5 da Borland e verificado seus códigos de máquina gerados através do Turbo Debbuger da Borland.

5.1 EXTENSÕES

A estrutura do tipo *registro* não foi implementada neste trabalho, bem como comandos de entrada e saída de dados. Sendo estas implementações, sugestões para trabalhos futuros.

Outras sugestões para trabalhos futuros são a implementação de unidades do tipo função (*function* no Pascal) e a ampliação do uso de *array's* para outros tipos além do inteiro.

Também como sugestão, fazer a otimização código *assembly* gerado pelo protótipo e um gerador de código executável (montador) que tenha como saída um arquivo no formato *.EXE*.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V, SETHI, Ravi, ULMAN, Jeffrey D**. Compiladores, princípios, técnicas e ferramentas**. Tradução de Daniel de Ariosto Pinto. Rio de Janeiro: Livros Técnicos e Científicos, 1995.

ANDRÉ, Geovânio Batista. **Protótipo de gerador de código executável a partir do ambiente FURBOL.** Blumenau, 2000. 65 f. Trabalho de conclusão de curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

BRUXEL, Jorge Luiz. **Definição de um interpretador para a linguagem Portugol, utilizando gramática de atributos.** Blumenau, 1996. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

GHEZZI, Carlo, JAZAYERI, Mehdi. **Conceitos de linguagens de programação**. 2.ed. Rio de Janeiro: Campus, 1987.

GUIMARÃES, Ângelo de Moura; LAGES, Newton A. de Castilho. **Algoritmos e estruturas de dados.** Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora S.A., 1985.

HOLZNER, Steven. **Linguagem Assembly avançada para IBM PC**. São Paulo: McGraw-Hill, 1990.

JOSÉ NETO, João. **Introdução à compilação**. Rio de Janeiro: Livros Técnicos e Científicos, 1987.

KNUTH, Donald E. Semantic of context-free languages. **Mathematical systems theory**, New York, v. 2, n. 2, p. 33-50, jan./mar. 1968.

KOWALTOWISKI, Tomasz**. Implementação de linguagem de programação**. Rio de Janeiro: Guanabara Dois, 1983.

MORGAN, Christopher L.. **8086/8088 manual do microprocessador de 16 bits**. São Paulo: McGraw-Hill, 1988.

NELSON, Ross P.. **Programação assembly 80386**. São Paulo: McGraw-Hill, 1991.

ORVIS, William J.. **Visual Basic for applications, técnicas de programação**. Rio de Janeiro: Axcel Books, 1994.

QUADROS, Daniel G. A..**PC Assembler usando DOS**. Rio de Janeiro: Campus, 1988.

RADLOFF, Marcelo. **Protótipo de um ambiente para programação em uma linguagem bloco estruturada com vocabulário na língua portuguesa**. Blumenau, 1997. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

SANTOS, Jeremias René D. P. dos. **Programando em Assembler 8086/8088.** São Paulo: McGraw-Hill, 1989.

SANTOS, Jeremias René D. P. dos. **Turbo Assembler e Macro Assembler**. São Paulo: McGraw-Hill, 1990.

SCHIMT, Héldio. **Implementação de produto cartesiano e métodos de passagem de parâmetros no ambiente FURBOL**. 1999. 86 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SEBESTA, Robert W. **Conceitos de linguagens de programação**. 4. ed. Tradução de José Carlos Barbosa dos Santos. Porto Alegre: Bookman, 2000. 624 p.

SILVA, José Roque V. et al. Execução controlada de programas. In: Simpósio Brasileiro de Engenharia de Software, 1., 1987, Petrópolis. **Anais**... Petrópolis: UFRJ, 1987. p. 12-19.

SILVA, José Roque V. **Linguagens de Programação.** Roteiro de aula da disciplina de Linguagens de Programação do Curso de Ciências da Computação da Universidade Regional de Blumenau. Blumenau, 2000.

SILVA, Joilson Marcos da**.** Desenvolvimento de um ambiente de programação para a linguagem portugol, **Dynamis**, Blumenau, v. 1, n. 5, p. 99-114, dez. 1993.

SWAN, Tom. **Mastering turbo assembler**. Indianápolis: Hayden Books, 1989.

VARGAS, Douglas N. **Editor dirigido por sintaxe**. Relatório de pesquisa n. 240 arquivado na Pró-Reitoria de Pesquisa da Universidade Regional de Blumenau, Blumenau, set. 1992.

VARGAS, Douglas Nazareno**. Definição e implementação no ambiente windows de uma ferramenta para o auxílio no desenvolvimento de programas**. Blumenau, 1993. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

WATT, David A. **Programming language syntax and semantics**. New York: Prentice Hall, 1991.

YEUNG, Bik Chung. **8086/8088 Assembly language programming**. Great Britain: John Wiley & Sons, 1985.