

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
(Bacharelado)

**IMPLEMENTAÇÃO DE MAPEAMENTO FINITO (ARRAYS)
DINÂMICO 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

STEPHAN DIETER BIEGING

BLUMENAU, JUNHO/2002

2002/1-72

IMPLEMENTAÇÃO DE MAPEAMENTO FINITO (ARRAYS) DINÂMICO NO AMBIENTE FURBOL

STEPHAN DIETER BIEGING

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^a. Joyce Martins

Prof. Antonio Carlos Tavares

SUMÁRIO

LISTA DE FIGURAS	VIII
LISTA DE QUADROS	X
RESUMO	XVI
ABSTRACT	XVII
1 INTRODUÇÃO	1
1.1 OBJETIVOS DO TRABALHO	2
1.2 ESTRUTURA DO TRABALHO	2
2 FUNDAMENTAÇÃO TEÓRICA.....	4
2.1 COMPILADORES	4
2.1.1 ANÁLISE LÉXICA	5
2.1.2 ANÁLISE SINTÁTICA	6
2.1.2.1 Gramáticas Livres de Contexto	6
2.1.2.2 Análise Sintática <i>top-down</i>	8
2.1.2.3 Fatoração à esquerda.....	9
2.1.2.4 Eliminação da recursividade À esquerda.....	10
2.1.2.5 Árvores Gramaticais	11
2.1.3 ANÁLISE SEMÂNTICA	12
2.1.3.1 Definições Dirigidas por Sintaxe.....	13
2.1.3.1.1 Gramáticas de Atributos	13
2.1.3.1.1.1 Atributos Sintetizados.....	13
2.1.3.1.1.2 Atributos Herdados	14
2.1.3.1.1.3 Eliminação da Recursividade à Esquerda de um Esquema de Tradução	15
2.2 O CONCEITO DE AMARRAÇÃO.....	17

2.2.1 VARIÁVEL	18
2.2.1.1 Escopo de uma Variável	18
2.2.1.2 O Tempo de Vida de uma Variável	18
2.2.1.3 O Valor de uma Variável	19
2.2.1.4 O Tipo de uma Variável	19
2.2.2 UNIDADES DE PROGRAMAS	20
2.2.2.1 Unidades de Ativação	20
2.2.2.1.1 Registros de Ativação em Linguagens Estáticas	20
2.2.2.1.2 Registros de Ativação em Linguagens Dinâmicas.....	21
2.3 ALOCAÇÃO DE MEMÓRIA	22
2.3.1 ALOCAÇÃO DE MEMÓRIA EM LINGUAGENS ESTÁTICAS	22
2.3.1.1 Organização de Memória.....	23
2.3.2 ALOCAÇÃO DE MEMÓRIA EM LINGUAGENS DINÂMICAS	24
2.3.2.1 Alocação Implícita de Memória	25
2.3.2.2 Alocação Explícita de Memória	25
2.3.2.3 Organização da Memória.....	27
2.3.2.3.1 Memória de Pilha.....	27
2.3.2.3.2 Memória <i>Heap</i>	29
2.4 CLASSIFICAÇÃO DE VARIÁVEIS CONFORME PROCESSO DE ALOCAÇÃO.....	29
2.5 MAPEAMENTO FINITO (<i>ARRAYS</i>).....	30
2.5.1 <i>ARRAYS</i> E ÍNDICES	30
2.5.2 CATEGORIAS DE <i>ARRAYS</i>	31
2.5.3 QUANTIDADE DE SUBSCRITOS EM UM <i>ARRAY</i>	31
2.5.4 INICIALIZAÇÃO DE <i>ARRAYS</i>	32
2.5.5 IMPLEMENTAÇÃO DE <i>ARRAYS</i>	32

2.6 ACESSO A VARIÁVEIS NÃO-LOCAIS.....	35
2.6.1 ESCOPO ESTÁTICO COM PROCEDIMENTOS ANINHADOS.....	35
2.6.1.1 Regra do Aninhamento mais Interno.....	37
2.6.1.2 Profundidade de Aninhamento.....	37
2.6.1.3 Elos de Acesso.....	37
2.7 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO.....	40
2.7.1 CÓDIGO DE TRÊS ENDEREÇOS.....	40
2.7.2 TRADUÇÃO DIRIGIDA PELA SINTAXE DO CÓDIGO DE TRÊS ENDEREÇOS.....	41
2.8 GERAÇÃO DE CÓDIGO-ALVO.....	42
2.8.1 PROGRAMAS-ALVO.....	42
2.8.2 GERENCIAMENTO DE MEMÓRIA.....	42
2.8.3 SELEÇÃO DE INSTRUÇÕES.....	43
2.8.4 ALOCAÇÃO DE REGISTRADORES.....	45
2.8.5 MÁQUINA-ALVO.....	45
2.8.5.1 Arquitetura dos Microprocessadores 8088.....	45
2.8.5.1.1 Registradores de Uso Geral.....	46
2.8.5.1.2 Registradores de Segmento.....	47
2.8.5.1.3 Registrador dos sinalizadores.....	47
2.9 LINGUAGEM <i>ASSEMBLY</i>	48
2.9.1 CARACTERÍSTICAS GERAIS DO <i>ASSEMBLY</i>	48
2.9.2 INSTRUÇÕES DA LINGUAGEM <i>ASSEMBLY</i>	49
2.9.2.1 Instruções para transferência de dados.....	49
2.9.2.2 Instruções Aritméticas.....	50
2.9.2.3 Instruções Lógicas.....	51
2.9.2.4 Instruções para Controle de Fluxo.....	51

2.9.2.5	Instruções para Manipulação de Cadeias.....	54
2.9.3	SUBPROGRAMAS	55
2.10	SERVIÇOS DO SISTEMA OPERACIONAL DOS.....	56
2.10.1	FORMA DE CHAMAR OS SERVIÇOS DO DOS	56
2.10.2	SERVIÇOS DO DOS	56
2.10.2.1	Funções de Caractere.....	56
2.10.2.2	Funções de Arquivo/Diretório	57
2.10.2.3	Funções de Gerenciamento de Memória	58
2.10.2.3.1	Aloca memória (Função 48h)	59
2.10.2.3.2	Libera Bloco de Memória (Função 49h).....	59
2.10.2.3.3	Modifica Tamanho de Bloco de Memória (Função 4Ah).....	60
2.10.2.4	Funções de Gerenciamento de Programas.....	60
2.10.2.5	Outras funções	60
2.11	AMBIENTE FURBOL.....	61
2.11.1	PRINCIPAIS CARACTERÍSTICAS DA LINGUAGEM FURBOL.....	61
2.11.2	INTERFACE DO AMBIENTE FURBOL	62
3	DESENVOLVIMENTO DO TRABALHO	65
3.1	APRESENTAÇÃO GERAL DO AMBIENTE.....	65
3.2	DEFINIÇÃO DE ESCOPO DA LINGUAGEM.....	66
3.2.1	IMPLEMENTAÇÃO DE ELOS DE ACESSOS NA LINGUAGEM FURBOL.....	66
3.2.2	ESPECIFICAÇÃO DAS CLASSES DE CONTROLE DE ESCOPO	68
3.2.2.1	Gerenciador de Símbolos.....	71
3.2.2.2	Lista de Tabelas de Símbolos	72
3.2.2.3	Pilha de Tabelas de Símbolos	73
3.2.2.4	Tabela de Símbolos.....	73

3.2.2.5	Representação dos Símbolos	74
3.2.2.6	Informações sobre a dimensão de matrizes	75
3.3	MAPEAMENTO FINITO (<i>ARRAYS</i>) DINÂMICO NA LINGUAGEM FURBOL.....	76
3.3.1	PASSOS NECESSÁRIOS PARA A ALOCAÇÃO DINÂMICA NO DOS	76
3.3.2	DESCRITOR PARA <i>ARRAY</i> DINÂMICO	77
3.3.3	CONSTRUÇÕES PARA TRABALHAR COM <i>ARRAYS</i> DINÂMICOS	78
3.3.3.1	Declaração de Um <i>Array</i> Dinâmico.....	78
3.3.3.2	Comando <i>Redimensiona</i>	79
3.3.3.3	Função <i>Dimensoes</i>	80
3.3.3.4	Comando <i>LimInf</i>	80
3.3.3.5	Comando <i>LimSup</i>	81
3.3.3.6	Acesso aos Elementos do <i>Array</i> Dinâmico	81
3.3.4	CÓDIGO GERADO PELA IMPLEMENTAÇÃO DO <i>ARRAY</i> DINÂMICO.....	82
3.4	ESPECIFICAÇÃO DA LINGUAGEM FURBOL	90
3.4.1	PROGRAMAS E BLOCOS.....	90
3.4.2	ESTRUTURA DE DADOS	91
3.4.3	ESTRUTURA DE SUBPROGRAMAS	93
3.4.4	ESTRUTURA DE COMANDOS	95
3.4.5	ESTRUTURA DE EXPRESSÕES	101
3.5	APRESENTAÇÃO DO PROTÓTIPO.....	108
3.5.1	PRINCIPAIS CARACTERÍSTICAS DO AMBIENTE APÓS A EXTENSÃO.....	109
3.5.2	INTERFACE DO PROTÓTIPO	110
4	CONCLUSÕES	112
4.1	EXTENSÕES	113
	REFERÊNCIAS BIBLIOGRÁFICAS	114

LISTA DE FIGURAS

FIGURA 1 - UM COMPILADOR	4
FIGURA 2 - UMA ÁRVORE GRAMATICAL	11
FIGURA 3 - UMA ÁRVORE SINTÁTICA	11
FIGURA 4 - ARMAZENAMENTO ESTÁTICO PARA OS OBJETOS ASSOCIADOS A IDENTIFICADORES LOCAIS NUM PROGRAMA FORTRAN 77	24
FIGURA 5 - LISTAS LIGADAS CONSTRUÍDAS PELO PROGRAMA DO QUADRO 5	26
FIGURA 6 - ACESSO A ARRAYS ABERTOS ALOCADOS DINAMICAMENTE	28
FIGURA 7 - DISPOSIÇÕES PARA UM ARRAY BIDIMENSIONAL.....	33
FIGURA 8 - ELOS DE ACESSO PARA SE ENCONTRAR O ARMAZENAMENTO DE NOMES NÃO-LOCAIS	38
FIGURA 9 - UNIDADE CENTRAL DE PROCESSAMENTO DO MICROPROCESSADOR 8088.....	46
FIGURA 10 - TELA PRINCIPAL DO AMBIENTE FURBOL.....	62
FIGURA 11 - DETECÇÃO DE ERROS NO PROTÓTIPO.....	63
FIGURA 12 - JANELA PRINCIPAL COM CÓDIGO INTERMEDIÁRIO.....	63
FIGURA 13 - JANELA PRINCIPAL COM CÓDIGO ASSEMBLY	64
FIGURA 14 - DIAGRAMA GERAL DO AMBIENTE	66
FIGURA 15 - DIAGRAMA DE CASO DE USO PARA O CONTROLE DE ESCOPO	69
FIGURA 16 - DIAGRAMA DE SEQÜÊNCIA PARA O PROCESSO <i>INSTALAR SÍMBOLOS</i>	69
FIGURA 17 - DIAGRAMA DE CLASSES PARA O CONTROLE DE ESCOPO.....	70
FIGURA 18 - ESPECIFICAÇÃO DA CLASSE <i>TGERENCIADORSIMBOLOS</i>	71
FIGURA 19 - ESPECIFICAÇÃO DA CLASSE <i>TLISTATABELAS</i>	72
FIGURA 20 - ESPECIFICAÇÃO DA CLASSE <i>TPILHATABELAS</i>	73
FIGURA 21 - ESPECIFICAÇÃO DA CLASSE <i>TTABELASIMBOLOS</i>	74
FIGURA 22 - ESPECIFICAÇÃO DA CLASSE <i>TSIMBOLO</i>	75
FIGURA 23 - ESPECIFICAÇÃO DA CLASSE <i>TDIMENSÃO</i>	76
FIGURA 24 - DESCRITOR DE UM ARRAY DINÂMICO.....	77
FIGURA 25 - TELA DE EDIÇÃO DE PROGRAMAS-FONTE DO AMBIENTE FURBOL	111

FIGURA 26 - TELA DE APRESENTAÇÃO DO CÓDIGO *ASSEMBLY* GERADO..... 111

LISTA DE QUADROS

QUADRO 1 - PRODUÇÃO GRAMATICAL REPRESENTANDO UM ENUNCIADO CONDICIONAL.....	7
QUADRO 2 - PRODUÇÕES PARA UM ANALISADOR SINTÁTICO PREDITIVO.....	8
QUADRO 3 - UMA PRODUÇÃO RECURSIVA À ESQUERDA	8
QUADRO 4 - EXEMPLO DE PRODUÇÃO QUE PODE SER FATORADA.....	9
QUADRO 5 - PRODUÇÃO ANTES DA FATORAÇÃO.....	9
QUADRO 6 - PRODUÇÕES APÓS FATORAÇÃO	9
QUADRO 7 - PRODUÇÃO RECURSIVA À ESQUERDA.....	10
QUADRO 8 - PRODUÇÕES PARA ELIMINAÇÃO DA RECURSÃO À ESQUERDA.....	10
QUADRO 9 - AGRUPAMENTO DAS PRODUÇÕES-A.....	10
QUADRO 10 - PRODUÇÕES NÃO RECURSIVAS	10
QUADRO 11 - DEFINIÇÃO S-ATRIBUÍDA PARA UMA CALCULADORA DE MESA SIMPLES	14
QUADRO 12 - DEFINIÇÃO DIRIGIDA PELA SINTAXE COM ATRIBUTOS HERDADOS	15
QUADRO 13 - ESQUEMA DE TRADUÇÃO COM UMA GRAMÁTICA RECURSIVA À ESQUERDA	16
QUADRO 14 - ESQUEMA DE TRADUÇÃO TRANSFORMADO COM GRAMÁTICA RECURSIVA À DIREITA.....	16
QUADRO 15 - ESQUEMA DE TRADUÇÃO ABSTRATO RECURSIVO À ESQUERDA	16
QUADRO 16 - GRAMÁTICA SEM RECURSIVIDADE À ESQUERDA.....	17
QUADRO 17 - ESQUEMA DE TRADUÇÃO SEM RECURSIVIDADE À ESQUERDA ...	17
QUADRO 18 - DEFINIÇÃO DE UMA CONSTANTE EM PASCAL	19
QUADRO 19 - DECLARAÇÃO DE TIPO EM PASCAL.....	19
QUADRO 20 - UM PROGRAMA FORTRAN 77	23
QUADRO 21 - UMA ATRIBUIÇÃO DE UMA CADEIA DE CARACTERES EM C, ONDE A MEMÓRIA É ALOCADA IMPLÍCITAMENTE	25
QUADRO 22 - EXEMPLO DE ATRIBUIÇÃO DE <i>STRING</i> EM OBJECT PASCAL.....	25
QUADRO 23 - ALOCAÇÃO DINÂMICA DE CÉLULAS USANDO <i>NEW</i> EM PASCAL .	26
QUADRO 24 - PSEUDOCÓDIGO DE UMA UNIDADE COM <i>ARRAYS</i> ABERTOS	28

QUADRO 25 - DECLARAÇÃO E INICIALIZAÇÃO DE ARRAYS EM FORTRAN 77.....	32
QUADRO 26 - INICIALIZAÇÃO DE ARRAYS EM ANSI C	32
QUADRO 27 - CÁLCULO DO IÉSIMO ELEMENTO DE UM ARRAY	33
QUADRO 28 - EXPRESSÃO DE CÁLCULO DE LOCALIZAÇÃO DE ELEMENTOS EM TEMPO DE COMPILAÇÃO	33
QUADRO 29 - EXPRESSÃO PARA CÁLCULO DE ENDEREÇOS RELATIVOS DE ELEMENTOS DE UM ARRAY BIDIMENSIONAL	34
QUADRO 30 - EXPRESSÃO PARA CÁLCULO DE ENDEREÇOS RELATIVOS DE ELEMENTOS DE UM ARRAY BIDIMENSIONAL EM TEMPO DE COMPILAÇÃO	34
QUADRO 31 - EXPRESSÃO GENÉRICA PARA CÁLCULO DE ENDEREÇOS RELATIVOS DE ARRAYS MULTIDIMENSIONAIS	34
QUADRO 32 - PROGRAMA PASCAL COM PROCEDIMENTOS ANINHADOS	36
QUADRO 33 - INDENTAÇÃO MOSTRANDO O ANINHAMENTO DE PROCEDIMENTOS DO QUADRO 32.....	36
QUADRO 34 - UM ENUNCIADO DE CÓDIGO DE TRÊS ENDEREÇOS	40
QUADRO 35 - CÓDIGO DE TRÊS ENDEREÇOS PARA A EXPRESSÃO $X + Y * Z$	40
QUADRO 36 - DEFINIÇÃO DIRIGIDA PELA SINTAXE PARA PRODUZIR CÓDIGO DE TRÊS ENDEREÇOS PARA ATRIBUIÇÕES	41
QUADRO 37 - EXEMPLO DE ENTRADA PARA O GERADOR DE CÓDIGO DE TRÊS ENDEREÇOS.....	42
QUADRO 38 - CÓDIGO GERADO PELO GERADOR DE CÓDIGO INTERMEDIÁRIO PARA A ENTRADA DO QUADRO 37.....	42
QUADRO 39 - EXEMPLO DE DECLARAÇÃO DE VARIÁVEIS EM PASCAL.....	43
QUADRO 40 - ENUNCIADO DE TRÊS ENDEREÇOS	43
QUADRO 41 - CÓDIGO GERADO PARA O ENUNCIADO DE TRÊS ENDEREÇOS DO QUADRO 40	43
QUADRO 42 - ENUNCIADOS DE TRÊS ENDEREÇOS	44
QUADRO 43 - CÓDIGO INEFICIENTE GERADO PARA A SEQÜÊNCIA DE ENUNCIADOS DE TRÊS ENDEREÇOS DO QUADRO 42.....	44
QUADRO 44 - INCREMENTO SIMPLES DE UMA VARIÁVEL.....	44

QUADRO 45 - GERAÇÃO DE CÓDIGO PARA O QUADRO 44 UTILIZANDO A INSTRUÇÃO <i>INC</i>	44
QUADRO 46 - GERAÇÃO DE CÓDIGO PARA O QUADRO 44 UTILIZANDO VÁRIAS INSTRUÇÕES.....	45
QUADRO 47 - FORMATO GERAL DE UMA LINHA DE COMANDO EM <i>ASSEMBLY</i> ..	48
QUADRO 48 - EXEMPLO DE PROGRAMA <i>ASSEMBLY</i>	49
QUADRO 49 - INSTRUÇÃO DE MOVIMENTAÇÃO DE DADOS	49
QUADRO 50 - INSTRUÇÕES PARA MOVIMENTAÇÃO DE ENDEREÇOS	50
QUADRO 51 - INSTRUÇÕES PARA ENTRADA E SAÍDA DE DISPOSITIVOS	50
QUADRO 52 - INSTRUÇÕES ARITMÉTICAS	51
QUADRO 53 - INSTRUÇÕES LÓGICAS.....	51
QUADRO 54 - INSTRUÇÕES PARA DESVIO INCONDICIONAIS.....	52
QUADRO 55 - INSTRUÇÕES DE DESVIO CONDICIONAL	52
QUADRO 56 - INSTRUÇÕES DE DESVIO CONDICIONAL (CONTINUAÇÃO DO QUADRO 55).....	53
QUADRO 57 - INSTRUÇÕES DE DESVIO PARA INTERRUPÇÕES.....	53
QUADRO 58 - INSTRUÇÕES DE MANIPULAÇÃO DE CADEIAS	54
QUADRO 59 - PREFIXOS DE REPETIÇÃO DE INSTRUÇÕES	55
QUADRO 60 - FORMA GERAL DE UM SUBPROGRAMA NA LINGUAGEM <i>ASSEMBLY</i>	55
QUADRO 61 - PROGRAMA <i>ASSEMBLY</i> QUE USA SERVIÇOS DO DOS.....	56
QUADRO 62 - FUNÇÕES DE CARACTERES DO DOS	57
QUADRO 63 - FUNÇÕES DE ARQUIVO/DIRETÓRIO.....	57
QUADRO 64 - FUNÇÕES PARA ARQUIVO/DIRETÓRIO (CONTINUAÇÃO DO QUADRO 63).....	58
QUADRO 65 - FORMATO DE UM BLOCO DE CONTROLE DE MEMÓRIA NO DOS..	59
QUADRO 66 - FUNÇÕES DE GERENCIAMENTO DE PROGRAMAS DO DOS.....	60
QUADRO 67 - OUTRAS FUNÇÕES DO DOS.....	61
QUADRO 68 - PROGRAMA FURBOL COM ACESSO A VARIÁVEIS NÃO-LOCAIS ...	67
QUADRO 69 - CÓDIGO <i>ASSEMBLY</i> GERADO PARA O PROGRAMA DO QUADRO 68	68
QUADRO 70 - CÓDIGO PARA DIMINUIR A MEMÓRIA ALOCADA PELO DOS.....	77

QUADRO 71 - SINTAXE PARA A DECLARAÇÃO DE UM ARRAY DINÂMICO	79
QUADRO 72 - EXEMPLO DE DECLARAÇÃO DE ARRAY DINÂMICO	79
QUADRO 73 - SINTAXE PARA O COMANDO <i>REDIMENSIONA</i>	79
QUADRO 74 - EXEMPLO DE REDIMENSIONAMENTO DE ARRAY DINÂMICO	79
QUADRO 75 - EXEMPLO DE REDIMENSIONAMENTO DE ARRAY DINÂMICO ONDE OS LIMITES SÃO EXPRESSÕES.....	80
QUADRO 76 - SINTAXE DA FUNÇÃO <i>DIMENSOES</i>	80
QUADRO 77 - EXEMPLO DE CÓDIGO COM A FUNÇÃO <i>DIMENSOES</i>	80
QUADRO 78 - SINTAXE PARA A FUNÇÃO <i>LIMINF</i>	80
QUADRO 79 - EXEMPLO DE CÓDIGO COM A FUNÇÃO <i>LIMINF</i>	81
QUADRO 80 - SINTAXE DA FUNÇÃO <i>LIMSUP</i>	81
QUADRO 81 - EXEMPLO DE CÓDIGO COM A FUNÇÃO <i>LIMSUP</i>	81
QUADRO 82 - EXEMPLO DE ACESSO AOS ELEMENTOS DE UM ARRAY DINÂMICO	82
QUADRO 83 - CÓDIGO GERADO PARA A DECLARAÇÃO DO QUADRO 72.....	82
QUADRO 84 - PROGRAMA QUE USA O COMANDO <i>REDIMENSIONA</i>	82
QUADRO 85 - CÓDIGO GERADO PARA O PROGRAMA DO QUADRO 84	83
QUADRO 86 - CÓDIGO GERADO PARA O PROGRAMA DO QUADRO 84 (CONTINUAÇÃO DO QUADRO 85)	84
QUADRO 87 - CÓDIGO GERADO PELO PROGRAMA DO QUADRO 84 (CONTINUAÇÃO DO QUADRO 86)	85
QUADRO 88 - CÓDIGO GERADO PELO PROGRAMA DO QUADRO 84 (CONTINUAÇÃO DO QUADRO 87)	86
QUADRO 89 - FÓRMULA DO CÁLCULO DO PRIMEIRO ELEMENTO DO ARRAY DINÂMICO	87
QUADRO 90 - PROGRAMA QUE UTILIZA A FUNÇÃO <i>DIMENSOES</i>	87
QUADRO 91 - CÓDIGO GERADO PARA A FUNÇÃO <i>DIMENSOES</i> DO QUADRO 90..	87
QUADRO 92 - PROGRAMA QUE UTILIZA A FUNÇÃO <i>LIMINF</i>	87
QUADRO 93 - CÓDIGO GERADO PARA A FUNÇÃO <i>LIMINF</i>	88
QUADRO 94 - PROGRAMA FURBOL COM ATRIBUIÇÃO A UM ELEMENTO DE UM ARRAY DINÂMICO	88

QUADRO 95 - CÓDIGO GERADO PELA ATRIBUIÇÃO DO PROGRAMA DO QUADRO 94	89
QUADRO 96 - CÓDIGO GERADO PARA A LIBERAÇÃO DE UM ARRAY DINÂMICO <i>M</i>	90
QUADRO 97 - DEFINIÇÃO DE PROGRAMAS E BLOCOS.....	91
QUADRO 98 - DEFINIÇÃO DE ESTRUTURAS DE DADOS	92
QUADRO 99 - PRODUÇÕES PARA A DECLARAÇÃO DE <i>ARRAYS</i>	92
QUADRO 100 - DEFINIÇÃO DE ESTRUTURAS DE SUBPROGRAMAS	94
QUADRO 101 - DEFINIÇÃO DA ESTRUTURA DE PARÂMETROS FORMAIS.....	95
QUADRO 102 - DEFINIÇÃO DE ESTRUTURA DE COMANDOS	96
QUADRO 103 - DEFINIÇÃO DA ESTRUTURA DE ATRIBUIÇÃO.....	97
QUADRO 104 - DEFINIÇÃO DA ESTRUTURA DE CHAMADA DE PROCEDIMENTOS	97
QUADRO 105 - DEFINIÇÃO DA ESTRUTURA DE COMANDO DE REPETIÇÃO	98
QUADRO 106 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS CONDICIONAIS.....	98
QUADRO 107 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS DE ENTRADA	99
QUADRO 108 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS DE SAÍDA.....	99
QUADRO 109 - DEFINIÇÃO DA ESTRUTURA DE INCREMENTO E DECREMENTO	99
QUADRO 110 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS PARA <i>ARRAYS</i> DINÂMICOS.....	100
QUADRO 111 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS PARA <i>ARRAYS</i> DINÂMICOS (CONTINUAÇÃO DO QUADRO 110).....	101
QUADRO 112 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES.....	101
QUADRO 113 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 112).....	102
QUADRO 114 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO	103
QUADRO 115 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 114).....	103
QUADRO 116 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 115).....	104

QUADRO 117 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 116).....	105
QUADRO 118 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 117).....	106
QUADRO 119 - DEFINIÇÃO DAS ESTRUTURAS PARA AS FUNÇÕES DE ARRAYS DINÂMICOS.....	107
QUADRO 120 - DEFINIÇÃO DAS ESTRUTURAS PARA AS FUNÇÕES DE ARRAYS DINÂMICOS (CONTINUAÇÃO DO QUADRO 119).....	108
QUADRO 121 - EXEMPLO DE PROGRAMA FURBOL QUE UTILIZA ARRAY DINÂMICO.....	109
QUADRO 122 - EXEMPLO DE PROGRAMA FURBOL QUE ACESSA VARIÁVEIS NÃO-LOCAIS.....	110

RESUMO

Este trabalho descreve o desenvolvimento de um protótipo de um ambiente de programação com uma linguagem bloco-estruturada com vocabulário na língua portuguesa. O trabalho é baseado no trabalho de conclusão de curso de Anderson Adriano (Adriano, 2001), estendendo-o através da inclusão de novas construções, como *arrays* dinâmicos e acesso a variáveis não-locais, e geração de código executável para estas construções. Para a especificação formal da sintaxe é utilizada a notação BNF (*Backus-Naur Form*) e para a semântica é utilizada a gramática de atributos.

ABSTRACT

This assignment describes the development of a prototype of a block structured programming language environment with vocabulary in the portuguese language. The work is based on the work of conclusion of course from Anderson Adriano (Adriano, 2001), extending it through the inclusion of new constructions like dynamic *arrays* and non-local variable access and generating executable code for those constructions. For the formal specification of the syntax, the BNF (*Backus-Naur Form*) notation is used and for the formal specification of the semantics, the attribute grammar is used.

1 INTRODUÇÃO

O ambiente FURBOL teve sua origem no ano de 1987, através do trabalho “Execução Controlada de Programas” (Silva, 1987) apresentado no I Simpósio de Engenharia de Software.

Após este passo inicial, outros trabalhos deram continuidade ao ambiente FURBOL, como os trabalhos de Vargas (1992) intitulado “Editor Dirigido por Sintaxe” e de Silva (1993) desenvolvido no primeiro semestre do ano de 1993, com o título “Desenvolvimento de um Ambiente de Programação para a Linguagem Portugol”.

Também no ano de 1993, o acadêmico Douglas Nazareno Vargas apresentou um trabalho na forma de Trabalho de Conclusão de Curso (TCC) com o título “Definição e Implementação no Ambiente Windows de uma Ferramenta para o Auxílio no Desenvolvimento de Programas” (Vargas, 1993).

Em 1996 a linguagem sofreu uma redefinição através do uso de gramática de atributos e passou a se chamar FURBOL. Esta redefinição foi realizada no trabalho “Definição de um Interpretador para a Linguagem Portugol utilizando Gramática de Atributos” (Bruxel, 1996).

O trabalho “Protótipo de um Ambiente para Programação em uma Linguagem Bloco Estruturada com Vocabulário na Língua Portuguesa” (Radloff, 1997) implementou vários recursos novos no ambiente. Alguns destes recursos foram as chamadas de procedimento, recursividade, melhoria da interface, geração de código para a Máquina de Execução para Pascal (MEPA) proposta por Kowaltowski (1983), entre outras.

Em 1999, o TCC com o título “Implementação de Produto Cartesiano e Métodos de Passagem de Parâmetros no Ambiente FURBOL” (Schimt, 1999) estendeu o trabalho de Radloff (1997) com a implementação do produto cartesiano e métodos para a passagem de parâmetros (cópia-valor e referência). Neste trabalho foram utilizadas a notação *Backus-Naur Form* (BNF) e gramática de atributos para a especificação formal da linguagem.

A geração de código executável foi implementada no trabalho “Protótipo do Gerador Executável a partir do ambiente FURBOL” (André, 2000). Este trabalho inclui a opção de visualização do código intermediário e do código *Assembly* gerados.

O trabalho “Implementação de Mapeamento Finito (*Arrays*) no Ambiente FURBOL” (Adriano, 2001), tornou possível a definição e utilização de *arrays* com limites fixos no ambiente.

O presente trabalho utiliza a especificação do ambiente FURBOL apresentada em Adriano (2001), estendendo-a através da implementação de *arrays* dinâmicos. Também é implementado o acesso a variáveis não-locais que atualmente não está presente na especificação.

A implementação do protótipo foi feita no ambiente Borland Delphi 5 (Cantu, 2000), visto que este facilita a criação de programas devido a vasta biblioteca de componentes disponível.

O conhecimento sobre compiladores e a continuidade de um ambiente que vem tendo seu valor agregado através de trabalhos de conclusão de curso deram motivação para a realização do presente trabalho.

1.1 OBJETIVOS DO TRABALHO

Este trabalho visa estender a definição formal na linguagem de programação com vocabulário na língua portuguesa FURBOL, apresentada por Adriano (2001). Esta extensão consiste na definição e implementação de *arrays* dinâmicos e acesso a variáveis não-locais. A atual especificação foi estendida para possibilitar a definição e manipulação de *arrays* e para o acesso a variáveis não-locais declaradas em procedimentos aninhados. A geração de código executável para tais construções também foi implementada.

1.2 ESTRUTURA DO TRABALHO

O presente capítulo apresenta a introdução do trabalho, os objetivos e a organização do mesmo.

O capítulo 2 apresenta a fundamentação teórica com uma breve descrição sobre os conceitos relacionados a compiladores e sobre as técnicas utilizados para construir os três módulos que o compõem (análises léxica, sintática e semântica). Neste capítulo também serão

apresentados conceitos básicos sobre escopo, alocação de memória em linguagens estáticas e dinâmicas e *arrays*.

O capítulo 3 apresenta o desenvolvimento do protótipo com a descrição das técnicas utilizadas para a implementação de escopos e *arrays* dinâmicos. Também neste capítulo é apresentada a especificação do protótipo da linguagem FURBOL.

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

2 FUNDAMENTAÇÃO TEÓRICA

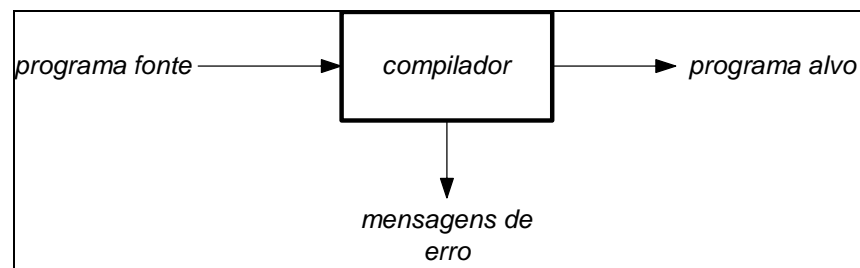
Neste capítulo serão apresentados os estudos relativos ao desenvolvimento do trabalho. Primeiramente serão apresentados os conceitos básicos relacionados a compiladores. Após serão vistos conceitos sobre o gerenciamento de variáveis e unidades de programa dentro das linguagens de programação. A seguir será apresentada a teoria que envolve a alocação de memória nas linguagens estáticas e dinâmicas. O mapeamento finito (*array*) será apresentado posteriormente, mencionando a organização e o acesso aos elementos do mesmo. Ainda serão vistos conceitos sobre a implementação do acesso a variáveis não-locais, linguagem *Assembly*, arquitetura do microprocessador 8088, serviços do sistema operacional DOS e geração de código-alvo.

2.1 COMPILADORES

Um 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.

Posto de forma simples, um compilador é um programa que lê um programa escrito numa linguagem e o traduz num programa equivalente numa outra linguagem, como mostrado na fig. 1. Como parte importante deste processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte (Aho, 1995).

FIGURA 1 - UM COMPILADOR



Fonte: Aho (1995)

Existem milhares de linguagens fontes, que vão das linguagens de programação tradicionais às linguagens especializadas que emergiram virtualmente em quase todas as áreas

de aplicação de compiladores. As linguagens alvos são igualmente variadas. Uma linguagem alvo pode ser uma outra linguagem de programação ou uma linguagem de máquina, que pode ser executada em um microprocessador ou até em um supercomputador (Aho, 1995).

Quando um programa-fonte é submetido a um compilador são realizadas as análises léxica, sintática e semântica, as quais serão descritas a seguir.

2.1.1 ANÁLISE LÉXICA

Segundo José (1987), a análise léxica é 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 função fundamental é a de, a partir do texto-fonte de entrada, fragmentá-lo em seus componentes básicos, identificando trechos elementares completos e com identidade própria, porém individuais para efeito de análise por parte dos demais programas do compilador (José, 1987).

Algumas das funções da análise léxica são (José, 1987):

- a) *extração e classificação dos átomos*: função primordial da análise léxica, transforma o texto-fonte em um outro texto formado pelos átomos que os símbolos componentes do texto-fonte representam;
- b) *eliminação de delimitadores e comentários*: os espaços em branco, símbolos separadores e comentários aumentam muito a legibilidade do programa, porém nas demais fases do compilador não são necessários. Desta forma, eles são retirados na análise léxica;
- c) *identificação de palavras reservadas*: palavras reservadas são tipos especiais de identificadores. Têm significados pré-determinados pela linguagem e ajudam na estruturação do texto-fonte. A análise léxica pode determinar e classificar as palavras reservadas e transmitir esta informação para as demais partes do compilador.

2.1.2 ANÁLISE SINTÁTICA

O analisador sintático é o segundo grande componente dos compiladores, que pode ser caracterizado como o mais importante, por sua característica de controlar as atividades do compilador. Sua função principal é a de promover a análise da seqüência com que os átomos componentes do texto-fonte apresentam-se (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 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.

Algumas funções da análise sintática são (José, 1987):

- a) identificação de sentenças;
- b) detecção, recuperação e correção de erros de sintaxe;
- c) ativação do analisador léxico;
- d) ativação de rotinas de análise linguagem;
- e) ativação de rotinas de análise semântica;
- f) ativação de rotinas de geração do código objeto.

A tônica da função dos analisadores sintáticos, em relação ao programa-fonte, nos compiladores de linguagens de alto nível, consiste em efetuar a análise e o reconhecimento das construções sintáticas mais complexas dessas linguagens. Na grande maioria dos casos estas construções são definidas através de gramáticas livres de contexto (José, 1987).

2.1.2.1 GRAMÁTICAS LIVRES DE CONTEXTO

José (1987) define gramáticas livres de contexto como sendo 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 $A ::= \alpha$, onde $A \in N$, $\alpha \in V^*$, 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 α é a sentença pela qual A deve ser substituído ao ser

aplicada esta regra de substituição, independentemente do contexto em que A está imerso. Daí o nome “livre de contexto” aplicado às gramáticas que obedecem a esta restrição.

Segundo Aho (1995) muitas construções de linguagens de programação possuem uma estrutura inerentemente recursiva que pode ser identificada por gramáticas livres de contexto. Para exemplificar será usado o enunciado definido pela regra: se S_1 e S_2 são enunciados e E é uma expressão, então “*if E then S₁ else S₂*” é um enunciado. Usando a variável sintática cmd para denotar a classe de comandos e $expr$ para a classe de expressões, pode-se prontamente expressar o enunciado anterior usando a produção gramatical do quadro 1.

QUADRO 1 - PRODUÇÃO GRAMATICAL REPRESENTANDO UM ENUNCIADO CONDICIONAL

$cmd ::= \mathbf{if} \ expr \ \mathbf{then} \ cmd \ \mathbf{else} \ cmd$
--

Fonte: adaptado de Aho (1995)

Segundo Aho (1995), uma gramática livre de contexto é definida por $G = (T, N, S, P)$ onde:

- a) T representa os terminais. Estes são os símbolos básicos a partir dos quais as sentenças são formadas. No quadro 1 cada palavra (*if*, *then*, *else*) é um terminal;
- b) N representa os não-terminais. Estes são variáveis sintáticas que denotam cadeias de caracteres. No quadro 1 cmd e $expr$ são não-terminais. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática. Também impõem uma estrutura hierárquica na linguagem que é útil tanto para análise sintática quanto para a tradução;
- c) S representa o símbolo de partida. Numa gramática, um não-terminal é distinguido como o símbolo de partida, e o conjunto de sentenças que o mesmo denota é a linguagem definida pela gramática;
- d) P representa as produções. As produções de uma gramática especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar estruturas. Cada produção consiste em um não-terminal, seguido pelo símbolo $::=$ e uma sentença de não-terminais e terminais.

Para se fazer a análise de gramáticas livres de contexto pode ser usada a análise sintática *top-down* ou a análise sintática *bottom-up*. Neste trabalho será utilizada a análise

sintática *top-down*, visto que o projeto atual da linguagem FURBOL já faz uso desta técnica. Mais detalhes sobre a análise sintática *bottom-up* podem ser vistas em Aho (1995).

2.1.2.2 ANÁLISE SINTÁTICA TOP-DOWN

Segundo Aho (1995), a análise sintática *top-down* pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada.

Um analisador sintático recursivo é aquele capaz de realizar a análise sintática *top-down*. Este analisador freqüentemente necessita de *retrocesso*, ou seja, analisar a mesma cadeia de entrada mais de uma vez para determinar a produção da gramática que deve ser usada para prosseguir com a análise sintática. Como o retrocesso é raramente necessitado para analisar sintaticamente construções de linguagem de programação, usa-se um caso especial de análise, chamada de *análise sintática recursiva preditiva*, onde o retrocesso não é exigido.

Para construir um analisador sintático preditivo, precisa-se conhecer, dado o símbolo corrente de entrada a e o não-terminal A a ser expandido, qual das alternativas da produção $A ::= \alpha_1/\alpha_2/.../\alpha_n$ é a única que deriva uma sentença começando por a . Ou seja, a alternativa adequada precisa ser detectável examinando-se apenas para o primeiro símbolo da cadeia que a mesma deriva. Na produção do quadro 2, as palavras-chave *if*, *while* e *begin* determinam qual regra de produção será utilizada, se a sentença de entrada for um comando (Aho, 1995).

QUADRO 2 - PRODUÇÕES PARA UM ANALISADOR SINTÁTICO PREDITIVO

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

Fonte: adaptado de Aho (1995)

Quando uma produção A é expandida, pode-se eventualmente encontrar a mesma produção A sem que nenhum caractere tenha sido consumido (recursão à esquerda). Esta situação leva o analisador sintático preditivo a um laço infinito. O quadro 3 mostra um exemplo de gramática onde isto acontece.

QUADRO 3 - UMA PRODUÇÃO RECURSIVA À ESQUERDA

```
expr ::= expr + termo
```

Fonte: adaptado de Aho (1995)

Para eliminar situações de conflito na análise sintática preditiva (como por exemplo o indeterminismo e a recursão à esquerda) existem procedimentos específicos, que serão mostrados a seguir.

2.1.2.3 FATORAÇÃO À ESQUERDA

A fatoração a esquerda é uma transformação gramatical útil para a criação de uma gramática adequada a análise sintática preditiva. A idéia básica está em, quando não estiver claro qual das duas produções alternativas usar para expandir um não-terminal A , reescrever a produção A de maneira a postergar a decisão até que tenha-se visto o suficiente da entrada para se realizar a escolha correta (Aho, 1995).

No quadro 4 é apresentado um exemplo onde existe uma ambigüidade que deve ser removida para que a análise sintática preditiva possa ser realizada. Quando o analisador sintático preditivo chega no terminal if , não pode imediatamente determinar qual produção escolher a fim de expandir cmd (Aho, 1995).

QUADRO 4 - EXEMPLO DE PRODUÇÃO QUE PODE SER FATORADA

$cmd ::= \text{if } expr \text{ then } cmd \text{ else } cmd$ $ \text{if } expr \text{ then } cmd$

Fonte: adaptado de Aho (1995)

Em geral, se $A ::= \alpha\beta_1 | \alpha\beta_2$, são duas produções- A , e a entrada começa por uma sentença α , não pode-se determinar se deve-se expandir A em $\alpha\beta_1$ ou em $\alpha\beta_2$. Neste caso pode-se postergar a decisão expandindo A para $\alpha A'$. Então, após atingir a entrada α , expande-se A' em β_1 ou em β_2 . A produção original é mostrada no quadro 5 e as produções após a fatoração são mostradas no quadro 6.

QUADRO 5 - PRODUÇÃO ANTES DA FATORAÇÃO

$A ::= \alpha\beta_1 \alpha\beta_2$

Fonte: adaptado de Aho (1995)

QUADRO 6 - PRODUÇÕES APÓS FATORAÇÃO

$A ::= \alpha A'$ $A' ::= \beta_1 \beta_2$
--

Fonte: adaptado de Aho (1995)

2.1.2.4 ELIMINAÇÃO DA RECURSIVIDADE À ESQUERDA

Uma gramática é recursiva à esquerda se possui um não-terminal A tal que exista uma produção $A ::= A\alpha$. Os métodos de análise sintática *top-down* não podem processar gramáticas recursivas à esquerda e, conseqüentemente, uma transformação que elimine a recursão é necessária (Aho, 1995).

No quadro 7 é apresentada uma produção onde existe uma recursão à esquerda. Para eliminar a recursão basta substituir a produção por aquelas apresentadas no quadro 8 (o símbolo ε representa uma sentença vazia). Isto é feito sem mudar o conjunto de cadeias de caracteres geradas por A . Esta regra é suficiente para muitas gramáticas (Aho, 1995).

QUADRO 7 - PRODUÇÃO RECURSIVA À ESQUERDA

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

Fonte: adaptado de Aho (1995)

QUADRO 8 - PRODUÇÕES PARA ELIMINAÇÃO DA RECURSÃO À ESQUERDA

$$\begin{aligned} A & ::= \beta A' \\ A' & ::= \alpha A' | \varepsilon \end{aligned}$$

Fonte: adaptado de Aho (1995)

Não importa quantas produções- A existam, pode-se eliminar a recursão das mesmas pela seguinte técnica. Primeiro, agrupa-se as produções- A , como mostrado no quadro 9, onde nenhum β começa por um A . Em seguida, substitui-se as produções- A por aquelas mostradas no quadro 10 (Aho, 1995).

QUADRO 9 - AGRUPAMENTO DAS PRODUÇÕES- A

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

Fonte: adaptado de Aho (1995)

QUADRO 10 - PRODUÇÕES NÃO RECURSIVAS

$$\begin{aligned} A & ::= \beta_1 A' | \beta_2 A' | \dots | \beta_n A' \\ A' & ::= \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \varepsilon \end{aligned}$$

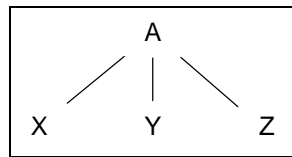
Fonte: adaptado de Aho (1995)

O não-terminal A agora não é recursivo e gera as mesmas sentenças que antes.

2.1.2.5 ÁRVORES GRAMATICAIS

Uma *árvore gramatical* mostra como o símbolo de partida de uma gramática deriva uma estrutura de linguagem. Se um não terminal A possui uma produção $A ::= XYZ$, então uma árvore gramatical pode ter um nó interior A , com três filhos rotulados X , Y e Z da esquerda para a direita, como mostrado na fig. 2 (Aho, 1995).

FIGURA 2 - UMA ÁRVORE GRAMATICAL



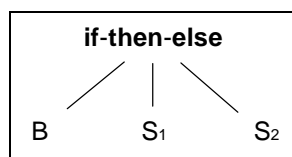
Fonte: Aho (1995)

Formalmente, dada uma gramática livre de contexto, uma árvore gramatical possui as seguintes propriedades (Aho, 1995):

- a raiz é rotulada pelo símbolo de partida;
- cada folha é rotulada por um átomo ou pela palavra vazia (ϵ);
- cada nó interior é rotulado por um não-terminal;
- se A é um não-terminal rotulando algum nó interior e X_1, X_2, \dots, X_n são os rótulos dos filhos daquele nó, da esquerda para a direita, então $A ::= X_1 X_2 \dots X_n$ é uma produção. Aqui X_1, X_2, \dots, X_n figuram no lugar de símbolos que sejam terminais ou não-terminais. Como um caso especial, se $A ::= \epsilon$, então um nó rotulado A deve possuir um único filho rotulado ϵ .

Uma *árvore sintática* é um caso especial de árvore gramatical onde os operadores e palavras-chave não figuram como folhas. Por exemplo, a produção $S ::= \text{if } B \text{ then } S_1 \text{ else } S_2$ é representada como mostrado na fig. 3 (Aho, 1995).

FIGURA 3 - UMA ÁRVORE SINTÁTICA



Fonte: Aho (1995)

2.1.3 ANÁLISE SEMÂNTICA

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

Segundo José (1987) não é uma tarefa trivial descrever completamente uma linguagem, ainda que simples, de tal modo que tanto sua sintaxe como sua semântica sejam contidas nesta descrição de maneira completa e precisa. As atividades de tradução, exercidas pelos compiladores, baseiam-se fundamentalmente em uma perfeita compreensão da semântica da 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.

As principais funções da análise semânticas são (José, 1987):

- a) criar e manter tabelas de símbolos;
- b) associar aos símbolos os correspondentes atributos (nome, tipo, etc.);
- 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 compatibilidade de tipos;
- h) efetuar o gerenciamento de memória;
- i) efetuar a tradução do programa;
- j) gerar de código.

Existem notações para associar regras semânticas às produções da gramática, tais como definições dirigidas pela sintaxe e esquemas de tradução. Maiores 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 associado de atributos, particionados em dois subconjuntos, chamados de atributos sintetizados e atributos herdados daquele símbolo gramatical. 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 gramatical é definido por uma regra semântica associada à produção usada naquele nó. O valor de um atributo sintetizado em um nó é computado a partir dos valores dos atributos dos filhos daquele nó na árvore gramatical. O valor de um atributo herdado é computado a partir dos valores dos atributos dos irmãos e pai daquele nó (Aho, 1995).

Uma árvore gramatical mostrando os valores dos atributos a cada nó é denominada de uma árvore gramatical anotada. O processo de computar os valores dos atributos a cada nó é chamado de *anotação* ou *decoração* da árvore (Aho, 1995).

2.1.3.1.1 GRAMÁTICAS DE ATRIBUTOS

Segundo Aho (1995), uma *gramática de atributos* é uma definição dirigida pela sintaxe, na qual as funções nas regras semânticas não têm efeitos colaterais, ou seja, esta função não altera seus parâmetros e nem variáveis não-locais.

Uma gramática de atributos pode ter atributos sintetizados e herdados.

2.1.3.1.1.1 ATRIBUTOS SINTETIZADOS

Os atributos sintetizados são usados extensivamente na prática. Uma definição dirigida pela sintaxe que use exclusivamente atributos sintetizados é dita uma *definição S-atribuída*. Uma árvore gramatical para uma definição S-atribuída pode ser sempre anotada através da avaliação das regras semânticas para os atributos a cada nó, de baixo para cima, das folhas para a raiz. Ou seja, o valor de um atributo sintetizado é determinado a partir dos valores dos atributos dos filhos daquele nó.

O exemplo do quadro 11 é uma definição S-atribuída que 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 (Aho, 1995).

QUADRO 11 - DEFINIÇÃO S-ATRIBUÍDA PARA UMA CALCULADORA DE MESA SIMPLES

Produção	Regras Semânticas
$L ::= E n$	$Imprimir (E.val)$
$E ::= E_1 + T$	$E.val := E_1.val + T.val$
$E ::= T$	$E.val := T.val$
$T ::= T_1 * F$	$T.val := T_1.val * F.val$
$T ::= F$	$T.val := F.val$
$F ::= (E)$	$F.val := E.val$
$F ::= \mathit{digito}$	$F.val := \mathit{digito.lexval}$

Fonte: adaptado de Aho (1995)

2.1.3.1.1.2 ATRIBUTOS HERDADOS

Segundo Aho (1995), um atributo herdado é aquele cujo valor é definido em termos do pai e/ou irmãos do nó que contém o atributo.

Os atributos herdados são convenientes para expressar a dependência de uma construção de linguagem de programação no contexto em que a mesma figura. Por exemplo, pode-se usar um atributo herdado para controlar se um identificador aparece do lado esquerdo ou direito de um comando de atribuição a fim de determinar se é necessário o endereço ou o valor de um identificador. Apesar de ser sempre possível reescrever uma definição dirigida pela sintaxe de forma a se usar somente atributos sintetizados, estas definições definidas pela sintaxe com atributos herdados são freqüentemente mais naturais (Aho, 1995).

O quadro 12 traz uma definição dirigida pela sintaxe de uma declaração de variáveis. Neste caso o atributo herdado distribui informações de tipo para os vários identificadores da declaração.

QUADRO 12 - DEFINIÇÃO DIRIGIDA PELA SINTAXE COM ATRIBUTOS HERDADOS

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

Fonte: adaptado de Aho (1995)

Uma declaração gerada pelo não-terminal D na definição dirigida pela sintaxe do quadro 12 consiste na palavra-chave *int* ou *real*, seguida por uma lista de identificadores. 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 a produção $D ::= T L$, 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 $L.in$.

2.1.3.1.1.3 ELIMINAÇÃO DA RECURSIVIDADE À ESQUERDA DE UM ESQUEMA DE TRADUÇÃO

Uma vez que a maioria dos operadores aritméticos é associativa à esquerda, é natural usar gramáticas recursivas à esquerda para expressões. Estende-se o algoritmo para eliminação da recursão à esquerda de forma a permitir atributos quando a gramática subjacente de um esquema de tradução for transformada. A transformação se aplica a esquemas de tradução com atributos sintetizados. Permite que muitas das definições dirigidas pela sintaxe sejam implementadas utilizando a análise sintática preditiva. O exemplo do quadro 13 mostra um esquema de tradução recursivo à esquerda (Aho, 1995).

Para a 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 $R_1.i$) é realizada após T ter sido completamente expandido em terminais e a segunda após R_1 ter sido completamente expandido (Aho, 1995).

QUADRO 13 - ESQUEMA DE TRADUÇÃO COM UMA GRAMÁTICA RECURSIVA À ESQUERDA

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

Fonte: adaptado de Aho (1995)

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

$E ::=$	T	$\{ R.i := T.val \}$
	R	$\{ E.val := R.s \}$
$R ::=$	$+$	
	T	$\{ R_1.i := R.i + T.val \}$
	R_1	$\{ R.s := R_1.s \}$
$R ::=$	$-$	
	T	$\{ R_1.i := R.i - T.val \}$
	R_1	$\{ R.s := R_1.s \}$
$R ::=$	ε	$\{ R.s := R.i \}$
$T ::=$	$($	
	E	
	$)$	$\{ T.val := E.val \}$
$T ::=$	\mathbf{Num}	$\{ T.val := \mathbf{num.val} \}$

Fonte: adaptado de Aho (1995)

Com a finalidade de adaptar à análise sintática preditiva a outros esquemas de tradução recursivos à esquerda, expressa-se o uso dos atributos $R.i$ e $R.s$ no quadro 14 mais abstratamente através do esquema de tradução do quadro 15 (Aho, 1995).

QUADRO 15 - ESQUEMA DE TRADUÇÃO ABSTRATO RECURSIVO À ESQUERDA

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

Fonte: adaptado de Aho (1995)

No quadro 15 cada símbolo gramatical possui um atributo sintetizado, escrito usando a letra minúscula correspondente, e f e g são funções arbitrárias. O algoritmo de eliminação de recursão à esquerda constrói a gramática do quadro 16 (Aho, 1995).

QUADRO 16 - GRAMÁTICA SEM RECURSIVIDADE À ESQUERDA

$A ::= X R$
$R ::= Y R \mid \varepsilon$

Fonte: adaptado de Aho (1995)

Considerando as ações semânticas, a gramática do quadro 16 torna-se a mostrada no quadro 17.

QUADRO 17 - ESQUEMA DE TRADUÇÃO SEM RECURSIVIDADE À ESQUERDA

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

Fonte: adaptado de Aho (1995)

O esquema transformado do quadro 17 usa os atributos i e s para R , como no quadro 14. Os resultados obtidos pelos esquemas do quadro 15 e quadro 17 são os mesmos (Aho, 1995).

2.2 O CONCEITO DE AMARRAÇÃO

Programas envolvem *entidades*, como variáveis, procedimentos, comandos e assim por diante. Estas entidades têm certas propriedades, chamadas de *atributos*. Por exemplo, uma variável tem um nome, um tipo, uma área de memória onde seu valor é armazenado; um procedimento tem um nome, parâmetros formais etc. Os atributos de uma entidade precisam ser especificados antes que a mesma seja processada. A especificação da natureza exata dos atributos de uma entidade se dá o nome de *amarração* (Ghezzi, 1991).

Ghezzi (1991) diz que uma amarração é *estática* se é estabelecida antes da execução do programa e não pode ser alterada depois; e é *dinâmica* se é estabelecida em tempo de execução e pode ser alterada, de acordo com regras especificadas pela linguagem.

2.2.1 VARIÁVEL

Computadores convencionais baseiam-se no conceito de uma memória principal que consiste em células elementares, cada qual identificada por um endereço. O conteúdo de uma célula contém seu *valor*. Este valor pode ser lido e/ou modificado. Esta modificação é feita pela substituição de um valor por outro. O conceito de variável é introduzido como uma abstração de células de memória e o conceito de comando de atribuição como uma abstração destrutiva destas células (Ghezzi, 1991).

Uma variável é caracterizada por um nome e quatro atributos básicos: *escopo*, *tempo de vida*, *valor* e *tipo* (Ghezzi, 1991).

2.2.1.1 ESCOPO DE UMA VARIÁVEL

O escopo de uma variável é a trecho de programa onde a variável é conhecida e pode assim ser usada. A variável é *visível* dentro do escopo e *invisível* fora dele. A amarração de variáveis a escopos pode ser feita *estática* ou *dinamicamente* (Ghezzi, 1991).

A *amarração estática a escopo* define o escopo de uma variável a partir da estrutura léxica de um programa, isto é, cada referência a uma variável é estaticamente amarrada a uma declaração (implícita ou explícita) desta variável (Ghezzi, 1991). A maioria das linguagens contemporâneas implementa a amarração estática a escopo.

Na *amarração dinâmica a escopo*, o escopo de uma variável é definido em função da execução do programa. Tipicamente, o efeito da declaração de uma variável se estende sobre todas as instruções executadas a seguir, até que uma nova declaração para uma variável com o mesmo nome seja encontrada. APL, LISP e SNOBOL4 são exemplos de linguagens com amarração dinâmica a escopo (Ghezzi, 1991).

2.2.1.2 O TEMPO DE VIDA DE UMA VARIÁVEL

O tempo de vida de uma variável é o intervalo de tempo durante o qual uma área de memória está amarrada a uma variável. Esta área guarda o valor da variável. Usa-se o termo *objeto de informação* (ou simplesmente *objeto*) para designar o par área de memória e valor armazenado (Ghezzi, 1991).

A ação que adquire áreas de memória para variáveis é chamada *alocação*. Em algumas linguagens a alocação é realizada antes da execução do programa (*alocação estática*). Em outras linguagens ela é feita em tempo de execução (*alocação dinâmica*) devido a um pedido explícito do programa ou automaticamente quando da ativação da unidade onde está declarada a variável (Ghezzi, 1991).

2.2.1.3 O VALOR DE UMA VARIÁVEL

O valor de uma variável é representado, de forma codificada, na área de memória amarrada à variável. Este código é interpretado de acordo com o tipo da variável. A amarração entre uma variável e o valor armazenado na área de memória correspondente é, em geral, dinâmica, já que este valor pode ser modificado por operações de atribuição (Ghezzi, 1991).

Algumas linguagens, entretanto, permitem o congelamento da amarração entre uma variável e seu valor, quando a amarração é estabelecida. A entidade resultante é, sob qualquer aspecto, uma *constante simbólica* definida pelo programador (Ghezzi, 1991). Um exemplo desta situação está no quadro 18.

QUADRO 18 - DEFINIÇÃO DE UMA CONSTANTE EM PASCAL

```
const pi = 3.1416;
```

Fonte: Ghezzi (1991)

No quadro 18, a variável *pi* está amarrada ao valor 3,1416 e este valor não pode ser modificado, isto é, o tradutor acusa um erro se existe atribuição a *pi* (Ghezzi, 1991).

2.2.1.4 O TIPO DE UMA VARIÁVEL

Segundo Ghezzi (1991), o tipo de uma variável pode ser considerado como uma especificação da classe de valores que podem ser associados à variável, bem como das operações que podem ser usadas para criar, acessar e modificar estes valores. O quadro 19 apresenta um exemplo.

QUADRO 19 - DECLARAÇÃO DE TIPO EM PASCAL

```
type t = array[1..10] of boolean;
```

Fonte: Ghezzi (1991)

A declaração do quadro 19 estabelece, em tempo de tradução, uma amarração entre o nome de tipo t e sua implementação (isto é, um *array* de 10 valores booleanos, cada valor acessado por um índice no intervalo 1 a 10). Como consequência desta amarração, o tipo t herda todas as operações da estrutura de dados usada na sua representação (o *array*). Assim, torna-se possível ler e modificar cada componente de um objeto do tipo t , por meio da indexação no *array* (Ghezzi, 1991).

2.2.2 UNIDADES DE PROGRAMAS

Linguagens de programação permitem que programas sejam compostos de *unidades*. Estas unidades podem ser desenvolvidas até certo ponto independentemente e podem algumas vezes ser traduzidas em separado e combinadas depois da tradução. As variáveis declaradas dentro de uma unidade são *locais* à unidade (Ghezzi, 1991).

Uma unidade não é uma parte autocontida, completamente independente do resto do programa. Se a unidade é um subprograma, então ela pode ser ativada por uma chamada de subprograma existente em outra unidade, que recupera o controle depois da execução do subprograma chamado (Ghezzi, 1991).

2.2.2.1 UNIDADES DE ATIVAÇÃO

A representação de uma unidade de programa durante sua execução é chamada *unidade de ativação*. A ativação de uma unidade consiste em um *segmento de código* e um *registro de ativação (RA)*. O segmento de código, cujo conteúdo é fixo durante a execução, contém as instruções da unidade. O conteúdo do registro de ativação pode variar. Este registro contém toda a informação necessária a execução da unidade, incluindo, entre outras coisas, os objetos associados as variáveis locais de uma ativação da unidade (Ghezzi, 1991).

2.2.2.1.1 REGISTROS DE ATIVAÇÃO EM LINGUAGENS ESTÁTICAS

Uma linguagem estática é composta de uma série de unidades: um programa ou procedimento principal e um conjunto (que pode ser vazio) de subprogramas (procedimentos e funções). O tamanho da área de memória necessária a cada variável local é fixo: é conhecido em tempo de tradução e não pode ser modificado durante a execução da unidade (Ghezzi, 1991).

Cada unidade é compilada e associada a um registro de ativação que é alocado antes da execução, ou seja, variáveis são criadas antes da execução e seu tempo de vida se estende por toda a duração do programa (*variáveis estáticas*) (Ghezzi, 1991).

2.2.2.1.2 REGISTROS DE ATIVAÇÃO EM LINGUAGENS DINÂMICAS

Linguagens dinâmicas possuem um mecanismo – a *estrutura de bloco* – para o controle do escopo de variáveis e para a divisão do programa em unidades. Duas unidades quaisquer de um programa podem ser *disjuntas* (sem partes comuns) ou *aninhadas* (uma unidade engloba a outra). A estrutura de um programa pode ser considerada como um *aninhamento estático de unidades* (Ghezzi, 1991).

Se uma variável é declarada como local a uma unidade U , ela é visível em U , mas não nas unidades que englobam U . Entretanto (com uma exceção), ela é visível a todas as unidades que estaticamente estão envolvidas por U . A exceção mencionada ocorre quando uma variável local a uma unidade tem o mesmo nome de uma outra variável declarada em uma unidade envolvente. Neste caso, o mesmo nome poderá denotar ou o objeto declarado localmente ou o objeto global declarado na unidade mais externa (Ghezzi, 1991).

Para que se possa modelar o comportamento em tempo de execução de linguagens dinâmicas, é necessário especificar as regras que governam o tempo de vida de variáveis e a criação de ambientes de referência para ativações de unidades. Em geral, todas as variáveis locais a uma unidade são automaticamente criadas quando a unidade é ativada, isto é, não existe operação explícita de criação de variáveis. Além disso, em algumas linguagens, a quantidade de memória necessária a cada variável tem que ser invariante, e conhecida estaticamente. Em outras linguagens, a quantidade de memória necessária a cada variável é conhecida somente durante a execução, quando a unidade é ativada. Em outros casos, ainda variáveis locais são explicitamente criadas pelo programador. Neste caso a memória necessária a um registro de ativação não é conhecida nem mesmo quando a unidade é ativada. O uso da memória pode aumentar dinamicamente quando novos comandos de criação são executados (Ghezzi, 1991). Quando o tamanho das variáveis é conhecido, pode-se também determinar o tamanho do registro de ativação das unidades.

2.3 ALOCAÇÃO DE MEMÓRIA

A alocação de memória nas diversas linguagens está relacionada à maneira como estas linguagens tratam a memória e ao momento das amarrações dos nomes (variáveis) às localizações de memória. Linguagens estáticas determinam a utilização da memória em tempo de compilação. Já as linguagens dinâmicas o fazem durante a execução do programa. A seguir são apresentadas as técnicas utilizadas por estes tipos de linguagens.

2.3.1 ALOCAÇÃO DE MEMÓRIA EM LINGUAGENS ESTÁTICAS

Em linguagens estáticas, as amarrações de nomes às localizações de memória são feitas em tempo de compilação. Desta forma um pacote de suporte em tempo de execução não é necessário. Como as amarrações de nomes não se alteram, em diversas chamadas de um mesmo procedimento, os nomes são amarrados às mesmas localizações de memória. Isto permite que os valores sejam *retidos*, ou seja, quando o controle deixar um procedimento, os valores de seus objetos de dados locais permanecerão intactos (Aho, 1995).

O compilador determina, a partir do tipo de um nome, o tamanho de memória necessário para armazenar os dados do mesmo. Desta maneira os endereços são determinados a partir de uma das extremidades do registro de ativação. É necessário então determinar onde ficarão os registros de ativação na memória. Uma vez fixados estes valores e endereços, eles não se alteram durante a execução do programa. Durante a compilação o código-alvo gerado pode ter os endereços preenchidos de maneira a encontrar os objetos de dados necessários à execução do programa (Aho, 1995).

Segundo Aho (1995), algumas limitações são encontradas na alocação estática de memória:

- a) o tamanho dos objetos de dados e as restrições sobre suas posições de memória precisam ser conhecidos em tempo de compilação;
- b) os procedimentos recursivos não podem ser usados uma vez que todas as ativações de um procedimento usam as mesmas localizações de memória para os objetos locais;
- c) estruturas dinâmicas de dados não podem ser usadas porque não existem mecanismos de alocação de memória em tempo de execução.

2.3.1.1 ORGANIZAÇÃO DE MEMÓRIA

As linguagens com alocação estática de memória possuem organizações bem simples. Normalmente ela consiste em um programa (ou procedimento) principal, sub-rotinas e funções. Um exemplo de uma linguagem estática é o Fortran 77. O quadro 20 apresenta um exemplo de programa Fortran (Aho, 1995).

QUADRO 20 - UM PROGRAMA FORTRAN 77

```

PROGRAM CNSUME
  CHARACTER * 50 BUF
  INTEGER NEXT
  CHARACTER C, PRDUCE
  DATA NEXT /1/, BUF, /' '/
6   C = PRDUCE( )
    BUF (NEXT:NEXT) = C
    NEXT = NEXT + 1
    IF ( C. NE. ' ' ) GOTO 6
    WRITE (*, '(A)') BUF
    END

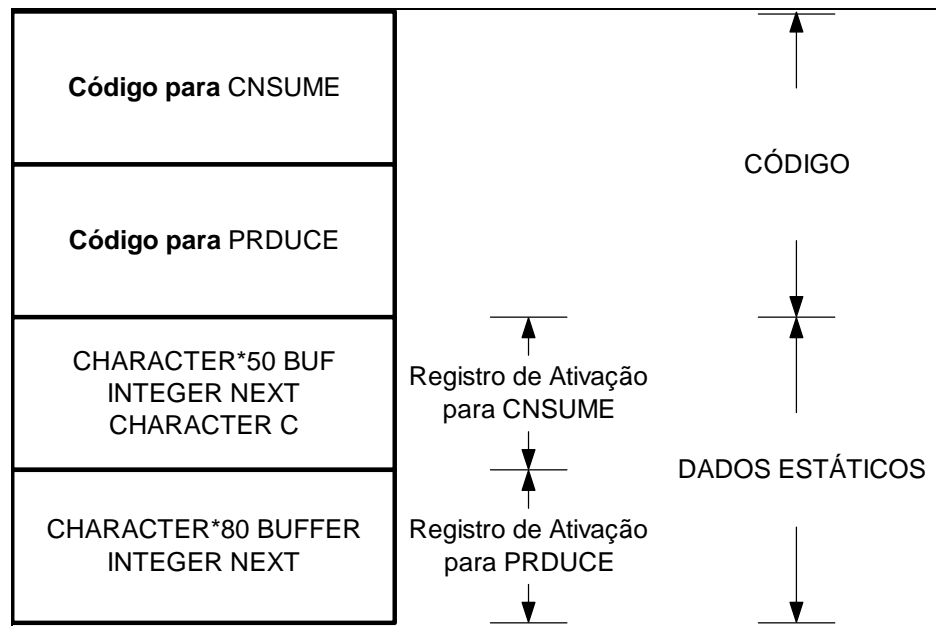
  CHARACTER FUNCTION PRDUCE( )
    CHARACTER * 80 BUFFER
    INTEGER NEXT
    SAVE BUFFER, NEXT
    DATA NEXT /81/
    IF ( NEXT .GT. 80 ) THEN
      READ (*, '(A)') BUFFER
      NEXT = 1
    END IF
    PRDUCE = BUFFER(NEXT:NEXT)
    NEXT = NEXT+1
    END

```

Fonte: Aho (1995)

Considerando a organização explicada na seção anterior, a disposição do código e dos registros de ativações para o programa do quadro 20 é mostrada na fig. 4.

FIGURA 4 - ARMAZENAMENTO ESTÁTICO PARA OS OBJETOS ASSOCIADOS A IDENTIFICADORES LOCAIS NUM PROGRAMA FORTRAN 77



Fonte: Aho (1995)

Dentro do registro de ativação para *CNSUME* existe espaço para seus objetos de dados locais. A memória amarrada a *BUF* abriga uma cadeia de 50 caracteres. É seguida por espaço para guardar um valor inteiro para *NEXT* e um valor caractere para *C*. O fato de *NEXT* ser também declarado em *PRDUCE* não apresenta problema porque os objetos de dados locais para os dois procedimentos obtêm espaço nos seus respectivos registros de ativação.

Como os tamanhos do código executável e do registro de ativação são conhecidos em tempo de compilação, são possíveis organizações de memória diferentes da apresenta na fig. 4, como por exemplo, colocar o registro de ativação para um procedimento juntamente com o código desse procedimento. Em alguns sistemas de computação é viável deixar a posição relativa dos registros de ativação inespecificada e permitir que o editor de ligações ligue os registros de ativação e o código executável (Aho, 1995).

2.3.2 ALOCAÇÃO DE MEMÓRIA EM LINGUAGENS DINÂMICAS

Linguagens dinâmicas permitem que a memória seja utilizada no momento em que ela é necessária. Estas linguagens implementam diferentes técnicas para a alocação memória. A seguir serão apresentadas estas técnicas.

2.3.2.1 ALOCAÇÃO IMPLÍCITA DE MEMÓRIA

A alocação implícita em linguagens dinâmicas acontece quando um registro de ativação é alocado. As variáveis locais são alocadas dentro do registro de ativação quando a unidade correspondente for ativada. Isto acontece sem que o programador tenha que utilizar recursos da linguagem para desencadear a alocação.

Em C acontece uma alocação implícita de memória quando uma cadeia de caracteres constante é atribuída para uma variável do tipo *array* de *char*. O armazenamento para este valor é determinado pelo compilador sem que o usuário precise informar. Um comando que gera esta alocação pode ser vista no quadro 21 (Sebesta, 2000).

QUADRO 21 - UMA ATRIBUIÇÃO DE UMA CADEIA DE CARACTERES EM C, ONDE A MEMÓRIA É ALOCADA IMPLÍCITAMENTE

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

Fonte: Sebesta (2000)

Outro exemplo de alocação implícita é o tipo *string* em Object Pascal. Variáveis deste tipo são alocadas implicitamente pela rotina de suporte em tempo de execução. Uma atribuição de uma literal *string* para uma variável do tipo *string* em Object Pascal é mostrada no quadro 22. Quando esta atribuição é executada, a linguagem aloca implicitamente um bloco de memória para abrigar o novo conteúdo. No registro de ativação do procedimento que abriga esta variável existe apenas uma referência para o bloco de memória alocado dinamicamente. A liberação desta variável é implícita, realizada ao fim do tempo de vida da mesma (Borland, 2001).

QUADRO 22 - EXEMPLO DE ATRIBUIÇÃO DE *STRING* EM OBJECT PASCAL

```
var
  Nome: String;
begin
  Nome := 'José da Silva';
end;
```

2.3.2.2 ALOCAÇÃO EXPLÍCITA DE MEMÓRIA

A alocação explícita de memória acontece quando o programa solicita explicitamente células de memória. Esta solicitação pode ser realizada através de recursos disponibilizados pela linguagem. Normalmente a área para tais dados é proveniente de uma área especialmente destinada à alocação dinâmica de memória (geralmente chamada de *heap*).

Pascal é um exemplo de linguagem onde a alocação explícita de memória é usada. Por exemplo, a alocação explícita é realizada usando o procedimento padrão *new*. A execução de *new(p)* reserva memória para o tipo de objeto apontado por *p* e o próprio *p* é deixado apontando para o objeto recém-alocado. A liberação é feita chamando-se o procedimento *dispose* na maioria das implementações de Pascal (Aho, 1995).

O programa Pascal do quadro 23 constrói a lista dinamicamente alocada da fig. 5 e imprime os inteiros armazenados nas células. Quando a execução do programa começa na linha 15, o armazenamento do apontador *cabeça* está no registro de ativação para todo o programa. A cada vez que o controle atinge a linha 11, a chamada *new(p)* resulta numa célula sendo alocada em algum lugar do *heap*. O comando *p^.chave* (linha 11) altera o campo chave do registro referenciado por *p* (Aho, 1995).

QUADRO 23 - ALOCAÇÃO DINÂMICA DE CÉLULAS USANDO *NEW* EM PASCAL

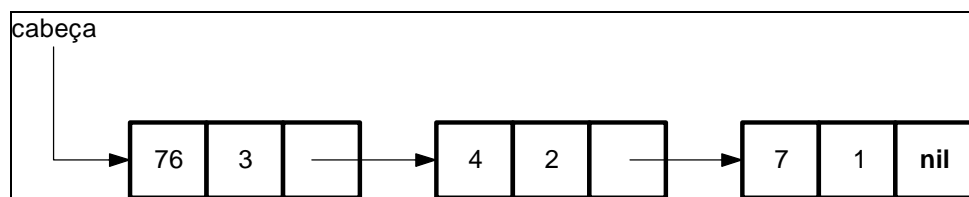
```

(1) program tabela (input, output);
(2) type link = ^celula;
(3)     celula = record
(4)         chave, info: integer;
(5)         proximo: link;
(6)     end;
(7) var cabeca : link;
(8) procedure inserir (k, i: integer);
(9)     var p: link;
(10)  begin
(11)     new (p); p^.chave := k; p^.info := i;
(12)     p^.proximo := cabeca; cabeca := p
(13)  end;
(14) begin
(15)  cabeca := nil;
(16)  inserir(7,1); inserir(4,2); inserir(76,3);
(17)  writeln(cabeca^.chave, cabeca^.info);
(18)  writeln(cabeca^.proximo^.chave, cabeca^
    .proximo^.info);
(19)  writeln(cabeca^.proximo^.proximo^.chave,
    cabeca^.proximo^.proximo^.proximo^.info);
(20) end.

```

Fonte: Aho (1995)

FIGURA 5 - LISTAS LIGADAS CONSTRUÍDAS PELO PROGRAMA DO QUADRO 5



Fonte: Aho (1995)

No exemplo do quadro 23, quando o controle retorna para o programa principal, proveniente de *inserir*, as células alocadas estão acessíveis. Em outras palavras, as células alocadas através do procedimento *new* durante a ativação de *inserir* são retidas quando o controle retorna para o programa principal (Aho, 1995).

2.3.2.3 ORGANIZAÇÃO DA MEMÓRIA

Aho (1995) menciona que existem dois tipos de memórias para as linguagens dinâmicas. Elas são:

- a) *memória de pilha*: consiste no gerenciamento de memória em tempo de execução em forma de pilha;
- b) *memória de heap*: reserva e libera áreas de memória na medida do necessário em tempo de execução, a partir de uma área de dados conhecida como *heap*.

2.3.2.3.1 MEMÓRIA DE PILHA

A memória de pilha esta organizada numa estrutura de pilha, como o próprio nome diz. Os registros de ativação são empilhados e desempilhados à medida que começam e terminam, respectivamente. Neste conceito os objetos locais são amarrados a uma nova área de memória toda vez que uma ativação de procedimento acontece. Também no término da execução do procedimento, os objetos locais são removidos porque a memória utilizada por eles é liberada (Aho, 1995).

Quando os tamanhos de todos os registros de ativação são conhecidos em tempo de compilação, a alocação de memória de pilha torna-se bastante simples. Suponha-se que *topo* seja o apontador do topo da pilha. Se o registro de ativação de um procedimento *p* tiver o tamanho *a*, então antes de *p* ser chamado, *topo* é incrementado de *a* e o controle passa para o procedimento *p*. Quando o controle retorna de *p*, *topo* é decrementado de *a* (Aho, 1995).

O quadro 24 apresenta um pseudocódigo onde existe um procedimento *p* com 3 *arrays* abertos. Os limites destes *arrays* somente serão conhecidos na ativação da unidade, fazendo com que o registro de ativação de *p* seja variável. Para o tratamento de registros de ativação com dados de tamanhos variáveis pode-se utilizar a estratégia sugerida na fig. 6, onde o procedimento *p* tem 3 *arrays*. O espaço de memória destes *arrays* não faz parte do registro de ativação para *p*. Somente um apontador para o início de cada *array* existe no registro de

ativação. Os endereços relativos destes apontadores são conhecidos e, portanto, os *arrays* são acessíveis através destes apontadores. Também um procedimento *q* é chamado por *p*. O registro de ativação para este procedimento começa além dos *arrays* de *p*. Os *arrays* de *q* começam depois deste ponto (Aho, 1995).

QUADRO 24 - PSEUDOCÓDIGO DE UMA UNIDADE COM ARRAYS ABERTOS

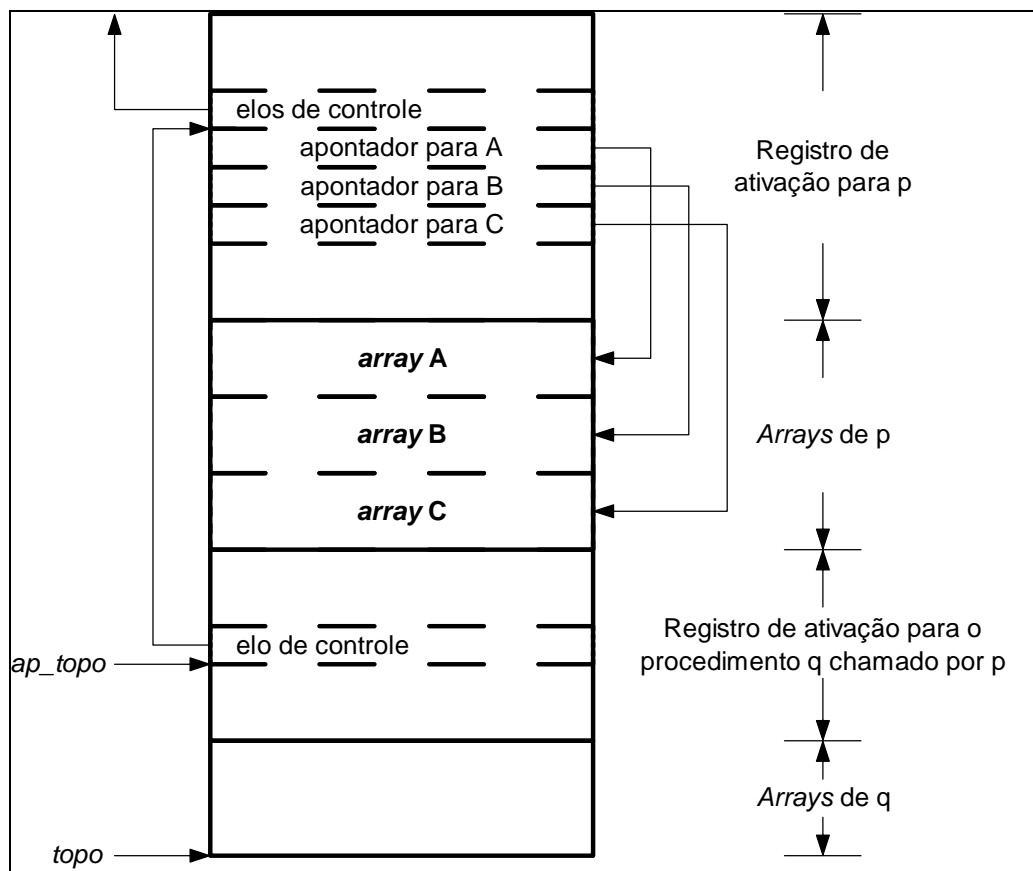
```

procedimento P(LimiteA, LimiteB, LimiteC: inteiro);
var
  A: array[1..LimiteA]: inteiro;
  B: array[1..LimiteB]: inteiro;
  C: array[1..LimiteC]: inteiro;
início
  Q; // Chamada do procedimento Q
fim;

procedimento Q;
  // Definição dos arrays de Q
início
  // Código para Q
fim;

```

FIGURA 6 - ACESSO A ARRAYS ABERTOS ALOCADOS DINAMICAMENTE



Fonte: Aho (1995)

Como mostrado na fig. 6, acesso aos dados da pilha se dá através de dois apontadores: *topo* e *ap_topo*. O primeiro marca o topo atual da pilha, onde o próximo registro de ativação irá ser alocado. O segundo aponta para o fim do campo de estado da máquina no registro de ativação de *q* (os dados variáveis estão além deste campo). Dentro deste campo existe um apontador para o valor prévio de *ap_topo*, quando o controle ainda estava na ativação de *p*.

Os apontadores *ap_topo* e *topo* podem ter seus valores manipulados por códigos gerados em tempo de compilação usando os tamanhos conhecidos do registro de ativação. Quando *q* retorna, o valor do *topo* pode ser determinado através do *ap_topo* menos o tamanho dos campos anteriores ao campo de estado da máquina no registro de ativação. Após isto o novo valor de *ap_topo* pode ser determinado usando o valor do elo de controle de registro de ativação de *q*.

2.3.2.3.2 MEMÓRIA HEAP

A alocação de memória *heap* fornece blocos de memória contíguos que podem ser usados para registros de ativação ou outros objetos. Estes podem ser liberados em qualquer ordem de maneira que, ao longo do tempo, a memória *heap* irá se constituir de blocos livres e em uso (Aho, 1995).

A diferença entre a alocação de memória *heap* e de pilha para os registros de ativação está no fato de que os registros de ativação podem ser organizados em uma ordem que não segue a ordem de alocação dos mesmos. Sejam A, B, C, D registros de ativação para procedimentos. Estes registros de ativação podem ser alocados na ordem A, D, C, B, mas a sua ordem na memória *heap* pode ser D, B, C, A. Isto pode acontecer porque a liberação de um dos registros de ativação da memória faz com que exista um espaço livre, que mais tarde pode ser usado quando for necessária a alocação de mais um registro de ativação (Aho, 1995).

2.4 CLASSIFICAÇÃO DE VARIÁVEIS CONFORME PROCESSO DE ALOCAÇÃO

As variáveis podem ser classificadas conforme o seu tipo de alocação. As categorias são as seguintes (Ghezzi, 1991):

- a) *variáveis estáticas*: são aquelas criadas antes da execução do programa e seu tempo de vida se estende por toda a duração do mesmo;

- b) *variáveis semi-estáticas*: são aquelas que têm tamanho conhecido em tempo de compilação, mas são amarradas as localizações de memória somente em tempo de execução;
- c) *variáveis semidinâmicas*: são aquelas automaticamente criadas quando a unidade onde ela está declarada é ativada, mas seu tamanho pode depender de valores que são conhecidos somente na ativação da unidade. Este é o caso dos *arrays abertos*, agregados cujos limites tornam-se conhecidos na ativação da unidade;
- d) *variáveis dinâmicas*: são aquelas que não têm tamanho conhecido na ativação da unidade onde ela está declarada. Um exemplo deste tipo de variável é o *array flexível*. Os seus limites podem variar durante a execução do programa, alterando assim o tamanho da memória necessária para o valor da variável. Nesta categoria ainda estão as variáveis requisitadas e controladas pelo programa. Estas variáveis são explicitamente alocadas e desalocadas a qualquer momento pelo programa através de operadores e/ou funções da linguagem. As áreas alocadas para estas variáveis são acessíveis através de referências localizadas no registro de ativação da unidade onde elas estão declaradas.

2.5 MAPEAMENTO FINITO (ARRAYS)

Segundo Sebesta (2000), um *array* é um agregado homogêneo de dados cujo elemento individual é identificado por sua posição no agregado em relação ao primeiro. Uma referência a um elemento freqüentemente é feita por um ou mais valores não constantes. Isto acarreta em cálculos para se determinar a posição de memória onde o elemento está armazenado. A necessidade de se modelar dados de um mesmo tipo de maneira conjunta torna evidente a necessidade do uso de *arrays*.

2.5.1 ARRAYS E ÍNDICES

Os *arrays* são compostos por um mecanismo sintático de dois níveis, onde o primeiro é o nome do *array* e o segundo é a lista de índices, também conhecidos como subscritos. Esta lista de índices forma o seletor de elementos que especifica unicamente um elemento no *array*. Este seletor pode ser estático quando os seus valores forem constantes. Caso contrário será dinâmico (Sebesta, 2000).

A sintaxe utilizada para *arrays* é praticamente universal. Utiliza-se o nome do *array* seguido da lista de subscriptos envolvidos em parênteses ou colchetes. A linguagem Ada utiliza os parênteses para envolver os subscriptos, já o Pascal utiliza o colchete (Sebesta, 2000).

Nesta construção, dois tipos são envolvidos: o tipo dos elementos e dos subscriptos. Este último normalmente é uma subfaixa de inteiros, mas linguagens como Pascal e Ada permitem o uso de tipos booleanos, caracteres e enumerações definidos pelo usuário (Sebesta, 2000).

2.5.2 CATEGORIAS DE ARRAYS

Segundo Sebesta (2000), existem quatro categorias de *arrays*:

- a) um *array estático* é aquele que tem a sua faixa de subscriptos e localizações de memória amarradas estaticamente. Nenhuma alocação precisa ser realizada durante a execução do programa;
- b) um *array stack-dinâmico fixo* (ou *semiestático*) tem o seu subscrito amarrado estaticamente mas a sua alocação é feita na elaboração da sua declaração durante a execução do programa;
- c) para um *array stack-dinâmico* (ou *semidinâmico*), os subscriptos são amarrados na execução do programa, assim como a memória do *array*. Mas uma vez amarrados, permanecem inalterados durante o tempo de vida da variável. Tem muita flexibilidade em relação ao *stack-dinâmico fixo*. Seu tamanho somente é conhecido quando o *array* estiver prestes a ser usado;
- d) um *array heap-dinâmico* (ou apenas *dinâmico*) é aquele em que, tanto a sua faixa de subscriptos quanto a sua memória são amarrados dinamicamente durante a execução. Podem mudar a qualquer momento durante o seu tempo de vida. Em relação às demais categorias é muito flexível. O seu tamanho pode mudar durante a execução para refletir as necessidades de espaço do programa.

2.5.3 QUANTIDADE DE SUBSCRITOS EM UM ARRAY

O projeto da linguagem pode limitar o número de subscriptos em um *array*, assim como aconteceu com o FORTRAN I. Esta linguagem limitou em três o número de subscriptos que podem ser usados em *arrays*, porque na época do seu desenvolvimento, a eficiência era uma

preocupação fundamental. Os projetistas do FORTRAN I desenvolveram um método muito rápido para acessar elementos de *arrays* de até três dimensões, mas não mais que três (Sebesta, 2000).

A linguagem C permite a utilização de *arrays* com somente um subscrito, mas estes podem ter outros *arrays* como elementos, suportando assim, os multidimensionais. Na maioria das outras linguagens contemporâneas não existe este limite (Sebesta, 2000).

2.5.4 INICIALIZAÇÃO DE ARRAYS

Algumas linguagens de programação permitem que os *arrays* sejam inicializados no momento da amarração do *array* a sua localização de memória. Em FORTRAN 77, toda memória é alocada estaticamente. Através do operador DATA, o *array* pode receber valores iniciais em tempo de carga do programa (*load-time*), como mostrado no quadro 25 (Sebesta, 2000). Os valores entre as duas barras serão atribuídos ao *array* lista durante a carga do programa.

QUADRO 25 - DECLARAÇÃO E INICIALIZAÇÃO DE ARRAYS EM FORTRAN 77

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

Fonte: Sebesta (2000)

O ANSI C também permite a inicialização de *arrays*, mas este deve ser feito na declaração do mesmo, como mostrado no quadro 26. O mesmo acontece com as *strings* de caracteres, que são implementadas como *arrays* de *char* (Sebesta, 2000).

QUADRO 26 - INICIALIZAÇÃO DE ARRAYS EM ANSI C

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

Fonte: Sebesta (2000)

2.5.5 IMPLEMENTAÇÃO DE ARRAYS

Os elementos de um *array* podem receber acesso rapidamente se forem armazenados num bloco de localizações consecutivas. Se o tamanho de cada elemento é w e o índice do elemento é i , a localização de um elemento em um *array* A começa na localização calculada pela expressão do quadro 27 (Aho, 1995).

QUADRO 27 - CÁLCULO DO *I*ÉSIMO ELEMENTO DE UM ARRAY

$$\text{base} + (i - \text{linf}) \times w$$

Fonte: Aho (1995)

No quadro 27, *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 $A[\text{linf}]$ (Aho, 1995).

A expressão do quadro 27 pode ser avaliada em tempo de compilação se for escrita como mostrado no quadro 28.

QUADRO 28 - EXPRESSÃO DE CÁLCULO DE LOCALIZAÇÃO DE ELEMENTOS EM TEMPO DE COMPILAÇÃO

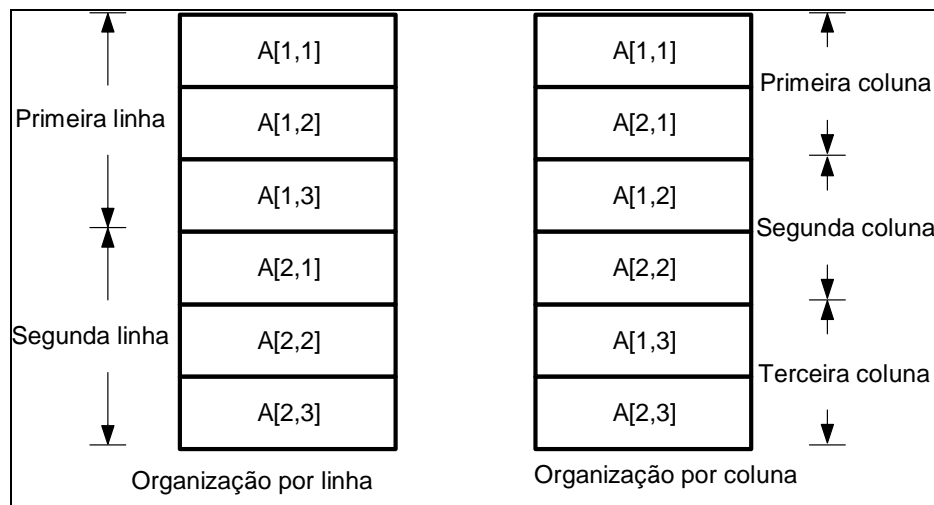
$$i \times w + (\text{base} - \text{linf} \times w)$$

Fonte: Aho (1995)

A subexpressão $c = \text{base} - \text{linf} \times w$ pode ser avaliada quando a declaração do *array* for encontrada. 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* (Aho, 1995).

Para *arrays* multidimensionais, o pré-cálculo em tempo de compilação também pode ser aplicado. Um *array bidimensional* é normalmente armazenado de duas formas, ou por *linha* (linha a linha) ou por *coluna* (coluna a coluna). A fig. 7 mostra a disposição de um *array* A 2×3 por linha e por coluna (Aho, 1995).

FIGURA 7 - DISPOSIÇÕES PARA UM ARRAY BIDIMENSIONAL



Fonte: adaptado de Aho (1995)

No quadro 29 é apresentada a expressão para cálculo do endereço relativo de um elemento $A[i_1, i_2]$.

QUADRO 29 - EXPRESSÃO PARA CÁLCULO DE ENDEREÇOS RELATIVOS DE ELEMENTOS DE UM ARRAY BIDIMENSIONAL

$$\text{base} + ((i_1 - \text{linf}_1) \times n_2 + i_2 - \text{linf}_2) \times w$$

Fonte: Aho (1995)

No quadro 29, linf_1 e linf_2 representam os limites inferiores sob os valores i_1 e i_2 . 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 $n_2 = \text{lsup}_2 - \text{linf}_2 + 1$. Assumindo que i_1 e i_2 são os únicos valores que não são conhecidos em tempo de compilação, pode-se reescrever a expressão acima como mostrado no quadro 30, onde o último termo pode ser guardado em tempo de compilação (Aho, 1995).

QUADRO 30 - EXPRESSÃO PARA CÁLCULO DE ENDEREÇOS RELATIVOS DE ELEMENTOS DE UM ARRAY BIDIMENSIONAL EM TEMPO DE COMPILAÇÃO

$$((i_1 \times n_2) + i_2) \times w + (\text{base} - ((\text{linf}_1 \times n_2) + \text{linf}_2) \times w)$$

Fonte: Aho (1995)

Pode-se generalizar a organização por linha e por coluna para várias dimensões. A generalização da organização por linha significa armazenar os elementos de tal forma que, à medida que se esquadrinha um bloco de memória, os subscritos mais à direita parecem variar mais rapidamente. A expressão do quadro 31 é a expressão genérica para o endereço relativo $A[i_1, i_2, \dots, i_n]$ (Aho, 1995).

QUADRO 31 - EXPRESSÃO GENÉRICA PARA CÁLCULO DE ENDEREÇOS RELATIVOS DE ARRAYS MULTIDIMENSIONAIS

$$((\dots ((i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w + \text{base} - ((\dots ((\text{linf}_1 n_2 + \text{linf}_2) n_3 + \text{linf}_3) \dots) n_k + \text{linf}_k) \times w$$

Fonte: Aho (1995)

Algumas linguagens permitem que os comprimentos de *arrays* sejam determinados durante a execução do programa. A alocação de tais *arrays* já foi discutida na seção 2.3.2.3.1. As fórmulas de acesso aos elementos destes *arrays* são as mesmas que aquelas para *arrays* de tamanhos fixos, mas os limites superior e inferior não são conhecidos em tempo de compilação (Aho, 1995).

2.6 ACESSO A VARIÁVEIS NÃO-LOCAIS

Segundo Aho (1995), as regras de escopo de uma linguagem determinam o tratamento das referências aos nomes não-locais. Uma regra comum, chamada de *regra de escopo léxico ou estático*, determina a declaração que se aplica a um nome pelo exame isolado do texto do programa. Pascal, C e Ada estão entre as muitas linguagens que usam o escopo estático. Uma regra alternativa, chamada de *regra de escopo dinâmico*, determina a declaração aplicável a um nome em tempo de execução pela consideração das ativações correntes. Lisp, APL e Snobol estão dentre as linguagens que usam o escopo dinâmico.

Maiores informações sobre a regra de escopo dinâmico podem ser vistas em Aho (1995). Neste trabalho será utilizada a regra de escopo estático por já ser utilizado pela linguagem FURBOL em trabalhos anteriores.

Para o escopo estático, pode-se utilizar duas abordagens: o escopo estático sem procedimentos aninhados e escopo estático com procedimentos aninhados. O primeiro aplica-se a linguagens como C onde os procedimentos não podem ser aninhados, ou seja, uma declaração de procedimento não pode figurar dentro de outra. O segundo aplica-se a linguagens como Pascal onde a declaração de procedimentos aninhados é permitida (Aho, 1995).

Maiores informações sobre o escopo estático sem procedimentos aninhados podem ser vistas em Aho (1995). Neste trabalho será utilizado o escopo estático com procedimentos aninhados.

2.6.1 ESCOPO ESTÁTICO COM PROCEDIMENTOS ANINHADOS

O uso de um nome não local a num procedimento Pascal é resolvido na declaração mais internamente aninhada de a num procedimento que envolve aquele onde a está sendo referenciado.

O aninhamento das definições de procedimentos no programa Pascal do quadro 32 é indicado pela indentação do quadro 33.

QUADRO 32 - PROGRAMA PASCAL COM PROCEDIMENTOS ANINHADOS

```

(1) program sort(input, output);
(2)   var a: array[0..10] of integer;
(3)     x: integer;

(4)   procedure readarray;
(5)     Var i: integer;
(6)     begin ... a ... end { readarray };

(7)   procedure exchange(i, j: integer);
(8)   begin
(9)     x := a[i]; a[i] := a[j]; a[j] := x;
(10)  end { exchange };

(11)  procedure quicksort(m, n: integer);
(12)    var k, v: integer;

(13)    Function partition(y, z: integer): integer;
(14)      var i, j: integer;
(15)      begin ... a ...
(16)                ... v ...
(17)                ... exchange (i,j); ...
(18)      end { partition };

(19)  begin ... end { quicksort };

(20) begin ... end { sort };

```

Fonte: Aho (1995)

QUADRO 33 - INDENTAÇÃO MOSTRANDO O ANINHAMENTO DE PROCEDIMENTOS DO QUADRO 32

```

      sort
        readarray
          exchange
            quicksort
              partition

```

Fonte: Aho (1995)

A ocorrência de *a*, na linha 15 no quadro 32, está dentro da função *partition*, a qual esta aninhada no procedimento *quicksort*. A declaração de *a* mais proximamente aninhada está à linha 2 do procedimento que consiste no próprio programa. A regra se aplica a nomes de procedimentos igualmente. O procedimento *exchange*, chamado por *partition* à linha 17, é não local a *partition*. Aplicando a regra, primeiro verifica-se se *exchange* está definido dentro de *quicksort*. Como não está, procura-se no programa principal *sort* (Aho, 1995).

2.6.1.1 REGRA DO ANINHAMENTO MAIS INTERNO

O escopo de uma declaração numa linguagem estrutura em blocos, como a linguagem *C*, é dado pela *regra de aninhamento mais interno* (Aho, 1995).

- a) o escopo de uma declaração num bloco *B* inclui *B*;
- b) se um nome *x* não for declarado num bloco *B*, uma ocorrência de *x* em *B* estará no escopo de uma declaração de *x* num bloco envolvente *B'* tal que:
 - *B'* possui uma declaração para *x*;
 - *B'* é o bloco mais internamente aninhado envolvendo *B*, em relação a qualquer outro bloco que contenha uma declaração para *x*.

2.6.1.2 PROFUNDIDADE DE ANINHAMENTO

A noção de *profundidade de aninhamento* de um procedimento é usada para implementar o escopo estático. Faça-se o nome do programa principal estar à profundidade 1. Adiciona-se 1 à profundidade de aninhamento à medida que se encaminha de um procedimento envolvente para um procedimento envolvido. No quadro 32, o procedimento *quicksort* à linha 11 está à profundidade de aninhamento 2, enquanto que *partition* à linha 13 está à profundidade de aninhamento 3. A cada ocorrência de um nome associa-se a profundidade de aninhamento do procedimento no qual é declarado. As ocorrências de *a*, *v* e *i*, às linhas 15-17 em *partition*, têm, por conseguinte, profundidades de aninhamento 1, 2 e 3, respectivamente (Aho, 1995).

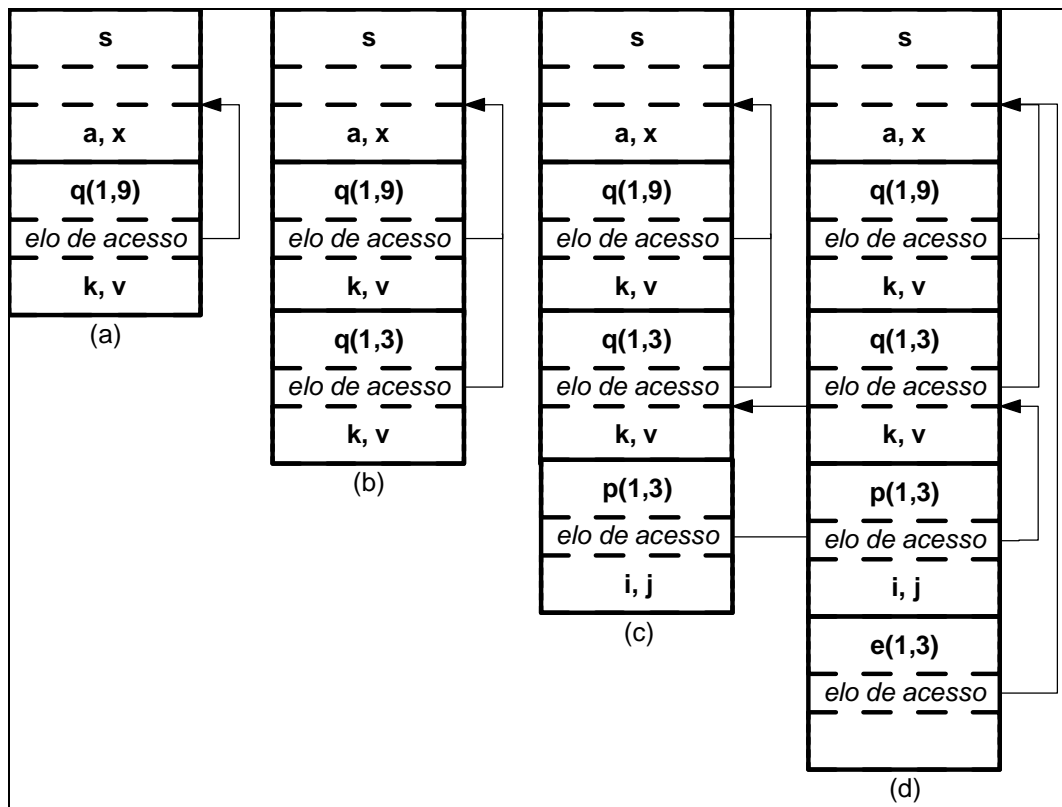
2.6.1.3 ELOS DE ACESSO

Uma implementação direta do escopo estático para procedimentos aninhados é obtida adicionando-se um apontador, chamado de *elo de acesso*, a cada registro de ativação. Se o procedimento *p* estiver aninhado imediatamente dentro de *q* no texto-fonte, então o elo de acesso num registro de ativação para *p* aponta para o elo de acesso do registro da ativação mais recente de *q* (Aho, 1995).

Instantâneos da pilha em tempo de execução, durante uma execução do programa do quadro 32 são mostrados na fig. 8. Para economizar espaço na figura, somente a primeira letra de cada nome de procedimento é mostrada. O elo de acesso para a ativação de *sort* está vazio, já que não há procedimento envolvente. O elo de acesso de *quicksort* aponta para o registro

de *sort*. Note-se que na fig. 8 o elo de acesso no registro de ativação para *partition(1,3)* (onde 1,3 são os valores dos parâmetros da chamada) aponta para o elo de acesso no registro de ativação mais recente de *quicksort*, explicitamente, *quicksort(1,3)* (Aho, 1995).

FIGURA 8 - ELOS DE ACESSO PARA SE ENCONTRAR O ARMAZENAMENTO DE NOMES NÃO-LOCAIS



Fonte: Aho (1995)

Suponha-se que o procedimento p à profundidade de aninhamento n_p se refira ao nome não local a com nível de aninhamento $n_a \leq n_p$. O armazenamento de a pode ser encontrada como segue (Aho, 1995):

- quando o controle estiver em p , um registro de ativação para p está ao topo da pilha. Seguir $n_p - n_a$ elos de acesso a partir do registro ao topo da pilha. O valor de $n_p - n_a$ pode ser pré-computado em tempo de compilação. Se o elo de acesso em um registro aponta o elo de acesso noutro, então um elo pode ser seguido realizando-se uma operação de indireção;
- após seguir $n_p - n_a$ elos, atinge-se um registro de ativação para o procedimento ao qual a é local. O armazenamento para a estará a um deslocamento fixo relativo a

posição do registro. Em particular, o deslocamento pode ser relativo ao elo de acesso.

Por exemplo, nas linhas 15-16 do quadro 32, o procedimento *partition*, à profundidade de aninhamento 3, referencia os nomes não-locais *a* e *v* às profundidades de aninhamento 1 e 2, respectivamente. Os registros de ativação contendo a memória para esses nomes não-locais são encontrados seguindo-se $3-1=2$ e $3-2=1$ elos de acesso, respectivamente, a partir do registro de *partition* (Aho, 1995).

O código para estabelecer os elos de acesso é parte da seqüência de chamada. Suponha-se que o procedimento *p* com profundidade de aninhamento n_p chame o procedimento *x* à profundidade de aninhamento n_x . O código para estabelecer o elo de acesso no procedimento chamado depende desse último estar aninhado ou não ao chamador, como segue (Aho, 1995):

- a) caso $n_p < n_x$: uma vez que o procedimento chamado *x* está aninhado mais profundamente do que *p*, terá que estar declarado dentro de *p*, ou não estaria acessível a *p*. Esse caso ocorre quando *sort* chama *quicksort* (fig. 8(a)) e quando *quicksort* chama *partition* (fig. 8(c)). Nesse caso, o elo de acesso no procedimento chamado terá de apontar para o elo de acesso no registro de ativação do chamador, exatamente abaixo na pilha;
- b) caso $n_p \geq n_x$: neste caso o procedimento chamado *x* não está aninhado no procedimento chamador *p* mas está acessível a ele. Esse caso acontece quando *quicksort* chama a si mesmo (fig. 8(b)) ou quando *partition* chama *exchange* (fig. 8(d)). Seguindo-se $n_p - n_x + 1$ elos de acesso a partir do chamador atinge-se o registro de ativação mais recente do procedimento que envolve mais proximamente tanto o procedimento chamado quanto o chamador no programa-fonte. O elo de acesso atingido é aquele para o qual o elo de acesso no procedimento chamado precisará apontar. Novamente, $n_p - n_x + 1$ pode ser computado em tempo de compilação.

2.7 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

O código intermediário é uma representação simplificada do programa-fonte. Nele as construções complexas do programa-fonte são quebradas em outras mais simples e os detalhes da máquina-alvo são abstraídas.

Apesar de se poder traduzir o programa-fonte diretamente na linguagem-alvo, alguns dos benefícios em se usar uma forma independente da máquina são:

- a) um compilador para uma máquina diferente pode ser criado atrelando-se um outro gerador de código a partir do código intermediário;
- b) um otimizador de código independente de máquina pode ser aplicado à representação intermediária.

O presente trabalho faz uso do código de três endereços para representar o código intermediário.

2.7.1 CÓDIGO DE TRÊS ENDEREÇOS

O código de três endereços é uma seqüência de enunciados da forma geral apresentada no quadro 34, onde x , y e z são nomes, constante ou objetos de dados temporários criados pelo compilador. O identificador op representa qualquer operador, como um operador aritmético ou lógico (Aho, 1995).

QUADRO 34 - UM ENUNCIADO DE CÓDIGO DE TRÊS ENDEREÇOS

$x := y \text{ op } z$

Fonte: Aho (1995)

Note-se que não são permitidas expressões aritméticas construídas, na medida em que só há um operador no lado direito de um enunciado. Por conseguinte, uma expressão de linguagem-fonte como $x + y * z$ poderia ser traduzida no código mostrado no quadro 35 (Aho, 1995).

QUADRO 35 - CÓDIGO DE TRÊS ENDEREÇOS PARA A EXPRESSÃO $X + Y * Z$

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

Fonte: Aho (1995)

No quadro 35, t_1 e t_2 são nomes temporários gerados pelo compilador. A simplificação das expressões aritméticas e de enunciados de fluxo de controle aninhados facilita a geração de código-alvo. O uso de nomes para valores intermediários computados por um programa permite que o código de três endereços seja facilmente reorganizado (Aho, 1995).

2.7.2 TRADUÇÃO DIRIGIDA PELA SINTAXE DO CÓDIGO DE TRÊS ENDEREÇOS

Quando o código de três endereços é gerado, os nomes temporários são construídos para o nós interiores da árvore sintática. O valor do não-terminal E ao lado esquerdo de $E \rightarrow E_1 + E_2$ 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 de $id.local := t$. Se uma expressão constituir-se em um único identificador y , o próprio y abrigará o valor da expressão.

A definição do quadro 36 gera o código de três endereços para enunciados de atribuição. Dada a entrada do quadro 37 a definição produz o código do quadro 38. O atributo sintetizado $S.Codigo$ representa o código de três endereços para a atribuição. O não-terminal E possui dois atributos: $E.local$ e $E.Codigo$. 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 . O símbolo \parallel representa uma concatenação entre cadeias de caracteres.

QUADRO 36 - DEFINIÇÃO DIRIGIDA PELA SINTAXE PARA PRODUZIR CÓDIGO DE TRÊS ENDEREÇOS PARA ATRIBUIÇÕES

Produção	Regras Semânticas
$S ::= id := E$	$S.Codigo := E.Codigo \parallel gerar (id.local := E.local)$
$E ::= E_1 + E_2$	$E.local := novo_temporário$ $E.Codigo := E_1.Codigo \parallel E_2.Codigo \parallel gerar (E.local := E_1.local + E_2.local)$
$E ::= E_1 * E_2$	$E.local := novo_temporário$ $E.Codigo := E_1.Codigo \parallel E_2.Codigo \parallel gerar (E.local := E_1.local * E_2.local)$
$E ::= -E_1$	$E.local := novo_temporário$ $E.Codigo := E_1.Codigo \parallel gerar (E.local := 'uminus' E_1.local)$
$E ::= (E_1)$	$E.local := E_1.local$ $E.Codigo := E_1.Codigo$
$E ::= \rightarrow id$	$E.local := id.local$ $E.Codigo := ''$

Fonte: adaptado de Aho (1995)

QUADRO 37 - EXEMPLO DE ENTRADA PARA O GERADOR DE CÓDIGO DE TRÊS ENDEREÇOS

$A := b * -c + b * -c$

Fonte: Aho (1995)

QUADRO 38 - CÓDIGO GERADO PELO GERADOR DE CÓDIGO INTERMEDIÁRIO PARA A ENTRADA DO QUADRO 37

$t_1 := -c$ $t_2 := b * t_1$ $t_3 := -c$ $t_4 := b * t_3$ $t_5 := t_2 + t_4$ $a := t_5$
--

Fonte: Aho (1995)

2.8 GERAÇÃO DE CÓDIGO-ALVO

Enquanto os detalhes são dependentes da máquina-alvo e do sistema operacional, temas como 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). Nesta seção, serão apresentados os temas genéricos do projeto de geradores de código.

2.8.1 PROGRAMAS-ALVO

A saída do gerador de código é o programa-alvo. Como o código intermediário, essa saída pode assumir uma variedade de formas (Aho, 1995):

- a) *linguagem absoluta de máquina*: possui a vantagem de poder ser carregado numa localização de memória fixa de memória e executado imediatamente;
- b) *linguagem relocável de máquina*: permite que subprogramas sejam compilados separadamente. Um conjunto de módulos-objeto relocáveis pode ser ligado e carregado para execução por um carregador/editor de ligações;
- c) *linguagem de montagem*: este tipo de saída facilita a geração de código. Pode-se gerar instruções simbólicas e usar as facilidades de processamento de macros do montador para auxiliar a geração de código.

2.8.2 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. Na seção 2.7 assumi-se que um nome num enunciado de três endereços se refere a uma entrada para o nome da tabela de símbolos. O tipo numa declaração determina a largura, isto é, a quantidade de memória necessitada para o nome declarado. As declarações apresentadas no quadro 39 informam ao compilador que as variáveis *i* e *j* são do tipo *integer* e necessitam do mesmo tamanho de memória para abrigar seus valores. A partir das informações da tabela de símbolos, pode ser determinado um endereço relativo para o nome na área de dados para o procedimento (Aho, 1995).

QUADRO 39 - EXEMPLO DE DECLARAÇÃO DE VARIÁVEIS EM PASCAL

```
var
  i, j: integer;
```

2.8.3 SELEÇÃO DE INSTRUÇÕES

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 (Aho, 1995).

A velocidade das instruções e os dialetos de máquina são fatores importantes. Se a preocupação com a eficiência do programa-alvo não existe, 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 do tipo mostrado no quadro 40, onde as localizações de memória para os nomes são alocadas estaticamente, pode ser traduzido na seqüência de código mostrada no quadro 41 (Aho, 1995).

QUADRO 40 - ENUNCIADO DE TRÊS ENDEREÇOS

```
x := y + z
```

Fonte: Aho (1995)

QUADRO 41 - CÓDIGO GERADO PARA O ENUNCIADO DE TRÊS ENDEREÇOS DO QUADRO 40

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

Fonte: Aho (1995)

Esse tipo de geração de código enunciado a enunciado, freqüentemente produz um código de baixa qualidade. Por exemplo, a seqüência de enunciados do quadro 42 seria traduzida no código apresentado no quadro 43.

QUADRO 42 - ENUNCIADOS DE TRÊS ENDEREÇOS

```

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

Fonte: Aho (1995)

QUADRO 43 - CÓDIGO INEFICIENTE GERADO PARA A SEQÜÊNCIA DE ENUNCIADOS DE TRÊS ENDEREÇOS DO QUADRO 42

```

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

Fonte: Aho (1995)

No quadro 43 o quarto enunciado é redundante, como também o será o terceiro caso *a* não venha a ser utilizado subseqüentemente.

A qualidade do código gerado é determinada por sua velocidade e tamanho. Uma máquina-alvo, com um rico conjunto de instruções, pode providenciar várias formas de se implementar uma dada operação. Uma vez que as diferenças de custo entre as diferentes implementações podem ser significativas, uma tradução ingênua do código intermediário pode levar a um código-alvo correto, porém inaceitavelmente ineficiente. Por exemplo, se a máquina-alvo possui uma instrução de incremento (*INC*), o enunciado de três endereços do quadro 44 pode ser implementado mais eficientemente pelo código do quadro 45 do que pela seqüência mais óbvia do quadro 46.

QUADRO 44 - INCREMENTO SIMPLES DE UMA VARIÁVEL

```

a := a + 1
```

Fonte: Aho (1995)

QUADRO 45 - GERAÇÃO DE CÓDIGO PARA O QUADRO 44 UTILIZANDO A INSTRUÇÃO *INC*

```

INC a
```

Fonte: Aho (1995)

QUADRO 46 - GERAÇÃO DE CÓDIGO PARA O QUADRO 44 UTILIZANDO VÁRIAS INSTRUÇÕES

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

Fonte: Aho (1995)

2.8.4 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 é freqüentemente subdividido em dois subproblemas (Aho, 1995):

- durante a *alocação de registradores*, seleciona-se o conjunto de variáveis que residirão nos registradores a um determinado ponto do programa;
- durante a fase subsequente 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.

2.8.5 MÁQUINA-ALVO

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

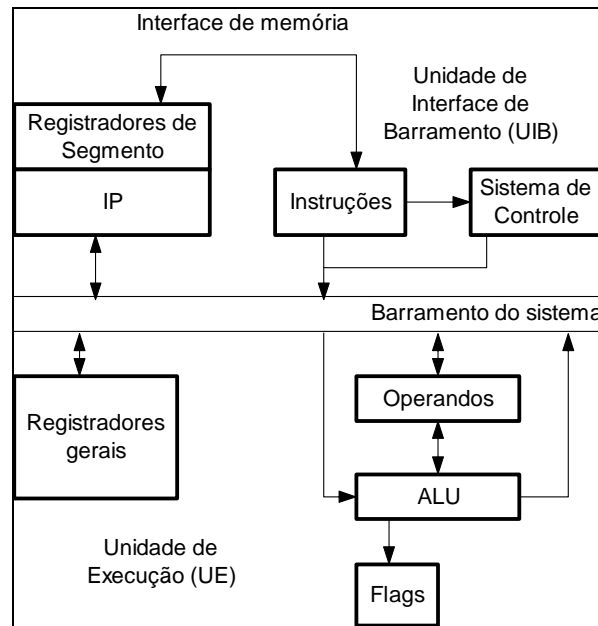
A máquina-alvo utilizada neste trabalho é o microprocessador 8088, explicado brevemente na seção seguinte. Maiores detalhes sobre a utilização da arquitetura do microprocessador 8088 no ambiente FURBOL podem ser vistos em Adriano (2001).

2.8.5.1 ARQUITETURA DOS MICROPROCESSADORES 8088

Segundo Yeung (1984), as funções internas do microprocessador 8088 são divididas em duas unidades de processamento, como mostrado na fig. 9.

A unidade de interface de barramento (UIB) serve funções relacionadas com obtenção de instruções, acesso a operandos e relocação de endereços. Ela também é responsável pelo controle básico do barramento. A unidade de execução (UE) recebe instruções da UIB com seu operandos. Os resultados são passados para a UIB na memória (Yeung, 1984).

FIGURA 9 - UNIDADE CENTRAL DE PROCESSAMENTO DO MICROPROCESSADOR 8088



Fonte: adaptado de Yeung (1984)

A unidade central de processamento possui registradores que podem ser usados para armazenar valores do programa que está sendo executado e definir comportamentos sobre a execução de instruções.

2.8.5.1.1 REGISTRADORES DE USO GERAL

Segundo Yeung (1984), o microprocessador 8088 possui um conjunto de 8 registradores de 16 bits para uso geral. Estes registradores estão divididos em dois subconjuntos:

- a) registradores de dados;
- b) registradores de apontadores e índices.

Os registradores de dados usados em operações de 16 bits são (Yeung, 1984):

- a) *AX*: acumulador;

- b) *BX*: base;
- c) *CX*: contador;
- d) *DX*: dados.

Os registradores de 8 bits são *AH, AL, BH, BL, CH, CL, DH, DL*. Os registradores de apontadores e índices são:

- a) *BP*: apontador de base;
- b) *SP*: apontador da pilha;
- c) *SI*: índice de origem;
- d) *DI*: índice de destino.

2.8.5.1.2 REGISTRADORES DE SEGMENTO

O microprocessador 8088 é capaz de endereçar um megabyte de memória. Este espaço de memória pode ser acessado através de quatro segmentos que podem ter até 64K bytes de tamanho. Estes segmentos são endereçáveis através de quatro registradores que são (Yeung, 1984):

- a) *CS*: segmento de código;
- b) *DS*: segmento de dados;
- c) *SS*: segmento de pilha;
- d) *ES*: segmento extra.

Um registrador adicional *IP* trabalha em conjunto com o registrador *CS* para indicar a localização da próxima instrução a ser executada.

2.8.5.1.3 REGISTRADOR DOS SINALIZADORES

O microprocessador 8088 possui nove sinalizadores, também conhecidos com *flags*, de 1 bit. Estes servem para indicar os resultados da última operação executada e para definir o comportamento do microprocessador na execução de instruções. Estes são divididos em dois tipos (Yeung, 1984):

- a) sinalizadores de status;
- b) sinalizadores de controle.

Os sinalizadores de status são (Yeung, 1984):

- a) *CF*: sinalizador de *carry*;

- b) *PF*: sinalizador de paridade;
- c) *AF*: sinalizador auxiliar;
- d) *ZF*: sinalizador de zero;
- e) *SF*: sinalizador de sinal;
- f) *OF*: sinalizador de estouro (*overflow*).

Os sinalizadores de controle são:

- a) *DF*: sinalizador de direção;
- b) *IF*: sinalizador de interrupções habilitadas;
- c) *TF*: permite a operação “passo a passo”. Este sinalizador não pode ser acessado pelo programa.

2.9 LINGUAGEM ASSEMBLY

A linguagem *Assembly* é uma linguagem de computador singular. O programa-fonte é composto de palavras de três ou quatro letras que representam os nomes das instruções, aparentemente sem haver uma ordem ou relacionamento umas com as outras.

2.9.1 CARACTERÍSTICAS GERAIS DO ASSEMBLY

Cada linha de comando é dividida em quatro campos que são separados por qualquer número de espaços (Yeung, 1984). O quadro 47 mostra o formato geral de uma linha de comando *Assembly*.

QUADRO 47 - FORMATO GERAL DE UMA LINHA DE COMANDO EM ASSEMBLY

[nome:] [operação] [operandos] [; comentários]
--

Fonte: adaptado de Yeung (1984)

Os campos mostrados no quadro 47 são (Yeung, 1984):

- a) *nome*: o nome, quando aparece, é a primeira entrada da linha, normalmente na coluna 1;
- b) *operação*: são os mnemônicos da linguagem *Assembly*;
- c) *operandos*: cada operando representa a origem ou o destino de dados que cada operação manuseia. Estes podem ser constantes, registradores de uso geral ou endereços de memória;

- d) *comentários*: comentários são precedidos por ponto-e-vírgula (;) e podem aparecer sozinhos ou depois dos campos. Servem para facilitar o entendimento do programa.

2.9.2 INSTRUÇÕES DA LINGUAGEM ASSEMBLY

As instruções da linguagem *Assembly* servem para indicar quais operações devem ser realizadas durante a execução do programa. O quadro 48 apresenta uma parte de um programa *Assembly* que zera duas mil posições de memória a partir do endereço 200h.

QUADRO 48 - EXEMPLO DE PROGRAMA ASSEMBLY

MOV	AX,200h	; endereço inicial -> AX
MOV	ES,AX	; -> registrado de segmento extra
MOV	DI,0	; 0 -> registrador indexador destino
MOV	CX,2000	; 2000 -> registrador contador
CLD		; limpa o sinalizador DF
MOV	AX,0	; limpa AX
REP	STOSW	; armazena AX em ES:DI, inc DI ; dec CX e repete até que CX = 0

Fonte: adaptado de Yeung (1984)

As instruções da linguagem *Assembly* podem ser classificadas em instruções para transferência de dados, aritméticas, lógicas, para controle de fluxo e para manipulação de cadeias.

2.9.2.1 INSTRUÇÕES PARA TRANSFERÊNCIA DE DADOS

A linguagem *Assembly* possui uma instrução para mover dados de uma localização para outra. Seus operandos podem ter 8 ou 16 bits de tamanho e estes podem ser registradores de uso geral, localização de memória ou dados imediatos (constantes). A forma geral da instrução é apresentada no quadro 49 (Yeung, 1984).

QUADRO 49 - INSTRUÇÃO DE MOVIMENTAÇÃO DE DADOS

Mnemônico	Operandos	Descrição
MOV	op_1, op_2	$op_1 := op_2$

Fonte: adaptado de Yeung (1984)

Também pode-se mover o endereço de um nome para um registrador. Isto pode ser feito através de quatro instruções mostradas no quadro 50 (Yeung, 1984).

QUADRO 50 - INSTRUÇÕES PARA MOVIMENTAÇÃO DE ENDEREÇOS

Mnemônico	Operandos	Descrição
MOV	<i>reg, offset nome</i>	Move o endereço de <i>nome</i> para <i>reg</i>
LEA	<i>reg, nome</i>	Move o endereço efetivo de <i>nome</i> para <i>reg</i>
LDS	<i>reg, nome</i>	Move o endereço de <i>nome</i> para <i>reg</i> + <i>DS</i>
LES	<i>reg, nome</i>	Move o endereço de <i>nome</i> para <i>reg</i> + <i>ES</i>

Fonte: Adaptado de Yeung (1984)

Ainda pode-se mover dados de dispositivos para registradores ou de registradores para dispositivos. Isto é feito através das instruções mostradas no quadro 51 (Yeung, 1984).

QUADRO 51 - INSTRUÇÕES PARA ENTRADA E SAÍDA DE DISPOSITIVOS

Mnemônico	Operandos	Descrição
IN	<i>reg, porta</i>	Move o dado de <i>porta</i> para <i>reg</i> . Se <i>porta</i> for um registrador, este indicará a porta a ser usada
OUT	<i>porta, reg</i>	Move o dado de <i>reg</i> para <i>porta</i> . Se <i>porta</i> for um registrador, este indicara qual a porta a ser usada

Fonte: adaptado de Yeung (1984)

2.9.2.2 INSTRUÇÕES ARITMÉTICAS

A linguagem *Assembly* possui quatro operações aritméticas básicas de 8 ou 16 bits, com e sem sinal (Yeung, 1984). A forma geral para as instruções aritméticas é mostrada no quadro 52.

QUADRO 52 - INSTRUÇÕES ARITMÉTICAS

Mnemônico	Operandos	Descrição
ADD	op_1, op_2	$op_1 := op_1 + op_2$
SUB	op_1, op_2	$op_1 := op_1 - op_2$
MUL (8 bits)	Op_1	$AX := AL * op_1$
MUL (16 bits)	op_1	$DX:AX := AX * op_1$
DIV (8 bits)	op_1	$AL := AX / op_1 \quad AH := AX \bmod op_1$
DIV (16 bits)	op_1	$AX := DX:AX / op_1 \quad DX := DX:AX \bmod op_1$

Fonte: adaptado de Yeung (1984)

Os operandos apresentados no quadro 52 podem ser registradores de 8 ou 16 bits e endereços de memória de 8 ou 16 bits.

2.9.2.3 INSTRUÇÕES LÓGICAS

A linguagem *Assembly* pode realizar quatro operações lógicas, apresentadas de forma geral no quadro 53 (Yeung, 1984).

QUADRO 53 - INSTRUÇÕES LÓGICAS

Mnemônico	Operandos	Descrição
AND	op_1, op_2	$op_1 := op_1 \text{ and } op_2$
OR	op_1, op_2	$op_1 := op_1 \text{ or } op_2$
XOR	op_1, op_2	$op_1 := op_1 \text{ xor } op_2$
NOT	op_1	$op_1 := \text{complemento de um de } op_1$

Fonte: adaptado de Yeung (1984)

Os operandos apresentados no quadro 53 podem ser valores imediatos, registradores de 8 ou 16 bits ou endereços de memória de 8 ou 16 bits.

2.9.2.4 INSTRUÇÕES PARA CONTROLE DE FLUXO

Instruções são obtidas da memória através dos registradores *CS* e *IP*. As instruções de controle de fluxo podem mudar o conteúdo destes registradores transferindo a execução do programa para um outro ponto especificado (Yeung, 1984). As instruções de desvio incondicional são mostradas no quadro 54.

QUADRO 54 - INSTRUÇÕES PARA DESVIO INCONDICIONAIS

Mnemônico	Operandos	Descrição
JMP	<i>label</i>	Desvia a execução para o rótulo <i>label</i>
JMP	<i>op₁</i>	Desvia indiretamente a execução para o endereço apontado por <i>op₁</i>
CALL	<i>name</i>	Chama o procedimento especificador por <i>name</i>
RET	<i>n</i>	Retorna de um procedimento e retira <i>n</i> localizações (<i>pop</i>) da pilha

Fonte: adaptado de Yeung (1984)

Nos quadros 55 e 56 são mostradas as instruções de desvio condicionais.

QUADRO 55 - INSTRUÇÕES DE DESVIO CONDICIONAL

Mnemônico	Operandos	Descrição
JA	<i>label</i>	Desvia se acima (CF ou ZF) = 0
JAE	<i>label</i>	Desvia se acima ou igual (CF = 0)
JB	<i>label</i>	Desvia se abaixo (CF = 1)
JBE	<i>label</i>	Desvia de abaixo ou igual (CF ou ZF) = 1
JC	<i>label</i>	Desvia se <i>carry</i> ligado (CF = 1)
JE	<i>label</i>	Desvia se igual (ZF = 1)
JG	<i>label</i>	Desvia se maior ((SF xor OF) ou ZF) = 0
JGE	<i>label</i>	Desvia se maior ou igual (SF xor OF) = 0
JL	<i>label</i>	Desvia se menor (SF xor OF) = 1
JLE	<i>label</i>	Desvia se menor ou igual (SF xor OF) ou ZF = 1
JNA	<i>label</i>	Desvia se não acima (CF ou ZF) = 1

Fonte: adaptado de Yeung (1984)

QUADRO 56 - INSTRUÇÕES DE DESVIO CONDICIONAL (CONTINUAÇÃO DO QUADRO 55)

JNAE	<i>label</i>	Desvia se não acima nem igual (CF = 1)
JNB	<i>label</i>	Desvia se não abaixo (CF = 0)
JNBE	<i>label</i>	Desvia se não abaixo nem igual (CF ou ZF) = 0
JNC	<i>label</i>	Desvia se <i>carry</i> desligado (CF = 0)
JNE	<i>label</i>	Desvia se não igual (ZF = 0)
JNO	<i>label</i>	Desvia se não estourou (OF = 0)
JNP	<i>label</i>	Desvia se não tem paridade (PF = 0)
JNS	<i>label</i>	Desvia se positivo (SF = 0)
JNZ	<i>label</i>	Desvia se não zero (ZF = 0)
JO	<i>label</i>	Desvia se estourou (OF = 1)
JP	<i>label</i>	Desvia se tem paridade (PF = 1)
JPE	<i>label</i>	Desvia se paridade par (PF = 1)
JPO	<i>label</i>	Desvia se paridade ímpar (PF = 0)
JS	<i>label</i>	Desvia se tem sinal (SF = 1)
JZ	<i>label</i>	Desvia se zero (ZF = 1)
JCXZ	<i>label</i>	Desvia se <i>CX</i> é zero
LOOP	<i>label</i>	Desvia se <i>CX</i> não é zero

Fonte: adaptado de Yeung (1984)

Também é possível utilizar instruções para realizar desvios para rotinas que tratam interrupções específicas da máquina-alvo. Estas instruções são mostradas no quadro 57.

QUADRO 57 - INSTRUÇÕES DE DESVIO PARA INTERRUPÇÕES

Mnemônico	Operandos	Descrição
INT	<i>n</i>	Chama a rotina de tratamento da interrupção <i>n</i> ;
INTO		Executa a interrupção quando em estouro (<i>overflow</i>);
IRET		Retorna de uma interrupção.

Fonte: adaptado de Yeung (1984)

2.9.2.5 INSTRUÇÕES PARA MANIPULAÇÃO DE CADEIAS

A linguagem *Assembly* tem um conjunto de instruções para realizar operações repetitivas. Estas instruções eliminam o processamento que o programa tem que fazer entre as iterações, tornando o programa mais eficiente. As instruções são mostradas no quadro 58 com seus nomes primitivos. Cada nome recebe um sufixo *b* ou *w* indicando o tamanho dos operandos (*byte* ou *word*).

QUADRO 58 - INSTRUÇÕES DE MANIPULAÇÃO DE CADEIAS

Mnemônico	Descrição
LODS	Carrega cada elemento da cadeia apontado por DS:SI no registrador AX. Atualiza SI;
STOS	Armazena o conteúdo do registrador AX na cadeia apontada por ES:DI. Atualiza DI;
MOVS	Move a cadeia apontada por DS:SI para a memória apontada por ES:DI. Atualiza SI e DI;
CMPS	Compara os elementos de duas cadeias, uma apontada por ES:DI e a outra por DS:SI. Atualiza SI e DI;
SCAS	Varre os elementos de uma cadeia apontada por ES:DI comparando cada elemento com o registrador AX e configurando os sinalizadores (<i>flags</i>) com o resultado da comparação. Atualiza DI.

Fonte: adaptado de Yeung (1984)

Também é possível realizar operações repetitivas através de *prefixos de repetição*, apresentados no quadro 59. Estes prefixos causam a execução repetitiva da próxima instrução até que o registrador *CX* seja zero (*CX* é decrementado em um a cada iteração).

QUADRO 59 - PREFIXOS DE REPETIÇÃO DE INSTRUÇÕES

Mnemônico	Descrição
REP	Repete a próxima instrução até que CX = 0
REPE	Repete a próxima instrução até que CX = 0 ou ZF = 1
REPZ	O mesmo que REPE
REPNE	Repete a próxima instrução até que CX = 0 ou ZF = 0
REPNZ	O mesmo que REPNE

Fonte: adaptado de Yeung (1984)

2.9.3 SUBPROGRAMAS

Quando a mesma seqüência de instruções precisa ser executada várias vezes em diferentes partes do programa, estas instruções podem ser definidas em um subprograma, na forma geral mostrada no quadro 60 (Yeung, 1984).

QUADRO 60 - FORMA GERAL DE UM SUBPROGRAMA NA LINGUAGEM
ASSEMBLY

```

nome      proc  <tipo>
...
...
nome      endp

```

Fonte: adaptado de Yeung (1984)

No quadro 60, *tipo* pode ser *near* or *far* indicando o tamanho da instrução de retorno (2 bytes para *near* e 4 bytes para *far*) (Yeung, 1984).

Cada programa *Assembly* é dividido em um módulo principal e um ou mais subprogramas. O módulo principal chama um subprograma através da instrução *call* e o subprograma retorna através da instrução *ret*, fazendo com que a execução retorne para a instrução imediatamente após a chamada do subprograma (Yeung, 1984).

A passagem de parâmetros para subprogramas pode ser feita através de registradores, localizações de memória e através da pilha de execução (Yeung, 1984).

2.10 SERVIÇOS DO SISTEMA OPERACIONAL DOS

O DOS possui um grande número de funções que podem ser chamadas por programas. Normalmente os parâmetros de entrada e saída destes serviços são passados através de registradores (Quadros, 1988).

2.10.1 FORMA DE CHAMAR OS SERVIÇOS DO DOS

A forma mais comum de se chamar serviços do DOS é através da interrupção 21h (hexadecimal), o qual consiste em carregar o número da função desejada no registrador AH e desviar a execução para o tratador de interrupção 21h. Um exemplo pode ser visto no quadro 61.

QUADRO 61 - PROGRAMA *ASSEMBLY* QUE USA SERVIÇOS DO DOS

MOV	BX, 0FFFFH	
MOV	AH, 48H	
INT	21H	; BX = TAMANHO DO MAIOR BLOCO
MOV	AH, 48H	
INT	21H	; ALOCA: AX=ENDEREÇO, BX=TAMANHO

Fonte: adaptado de Quadros (1988)

2.10.2 SERVIÇOS DO DOS

Nesta seção serão apresentados os serviços do DOS, enfatizando as funções para gerenciamento de memória utilizadas no trabalho.

2.10.2.1 FUNÇÕES DE CARACTERE

Estas funções (quadro 62) permitem o acesso a dispositivos de entrada e saída padrões e impressora (Quadros, 1988).

Mais informações sobre as funções de caractere do DOS pode ser vista em Quadros (1988).

QUADRO 62 - FUNÇÕES DE CARACTERES DO DOS

Código	Descrição
01h	Leitura de console
02h	Escrita no console
03h	Leitura do dispositivo auxiliar
04h	Escrita no dispositivo auxiliar
05h	Escrita na impressora
06h	Entrada e saída direta no console
07h	Entrada direta do console, sem eco
08h	Entrada do console, sem eco
09h	Escrita de cadeia no console
0Ah	Leitura do console com edição
0Bh	Informa estado do console
0Ch	Leitura do console com espera

Fonte: adaptado de Quadros (1988)

2.10.2.2 FUNÇÕES DE ARQUIVO/DIRETÓRIO

Estas funções permitem acesso a dispositivos, arquivos e diretórios do DOS (Quadros, 1988). As principais funções de arquivo/diretório do DOS estão listadas no quadro 63. Mais informações sobre funções de arquivos/diretórios pode ser vistas em Quadros (1988).

QUADRO 63 - FUNÇÕES DE ARQUIVO/DIRETÓRIO

Código	Descrição
0Dh	Reinicia disco
0Eh	Seleciona unidade padrão
0Fh	Abertura de arquivo, via FCB
10h	Fechamento de arquivo, via FCB
11h	Busca primeira entrada, via FCB
12h	Busca próxima entrada, via FCB
13h	Suprime arquivo, via FCB
14h	Leitura seqüencial, via FCB

Fonte: adaptado de Quadros (1988)

QUADRO 64 - FUNÇÕES PARA ARQUIVO/DIRETÓRIO (CONTINUAÇÃO DO QUADRO 63)

15h	Escrita seqüencial, via FCB
16h	Cria arquivo, via FCB
17h	Renomeia arquivo, via FCB
19h	Informa unidade padrão
1Ah	Altera DTA
1Bh	Obtém informações da FAT da unidade padrão
1Ch	Obtém informações da FAT
21h	Leitura randômica, via FCB
22h	Escrita randômica, via FCB
23h	Informa tamanho de arquivo, via FCB
24h	Posiciona registro relativo
27h	Leitura de bloco, via FCB
28h	Escrita de bloco, via FCB
29h	Analisa nome de arquivo
2Eh	Liga/desliga opção de verificação

Fonte: adaptado de Quadros (1988)

2.10.2.3 FUNÇÕES DE GERENCIAMENTO DE MEMÓRIA

A partir da versão 2.0 o DOS dispõe de um gerenciamento de memória, controlando quais blocos estão livres e quais estão em uso (Quadros, 1988).

Todo bloco de memória começa em um endereço múltiplo de 16 bytes (um parágrafo) e possui nos 16 bytes anteriores um bloco de controle para este bloco com o formato mostrado no quadro 65 (Quadros, 1988).

QUADRO 65 - FORMATO DE UM BLOCO DE CONTROLE DE MEMÓRIA NO DOS

Deslocamento	Tamanho	Conteúdo
0	1 byte	'Z' se último bloco
1	2 bytes	0 se bloco livre Prefixo de Segmento de Programa de quem alocou se em uso
3	2 bytes	tamanho do bloco em parágrafos
5	11 bytes	Reservado

Fonte: adaptado de Quadros (1988)

O usuário não tem acesso aos blocos de controle, sendo utilizado nas chamadas dos serviços do DOS o endereço do bloco de memória (Quadros, 1988). Também nas funções de alocação de memória, o DOS retorna o endereço inicial do bloco de memória e não o endereço do bloco de controle.

Normalmente, quando um programa é carregado pelo DOS, toda a memória disponível está alocada para ele (a não ser que o arquivo executável especifique diferente) (Quadros, 1988). Para fazer uso das funções de alocação de memória no DOS é preciso liberar esta memória alocada pelo sistema operacional.

A seguir serão apresentadas as funções para gerenciamento de memória no DOS.

2.10.2.3.1 ALOCA MEMÓRIA (FUNÇÃO 48H)

Esta função está presente no DOS somente a partir da versão 2.0. Esta função procura um bloco livre com o tamanho maior ou igual ao desejado. Se encontrar, quebra-o em duas partes, alocando a primeira ao programa e marcando o restante como livre. Caso contrário, informa o tamanho do maior bloco disponível (Quadros, 1988).

Para usar a função deve-se informar o número de parágrafos (blocos de 16 bytes) que devem ser alocados no registrador BX. Se a função for executada com sucesso, o registrador AX conterá o endereço do bloco alocado. Caso contrário, o registrador BX conterá o tamanho do maior bloco disponível.

2.10.2.3.2 LIBERA BLOCO DE MEMÓRIA (FUNÇÃO 49H)

Esta função libera totalmente um bloco de memória, marcando-o como livre. Para liberar parte de um bloco de memória, deve-se utilizar a função 4Ah (Quadros, 1988).

Para utilizar a função deve-se informar no registrador ES o endereço inicial do bloco de memória. Se a função for executada com sucesso, o DOS terá liberado a memória.

2.10.2.3 MODIFICA TAMANHO DE BLOCO DE MEMÓRIA (FUNÇÃO 4AH)

Esta função permite diminuir ou aumentar o tamanho de um bloco de memória. Só é possível aumentar se existir um bloco livre contíguo ao bloco em questão (Quadro, 1988).

Para usar a função é necessário informar o endereço inicial do bloco atual no registrador ES. O tamanho do novo bloco deve ser informado no registrador BX. Se a função for executada com sucesso, o DOS terá aumentado o tamanho do bloco de memória conforme a requisição.

2.10.2.4 FUNÇÕES DE GERENCIAMENTO DE PROGRAMAS

Estas funções (quadro 66) permitem que os programas sejam gerenciados através do DOS (Quadros, 1988). Mais informações sobre funções de gerenciamento de programas do DOS podem ser vistas em Quadros (1988).

QUADRO 66 - FUNÇÕES DE GERENCIAMENTO DE PROGRAMAS DO DOS

Código	Descrição
00h	Termina programa
26h	Cria nova Prefixo de Segmento de Programa

Fonte: adaptado de Quadros (1988)

2.10.2.5 OUTRAS FUNÇÕES

Somente as principais funções estão listadas no quadro 67. Mais informações sobre funções do DOS podem ser vistas em Quadros (1988).

QUADRO 67 - OUTRAS FUNÇÕES DO DOS

Código	Descrição
25h	Altera vetor de interrupção
2Ah	Informa data atual
2Bh	Altera data atual
2Ch	Informa hora atual
2Dh	Altera hora atual
30h	Informa versão do DOS

Fonte: adaptado de Quadros (1988)

2.11 AMBIENTE FURBOL

O ambiente FURBOL vem sendo desenvolvido na Universidade Regional de Blumenau, objetivando o entendimento da construção de compiladores e 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 foi utilizada a notação BNF. Para montagem da definição de escopo e geração de código foram utilizadas ações semânticas, descritas através de gramática de atributos. O código descrito é também gerado em código de máquina para o microprocessador 8088.

2.11.1 PRINCIPAIS CARACTERÍSTICAS DA LINGUAGEM FURBOL

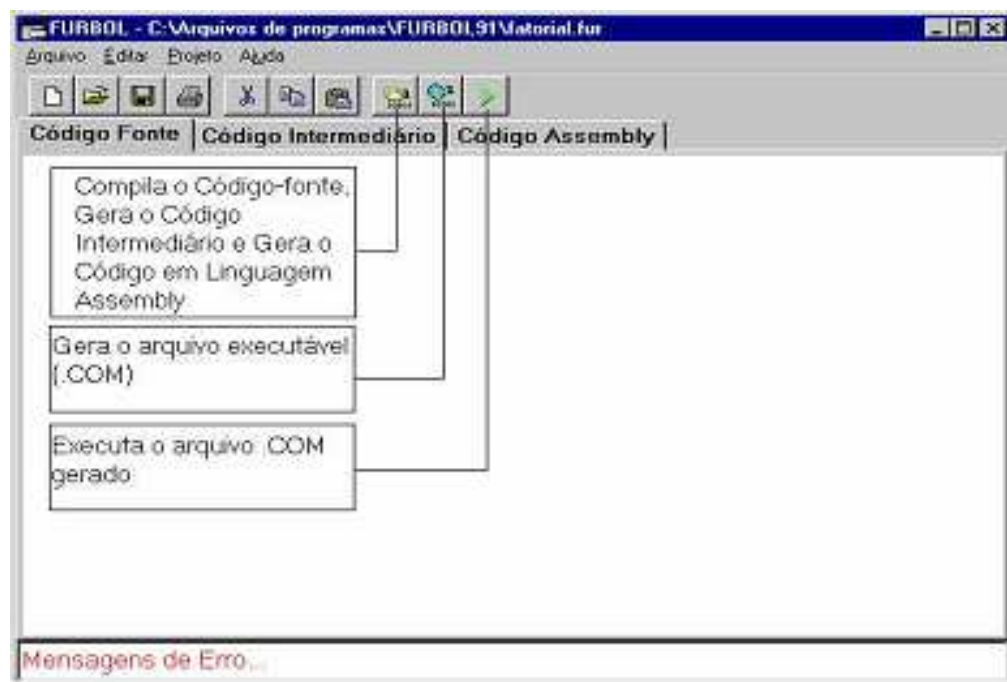
As características principais da linguagem FURBOL são:

- a) utilização de dados do tipo *inteiro*, *lógico* e *array semi-estático*;
- 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 parâmetros por cópia-valor e por referência.

2.11.2 INTERFACE DO AMBIENTE FURBOL

O protótipo desenvolvido possui a janela principal onde são editados os programas fonte. Esta janela possui um menu com funções disponíveis. Possui também uma barra de ferramentas onde podem ser acionadas as principais funções do menu. Esta janela principal é mostrada na fig. 10.

FIGURA 10 - TELA PRINCIPAL DO AMBIENTE FURBOL



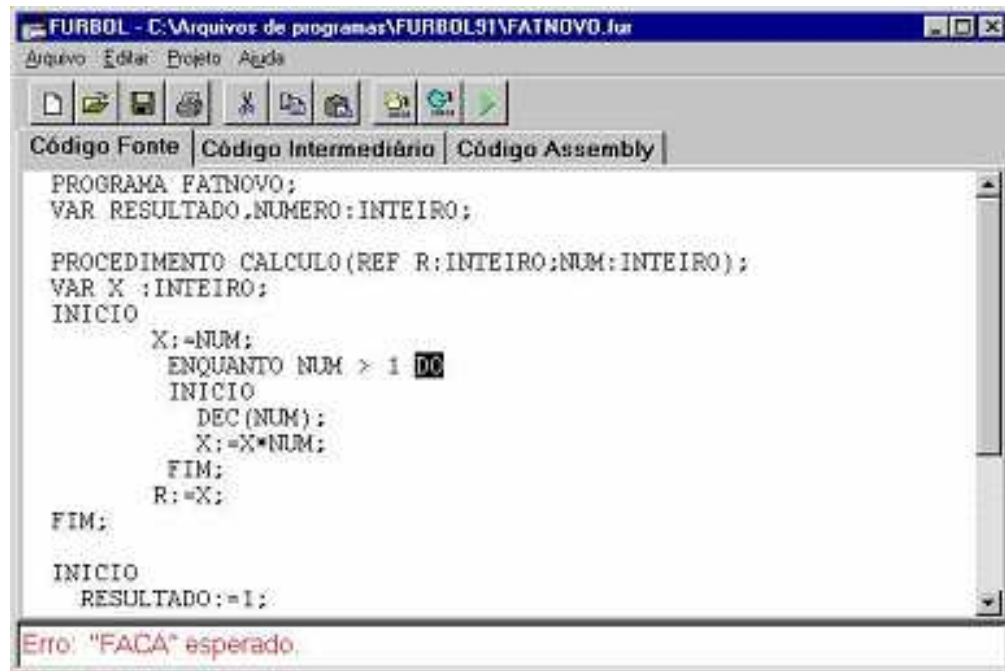
A opção “Projeto” do menu principal oferece três funções que são:

- a) *compilar*: que efetua a compilação do programa editado informando a existência ou não de erros (fig. 11). 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” (fig. 12) e o código em linguagem *Assembly* será criado na guia “Código Assembly”, como mostrado na fig. 13;
- b) *gerar código*: gera o código em linguagem de máquina a partir do código em linguagem *Assembly*. O montador *Turbo Assembler* (Borland, 1988) é acionado e

um programa com a extensão *.COM* é criado a partir do arquivo *.ASM* que contém 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 11 - DETECÇÃO DE ERROS NO PROTÓTIPO



The screenshot shows the FURBOL IDE window titled "FURBOL - C:\Arquivos de programas\FURBOL91\FATNOVO.fur". The menu bar includes "Arquivo", "Editar", "Projeto", and "Ajuda". The toolbar contains icons for file operations and execution. The main window has three tabs: "Código Fonte", "Código Intermediário", and "Código Assembly", with "Código Fonte" selected. The source code is as follows:

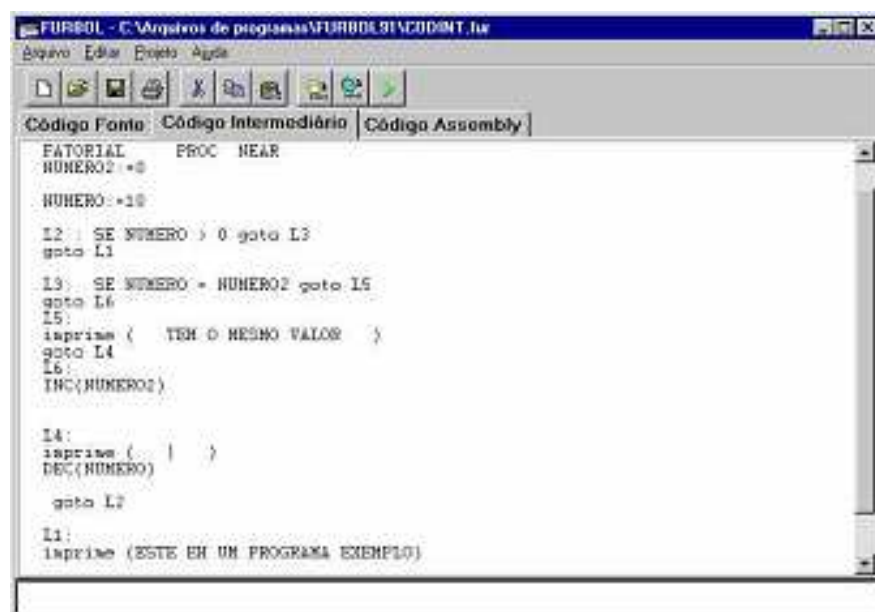
```
PROGRAMA FATNOVO;
VAR RESULTADO,NUMERO:INTEIRO;

PROCEDIMENTO CALCULO(REF R:INTEIRO;NUM:INTEIRO);
VAR X :INTEIRO;
INICIO
    X:=-NUM;
    ENQUANTO NUM > 1 DO
    INICIO
        DEC(NUM);
        X:=X*NUM;
    FIM;
    R:=X;
FIM;

INICIO
    RESULTADO:=1;
```

At the bottom of the window, a red error message reads: "Erro: 'FACA' esperado."

FIGURA 12 - JANELA PRINCIPAL COM CÓDIGO INTERMEDIÁRIO



The screenshot shows the FURBOL IDE window titled "FURBOL - C:\Arquivos de programas\FURBOL91\CODINT.fur". The menu bar includes "Arquivo", "Editar", "Projeto", and "Ajuda". The toolbar contains icons for file operations and execution. The main window has three tabs: "Código Fonte", "Código Intermediário", and "Código Assembly", with "Código Intermediário" selected. The intermediate code is as follows:

```
FATORIAL PROC NEAR
NUMERO2:=0
NUMERO:=10

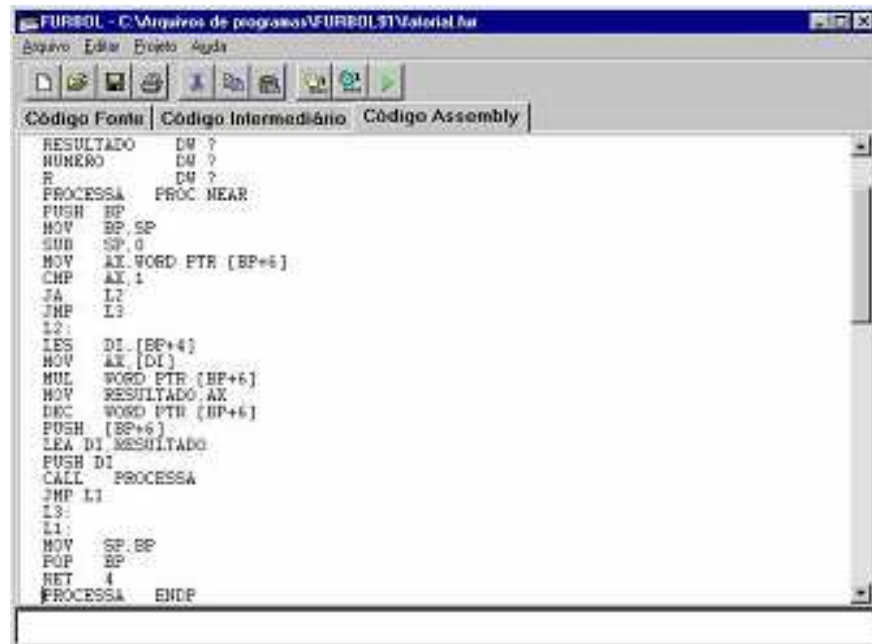
L2: SE NUMERO > 0 goto L3
goto L1

L3: SE NUMERO = NUMERO2 goto L5
goto L6
L5:
imprime ( TEM O MESMO VALOR )
goto L4
L6:
INC(NUMERO2)

L4:
imprime ( . | )
DEC(NUMERO)
goto L2

L1:
imprime (ESTE EH UM PROGRAMA EXEMPLO)
```


FIGURA 13 - JANELA PRINCIPAL COM CÓDIGO ASSEMBLY



The image shows a screenshot of the MASM32 IDE window titled "MASM32 - C:\Arquivos de programas\MASM32\Material As". The window has a menu bar with "Arquivo", "Editar", "Projeto", and "Ajuda". Below the menu bar is a toolbar with various icons. The main area is divided into three tabs: "Código Fonte", "Código Intermediário", and "Código Assembly", with the "Código Assembly" tab selected. The assembly code is displayed in the main area, showing a procedure named "PROCESSA" that calculates the factorial of a number stored in "NUMERO".

```
RESULTADO    DW ?
NUMERO       DW ?
PROCESSA     PROC NEAR
    PUSH    BP
    MOV     BP, SP
    SUB     SP, 0
    MOV     AX, WORD PTR [BP+6]
    CMP     AX, 1
    JA      L2
    JMP     L3
L2:
    LES     DI, [BP+4]
    MOV     AX, [DI]
    MUL     WORD PTR [BP+6]
    MOV     RESULTADO, AX
    DEC     WORD PTR [BP+6]
    PUSH   [BP+6]
    LEA    DI, RESULTADO
    PUSH  DI
    CALL  PROCESSA
    JMP   L1
L3:
L1:
    MOV   SP, BP
    POP  BP
    RET  4
PROCESSA ENDP
```

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo será apresentado o desenvolvimento do protótipo, conforme a definição proposta. Esta definição consiste na revisão da especificação proposta por Adriano (2001) com o acréscimo de produções para a implementação de *arrays* dinâmicos e a revisão da definição de escopo da linguagem para permitir que sejam acessadas variáveis não-locais.

O nome FURBOL é a concatenação da sigla da Universidade Regional de Blumenau (FURB) com ALGOL, que é uma linguagem que usa estrutura de blocos para controlar o escopo de variáveis e a divisão do programa em unidades.

O ambiente FURBOL foi implementado no ambiente Borland Delphi 5 (Cantu, 2000) utilizando a linguagem *Object Pascal*. O código já implementado por Adriano (2001) foi aproveitado.

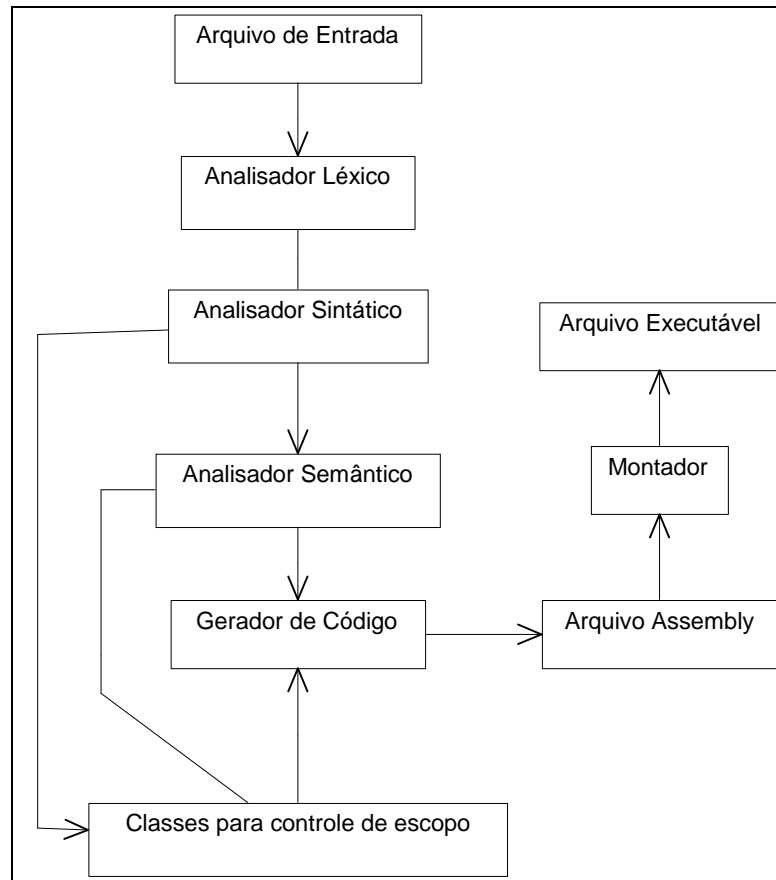
A implementação do protótipo consiste em quatro etapas: definição de escopo da linguagem, definição do *array* dinâmico, especificação formal da linguagem, implementação e apresentação do protótipo.

3.1 APRESENTAÇÃO GERAL DO AMBIENTE

A fig. 14 apresenta o diagrama geral do ambiente, onde são apresentadas as partes que compõem o ambiente e suas relações. As setas indicam o caminho da informação enquanto que as relações sem setas são bidirecionais.

O controle das informações sobre os símbolos (variáveis, procedimentos e *arrays*) e escopo está concentrado no item *Classes para controle de escopo*. Este controle é realizado por um conjunto de classes que são usadas pelos analisadores sintático e semântico e pelo gerador de código. O analisador sintático é o gerenciador das atividades do compilador.

FIGURA 14 - DIAGRAMA GERAL DO AMBIENTE



3.2 DEFINIÇÃO DE ESCOPO DA LINGUAGEM

O escopo da linguagem FURBOL segue a regra do aninhamento mais interno (2.6.1.1), ou seja, a procura dos nomes durante a compilação parte do escopo local e segue para a unidade envolvente até que se tenha chegado na unidade mais envolvente possível ou se tenha encontrado o nome. O nome usado em uma determinada parte do programa-fonte será aquele declarado num escopo acessível ao escopo local (onde a variável está sendo usada) e mais aninhado possível. Esta regra é implementada usando-se *elos de acesso* (2.6.1.3).

3.2.1 IMPLEMENTAÇÃO DE ELOS DE ACESSOS NA LINGUAGEM FURBOL

Para que se possa ter acesso a variáveis não-locais, é necessário também ter acesso ao registro de ativação onde a variável não-local está declarada. O registro de ativação que deve ser acessado é determinado pelo tipo de escopo que a linguagem implementa. A linguagem

FURBOL implementa escopo estático com procedimento aninhados (2.6.1). Para que seja possível o acesso a variáveis não-locais é necessário utilizar os elos de acesso.

Na linguagem FURBOL os elos de acesso foram implementados como ponteiros para a base das variáveis no registro de ativação do pai estático. A cada chamada de procedimento, empilham-se os parâmetros (se existirem), a base da pilha de execução e realiza-se a chamada. A base da pilha vai servir de elo de acesso para as variáveis do pai estático.

O quadro 68 mostra um programa FURBOL que faz acesso a variáveis não-locais.

QUADRO 68 - PROGRAMA FURBOL COM ACESSO A VARIÁVEIS NÃO-LOCAIS

```

programa Escopo;
  procedimento P;
  var
    VarLocalP: Inteiro;

    procedimento Q;
    var
      VarLocalQ: Inteiro;
    inicio
      VarLocalQ := VarLocalP;
    fim;
  inicio
    Q;
  fim;

inicio
  P;
fim.

```

O código em linguagem *Assembly* gerado para este programa é apresentado no quadro 69. O código *Assembly* gerado possui a instrução *MOV DI,[BP+4]*. Esta instrução move o elo de acesso para o registrador *DI*. Logo após um acesso a variável com deslocamento 2 é feito usando-se este registrador. Este deslocamento é o deslocamento da variável em questão dentro do registro de ativação da unidade onde ela está declarada.

QUADRO 69 - CÓDIGO ASSEMBLY GERADO PARA O PROGRAMA DO QUADRO 68

```

CODIGO    SEGMENT
ASSUME    CS:CODIGO, DS:CODIGO
          ORG    100H
ENTRADA:  JMP    ESCOPO

P  PROC NEAR                                ; Início do proc. P
PUSH     BP
MOV      BP,SP
SUB      SP,2
PUSH     BP                                ; Empilhar o elo de acesso
CALL     Q                                  ; Chamada do proc. Q por P
MOV      SP,BP
POP      BP
RET      2
P  ENDP

Q  PROC NEAR                                ; Início do procedimento Q
PUSH     BP
MOV      BP,SP
SUB      SP,2
MOV      DI,[BP+4]                          ; DI tem o elo de acesso
MOV      AX,WORD PTR [DI-2]                  ; É usado o deslocamento
MOV      WORD PTR [BP-2],AX                  ; variável no registro
MOV      SP,BP                              ; onde ela está
POP      BP
RET      2                                  ; Instrução RET retira o elo da pilha
Q  ENDP

ESCOPO   PROC NEAR
PUSH     BP
MOV      BP,SP
PUSH     BP
CALL     P
POP      BP
int     20h
ESCOPO   ENDP

ULTIMO_ROTULO:
CODIGO   ENDS
END      ENTRADA

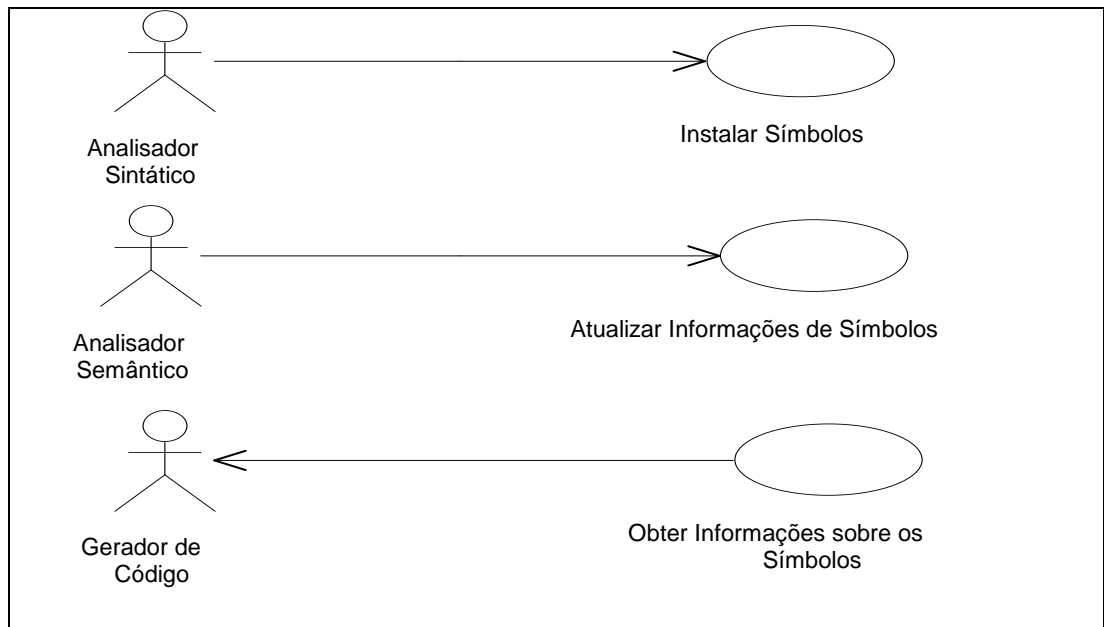
```

3.2.2 ESPECIFICAÇÃO DAS CLASSES DE CONTROLE DE ESCOPO

A fig. 15 apresenta o diagrama de caso de uso em UML (Furlan, 1998) do controle de escopo implementado. Este diagrama mostra os três principais processos desenvolvidos pelas classes responsáveis pelo escopo. O processo *Instalar Símbolos* (fig. 15) é responsável pela entrada dos símbolos encontrados no programa-fonte. A entrada para este processo é o nome do símbolo encontrado. O resultado deste processo é a instalação do novo símbolo no escopo atual. O processo *Atualizar Informações de Símbolos* é utilizado pelo analisador semântico.

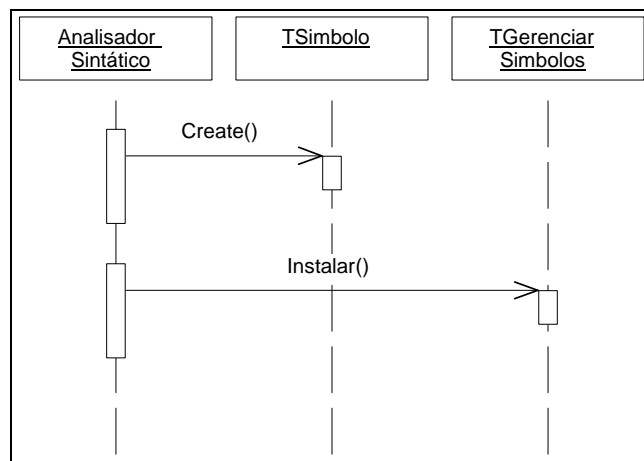
As entradas são informações sobre os símbolos (tipo, dimensões etc.) e o resultado é a atualização das informações do símbolo. O processo *Obter Informações sobre os Símbolos*, utilizado pelo gerador de código, tem como entrada o nome do símbolo e como resultado as informações (tipo, dimensões etc.) referentes ao símbolo requisitado.

FIGURA 15 - DIAGRAMA DE CASO DE USO PARA O CONTROLE DE ESCOPO



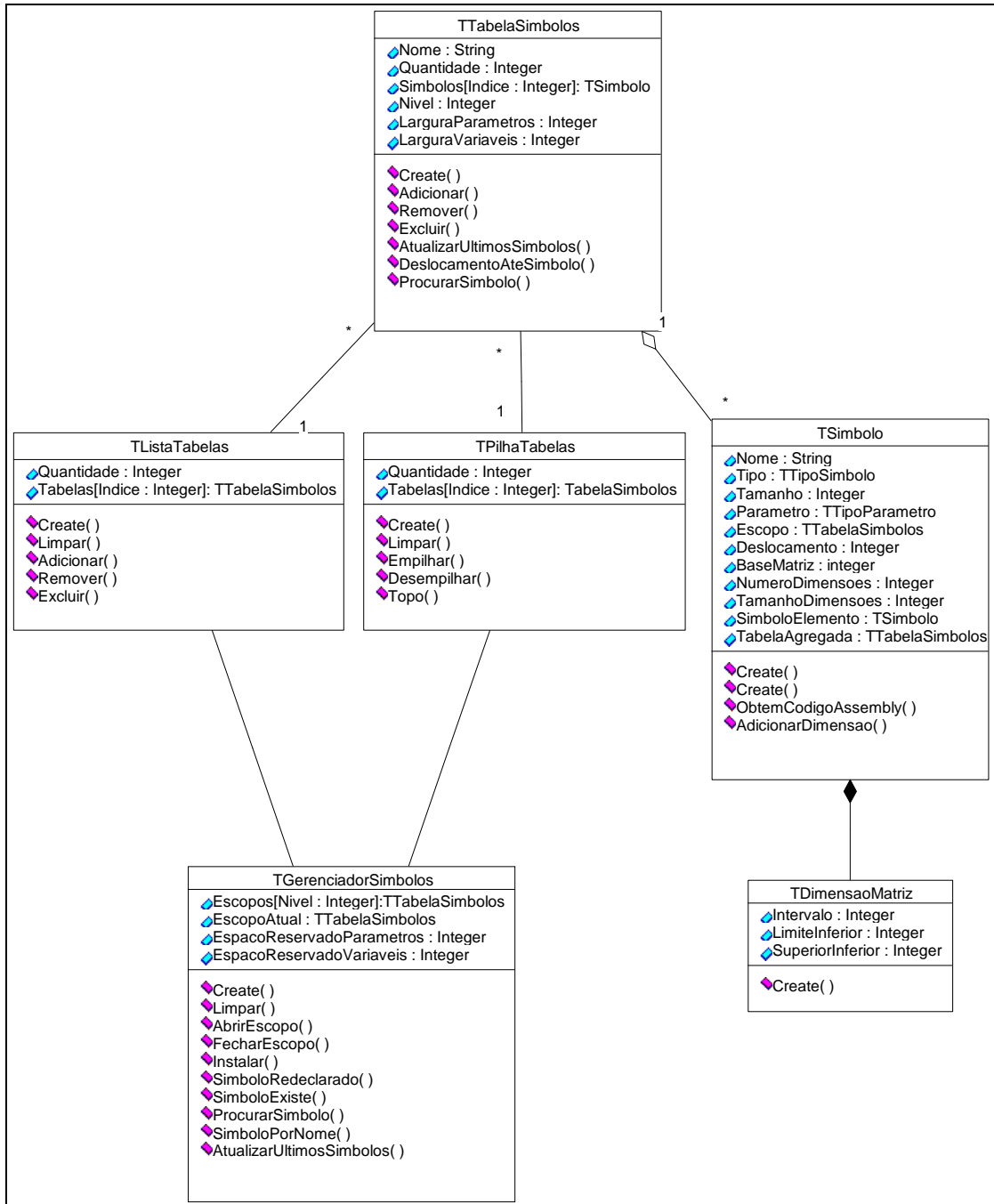
A fig. 16 apresenta o diagrama de seqüência para o processo *Instalar Símbolos*. Os demais processos apenas requisitam o símbolo de um determinado nome e alteram dados diretamente neste símbolo.

FIGURA 16 - DIAGRAMA DE SEQÜÊNCIA PARA O PROCESSO *INSTALAR SÍMBOLOS*



A fig. 17 apresenta o diagrama de classe do controle de escopo implementado no protótipo. Neste diagrama as assinaturas dos métodos foram removidas para ressaltar o relacionamento entre as classes. A classe mais importante para o controle de escopo da linguagem é a classe *TGerenciadorSimbolos*.

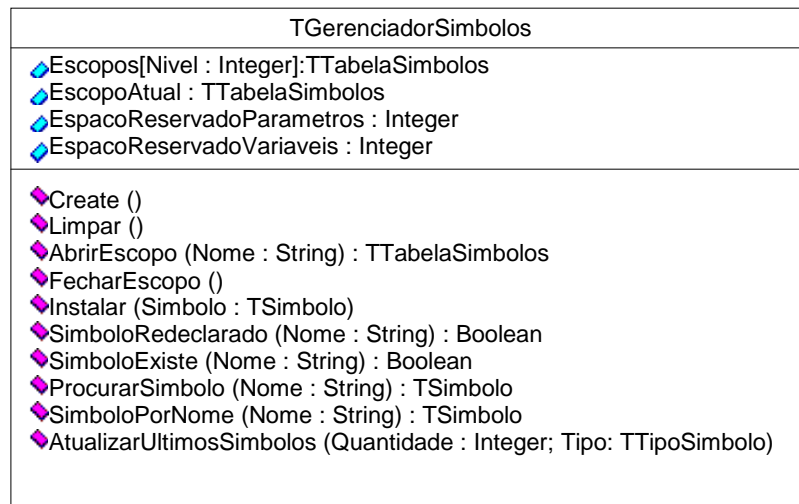
FIGURA 17 - DIAGRAMA DE CLASSES PARA O CONTROLE DE ESCOPO



3.2.2.1 GERENCIADOR DE SÍMBOLOS

A classe *TGerenciadorSimbolos* é a interface entre o compilador e os símbolos. A instalação, a procura e a atualização de símbolos devem ser realizadas utilizando-se esta classe. A fig. 18 mostra todas as operações disponíveis desta classe.

FIGURA 18 - ESPECIFICAÇÃO DA CLASSE *TGERENCIADORSIMBOLOS*



Os atributos da classe *TGerenciadorSimbolos* são:

- Escopos*: permite o acesso aos escopos ativos durante a compilação;
- EscopoAtual*: indica a tabela de símbolos do escopo atual;
- EspacoReservadoParametros*: indica o número de bytes reservados para o compilador na alocação de espaço para parâmetros;
- EspacoReservadoVariaveis*: indica o número de bytes reservados para o compilador na alocação de espaço para variáveis.

As principais operações da classe *TGerenciadorSimbolos* são:

- AbrirEscopo*: responsável pela criação de um novo escopo. Retorna a tabela de símbolos que representa o escopo recém-aberto. As tabelas de símbolos quando são criadas são colocadas em uma pilha que representam os escopos abertos. O escopo mais ao topo da pilha é o escopo local;
- FecharEscopo*: fecha o último escopo aberto;
- Instalar*: coloca o símbolo indicado na tabela de símbolos do escopo atual;

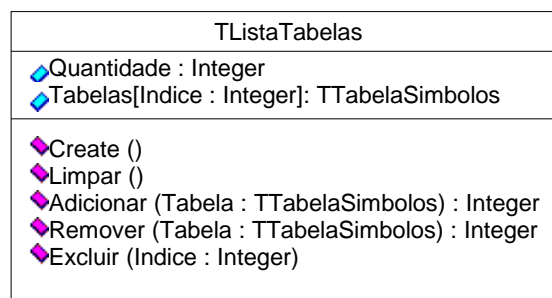
- d) *SimboloRedeclarado*: procura o símbolo no escopo atual e retorna verdadeiro se o mesmo foi encontrado. Esta operação é usada para descobrir se um mesmo símbolo está sendo declarado duas vezes no mesmo escopo;
- e) *SimboloExiste*: retorna verdadeiro caso o símbolo especificado seja encontrado. Esta procura acontece na pilha de escopos abertos partindo do último escopo aberto (escopo atual) até o primeiro escopo (escopo global);
- f) *ProcurarSimbolo*: retorna o símbolo associado ao nome especificado. A procura acontece da mesma maneira que é feita na operação *SimboloExiste*;
- g) *SimboloPorNome*: retorna o símbolo associado a um nome. Se um símbolo com o nome especificado não for encontrado, uma exceção é levantada e é capturada pelo compilador que relata posteriormente ao usuário que o símbolo não foi encontrado.

A classe *TGerenciadorSimbolos* usa duas classes auxiliares para armazenar os símbolos: *TListaTabelas* e *TPilhaTabelas*.

3.2.2.2 LISTA DE TABELAS DE SÍMBOLOS

A classe *TListaTabelas* armazena as tabelas de símbolos que estão ou foram usadas durante a compilação. Além de permitir a consulta de símbolos durante todo o processo de compilação, esta classe garante a liberação de memória alocada para os símbolos, uma vez que ela mantém a lista de símbolos até o final da compilação. A fig. 19 mostra a sua especificação.

FIGURA 19 - ESPECIFICAÇÃO DA CLASSE *TLISTATABELAS*



As principais operações da classe *TListaTabelas* são:

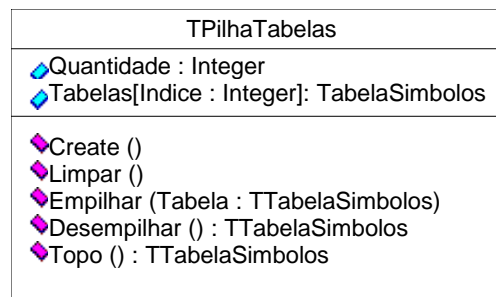
- a) *Adicionar*: adiciona a tabela de símbolos especificada ao final da lista;
- b) *Remover*: remove a tabela de símbolos especifica da lista;

c) *Excluir*: remove a tabela de símbolos no índice especificado (baseado em zero).

3.2.2.3 PILHA DE TABELAS DE SÍMBOLOS

A pilha de tabelas de símbolos está representada na classe *TPilhaTabelas*. A maneira como as tabelas estão organizadas nesta pilha indica o nível de aninhamento do escopo. As tabelas que estão mais ao topo possuem nível de aninhamento maior das que estão ao fundo da pilha. Também a pesquisa de símbolos acontece do topo para o fundo da pilha, para refletir a regra do aninhamento mais interno. A especificação da classe *TPilhaTabelas* está na fig. 20.

FIGURA 20 - ESPECIFICAÇÃO DA CLASSE *TPILHATABELAS*

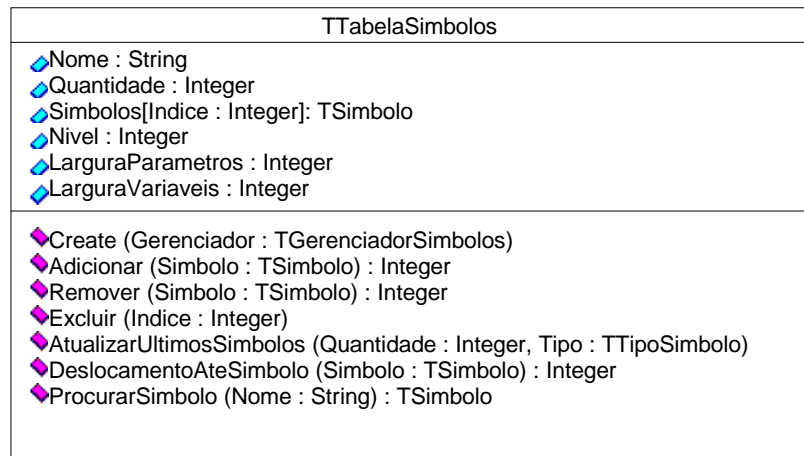


As principais operações da classe *TPilhaTabelas* são:

- Empilhar*: empilha a tabela de símbolos especificada;
- Desempilhar*: remove e retorna a tabela de símbolos ao topo da pilha;
- Topo*: retorna a tabela de símbolos ao topo da pilha. Não a remove da pilha.

3.2.2.4 TABELA DE SÍMBOLOS

A classe *TTabelaSimbolos* é responsável por armazenar os símbolos de maneira que os mesmos possam ser recuperados a qualquer momento durante a compilação. Ela também provê informações sobre o tamanho que os símbolos conterão na pilha e o nível do aninhamento do seu escopo. A fig. 21 mostra a especificação da classe.

FIGURA 21 - ESPECIFICAÇÃO DA CLASSE *TTabelaSimbolos*

Os principais atributos da classe *TTabelaSimbolos* são:

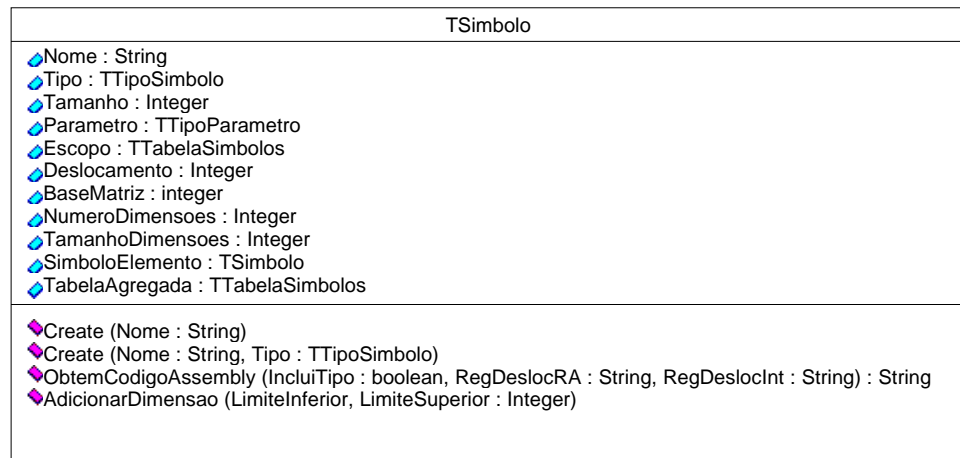
- LarguraParametros*: indica o tamanho necessário para os parâmetros no escopo representado por esta tabela de símbolos;
- LarguraVariaveis*: indica o tamanho necessário para as variáveis no escopo representado por esta tabela de símbolos;

As principais operações da classe *TTabelaSimbolos* são:

- DeslocamentoAteSimbolo*: retorna o deslocamento até o símbolo especificado. O deslocamento é a soma dos tamanhos dos símbolos anteriores ao símbolo especificado na lista. Os símbolos estão organizados pela ordem de declaração no programa-fonte;
- ProcurarSimbolo*: retorna o símbolo associado ao nome especificado.

3.2.2.5 REPRESENTAÇÃO DOS SÍMBOLOS

A classe responsável por armazenar as informações dos símbolos é a *TSimbolo*. Informações sobre nome, tipo, dimensões são providenciadas pelo compilador e armazenadas nesta classe. Já informações como tamanho, deslocamento e escopo são determinadas pela própria classe através do estado atual do escopo. O deslocamento de um símbolo *S*, por exemplo, pode ser determinado através da soma dos tamanhos dos símbolos declarados antes de *S*. Isto pode ser feito porque a ordem de instalação dos símbolos na tabela de símbolos obedece a ordem de declaração no programa fonte. A fig. 22 mostra a especificação desta classe.

FIGURA 22 - ESPECIFICAÇÃO DA CLASSE *TSIMBOLO*

Os principais atributos da classe *TSimbolo* são:

- a) *Escopo*: indica a tabela de símbolos que representa o escopo ao qual este o símbolo está associado;
- b) *Deslocamento*: indica o deslocamento do símbolo representado por esta classe dentro do escopo onde ele se encontra;
- c) *NumeroDimensoes*: quando este símbolo representa uma matriz, este atributo indica o número de dimensões desta matriz;
- d) *TamanhoDimensoes*: indica o número de elementos que são representados pelas dimensões deste símbolo;
- e) *SimboloElemento*: se este símbolo representa uma matriz, este atributo possui uma definição de símbolo que representa o elemento desta matriz;
- f) *TabelaAgregada*: se este símbolo representa um procedimento, este atributo guarda a tabela de símbolos que representa o escopo deste procedimento.

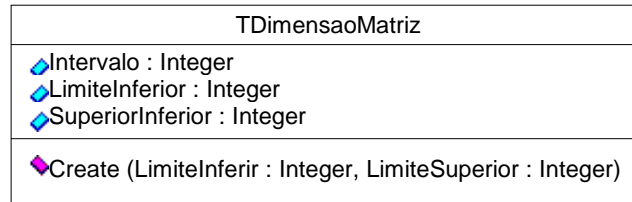
A operação *ObtemCodigoAssembly* retorna a representação deste símbolo em *Assembly*. Esta representação pode ser usada diretamente para gerar código *Assembly* que necessite acessar a localização de memória deste símbolo em tempo de execução.

3.2.2.6 INFORMAÇÕES SOBRE A DIMENSÃO DE MATRIZES

Esta informação é determinada pelo compilador e armazenada na classe *TSimbolo* através do seu método *AdicionarDimensao*. Internamente é criada uma classe *TDimensaoMatriz* que serve para armazenar este dado. O compilador não tem acesso direto

para criar instâncias desta classe. Isto somente poderá ser feito através da classe *TSimbolo*. A fig. 23 mostra a especificação desta classe.

FIGURA 23 - ESPECIFICAÇÃO DA CLASSE *TDIMENSÃO*



O atributo *Intervalo* da classe *TDimensaoMatriz* indica o intervalo da dimensão (limite superior – limite inferior + 1). Esta informação é usada para cálculos relacionados aos subscritos de matrizes.

3.3 MAPEAMENTO FINITO (ARRAYS) DINÂMICO NA LINGUAGEM FURBOL

O *array* dinâmico na linguagem FURBOL consiste em um *array* que pode ter seu tamanho alterado a qualquer momento durante a execução do programa. A sua área de armazenamento é alocada no *heap* e todo acesso e manutenção do *array* é realizado através de um descritor (também alocado dinamicamente) com campos que informam a situação atual do *array* na memória.

3.3.1 PASSOS NECESSÁRIOS PARA A ALOCAÇÃO DINÂMICA NO DOS

O sistema operacional DOS quando carrega programas na memória para serem executados, sempre aloca toda a memória disponível para o programa. Qualquer requisição de memória que se faça durante a execução do programa será negada pelo sistema operacional, pelo fato de não haver memória disponível.

Para tornar possível a alocação de memória, é necessário liberar a memória alocada pelo sistema operacional. Como arquivos com o formato *COM* gerados pelo *Turbo Assembler* (Borland, 1988) não podem ser maiores do que 64 Kbytes, pode-se liberar a memória acima de 64 Kbytes. O código para diminuir a área de memória alocada pelo DOS é mostrado no quadro 70 (Moon, 1999).

QUADRO 70 - CÓDIGO PARA DIMINUIR A MEMÓRIA ALOCADA PELO DOS

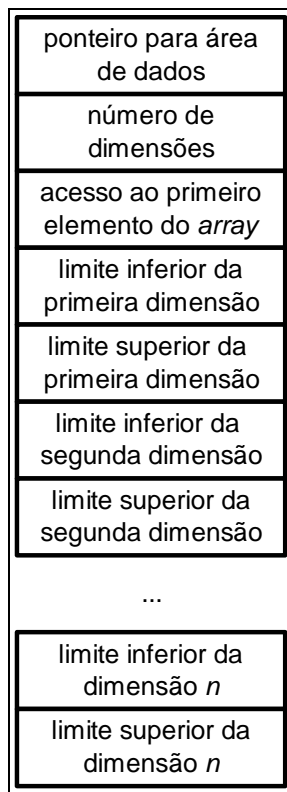
MOV	AX,CS	; Move o segmento de código
MOV	ES,AX	; para o registrador ES
MOV	BX,4096	; 4096 é o no. de parágrafos
MOV	AH,4AH	; 4Ah - realocação de memória.
INT	21H	; Chama a função do DOS para
		; alterar o tamanho da memó-
		; ria alocada para este prog.

Após a execução do código do quadro 70 torna-se possível a alocação dinâmica de memória no DOS. No ambiente FURBOL o código do quadro 70 é inserido automaticamente no início em todo o programa-alvo gerado.

3.3.2 DESCRITOR PARA ARRAY DINÂMICO

Um descritor armazena campos com informações necessárias sobre algum recurso. O *array* dinâmico implementado na linguagem FURBOL é uma referência para um descritor, alocado no *heap*, com os campos mostrados na fig. 24.

FIGURA 24 - DESCRITOR DE UM ARRAY DINÂMICO



Os campos que constituem o descritor mostrado na fig. 24 são:

- a) *ponteiro para a área de dados*: representa o ponteiro para a área de dados do *array* localizado na *heap*;
- b) *número de dimensões*: indica o número de dimensões atuais do *array*;
- c) *acesso ao primeiro elemento do array*: como os limites inferiores são definidos pelo usuário, é necessário que se tenha a diferença entre o endereço relativo zero (0) do *array* até o primeiro elemento definido pelo programador. A cada acesso ao *array* através dos seus subscritos, é realizado o cálculo do endereço do elemento e diminuído o número especificado por este campo;
- d) *limite inferior da dimensão n*: indica o limite inferior da *n*ésima dimensão do *array*;
- e) *limite superior da dimensão n*: indica o limite superior da *n*ésima dimensão do *array*.

Todo o *array* dinâmico na linguagem FURBOL é inicializado com um valor nulo (zero) no início da execução do escopo onde o mesmo está declarado. Um novo descritor é alocado dinamicamente toda vez que é executado o comando para redimensionamento do *array*. A referência deste novo descritor é então atribuída para a variável em questão.

A manipulação dos *arrays* dinâmicos é realizada através das funções *redimensiona*, *dimensoes*, *liminf* e *limsup*.

3.3.3 CONSTRUÇÕES PARA TRABALHAR COM ARRAYS DINÂMICOS

Para tornar possível a manipulação dos *arrays* dinâmicos na linguagem FURBOL, foram introduzidas novas construções. Estas construções incluem a declaração de *arrays* dinâmicos, o comando *redimensiona* e as funções *dimensoes*, *liminf* e *limsup*.

3.3.3.1 DECLARAÇÃO DE UM ARRAY DINÂMICO

A declaração de um *array* dinâmico é semelhante a declaração do *array* estático, exceto o número de dimensões que não é informada. A sintaxe é mostrada no quadro 71. Um exemplo de declaração de *array* dinâmico é mostrado no quadro 72.

QUADRO 71 - SINTAXE PARA A DECLARAÇÃO DE UM ARRAY DINÂMICO

```
<identificador>: matriz: <tipo>
```

QUADRO 72 - EXEMPLO DE DECLARAÇÃO DE ARRAY DINÂMICO

```
var
  M: matriz: inteiro;
```

3.3.3.2 COMANDO *REDIMENSIONA*

Esta função é responsável por determinar o tamanho, o número de dimensões e os limites inferior e superior de um *array* dinâmico. A sintaxe para este comando é mostrada no quadro 73.

QUADRO 73 - SINTAXE PARA O COMANDO *REDIMENSIONA*

```
redimensiona( <identificador>, <expressão>..<expressão> ... );
```

Este comando aloca um *array* dinâmico e atribui uma referência para *<identificador>*. Este identificador precisa ser declarado como *array* dinâmico. Cada *<expressão>* indica o limite inferior e superior da dimensão, respectivamente. Este último parâmetro pode ser repetido *n* vezes para refletir a quantidade de dimensões que devem ser alocadas. Por exemplo, o fragmento de programa FURBOL do quadro 74 aloca um *array* dinâmico de 10 linhas por 5 colunas, cujos limites inferior e superior são 2 e 11 para a primeira dimensão e 5 e 9 para a segunda dimensão, respectivamente.

QUADRO 74 - EXEMPLO DE REDIMENSIONAMENTO DE ARRAY DINÂMICO

```
var
  M: matriz: Inteiro;
inicio
  redimensiona(M, 2..11, 5..9);
fim;
```

Também é possível utilizar expressões do tipo inteiro para determinar as dimensões do *array*. O fragmento de código do quadro 75 aloca um *array* dinâmico com os mesmos limites e dimensões do quadro 74.

QUADRO 75 - EXEMPLO DE REDIMENSIONAMENTO DE ARRAY DINÂMICO ONDE OS LIMITES SÃO EXPRESSÕES

```

var
  x, y: Inteiro
  M: matriz: Inteiro;
inicio
  x := 2; y := 11;
  redimensiona(M, x..y, x + 3..y - 2);
fim;

```

3.3.3.3 FUNÇÃO *DIMENSOES*

Esta função retorna o número de dimensões atuais do *array* especificado. A sintaxe é mostrada no quadro 76.

QUADRO 76 - SINTAXE DA FUNÇÃO *DIMENSOES*

```

dimensoes( <identificador> );

```

O fragmento de código do quadro 77 atribui para a variável *x* o valor 2.

QUADRO 77 - EXEMPLO DE CÓDIGO COM A FUNÇÃO *DIMENSOES*

```

var
  x, y: Inteiro
  M: matriz: Inteiro;
inicio
  x := 2; y := 11;
  redimensiona(M, x..y, x + 3..y - 2);
  x := dimensoes(M);
fim;

```

3.3.3.4 COMANDO *LIMINF*

A função *LimInf* retorna o limite inferior de um *array* dinâmico em uma determinada dimensão. A sintaxe é mostrada no quadro 78.

QUADRO 78 - SINTAXE PARA A FUNÇÃO *LIMINF*

```

liminf( <identificador>, <expressão> )

```

No quadro 78, *identificador* precisa ser uma variável do tipo *array* dinâmico e *expressão* precisa ser do tipo *inteiro*. Esta expressão representa o número da dimensão (baseado em um) cujo limite inferior deve ser retornado. O exemplo do quadro 79 atribui à variável *x* o valor 5.

QUADRO 79 - EXEMPLO DE CÓDIGO COM A FUNÇÃO *LIMINF*

```

var
  x: Inteiro
  M: matriz: Inteiro;
inicio
  redimensiona(M, 2..11, 5..9);
  x := liminf(M, 2);
fim;

```

3.3.3.5 COMANDO *LIMSUP*

A função *LimSup* retorna o limite superior de um *array* dinâmico em uma determinada dimensão. A sintaxe é mostrada no quadro 80.

QUADRO 80 - SINTAXE DA FUNÇÃO *LIMSUP*

```

limsup( <identificador>, <expressão> )

```

No quadro 80, <identificador> precisa ser uma variável declarada como *array* dinâmico e <expressão> precisa ser do tipo *inteiro*. Esta expressão representa o número da dimensão (baseado em um) cujo limite superior deve ser retornado. O exemplo do quadro 81 atribui à variável *x* o valor 11.

QUADRO 81 - EXEMPLO DE CÓDIGO COM A FUNÇÃO *LIMSUP*

```

var
  x: Inteiro
  M: matriz: Inteiro;
inicio
  redimensiona(M, 2..11, 5..9);
  x := limsup(M, 1);
fim;

```

3.3.3.6 ACESSO AOS ELEMENTOS DO ARRAY DINÂMICO

O acesso aos elementos de um *array* dinâmico ocorre da mesma maneira do que o acesso aos elementos do *array* semi-estático. O quadro 82 mostra um exemplo de atribuição de valor para um *array* dinâmico.

QUADRO 82 - EXEMPLO DE ACESSO AOS ELEMENTOS DE UM ARRAY DINÂMICO

```

var
  M: matriz: Inteiro;
inicio
  ...
  M[2, 8] := 10;
fim;

```

3.3.4 CÓDIGO GERADO PELA IMPLEMENTAÇÃO DO ARRAY DINÂMICO

Nesta seção será apresentado o código-alvo gerado para cada construção relativa aos *arrays* dinâmicos.

Todo *array* dinâmico é implementado como uma referência para um descritor alocado na *heap*. Quando um *array* dinâmico é declarado, é gerado código-alvo para inicializar esta referência. Esta geração ocorre no início do código para o escopo onde o *array* dinâmico está declarado, antes que qualquer comando do programador seja executado. O quadro 72 mostra uma declaração de um *array* dinâmico *M* de inteiros. O código-alvo gerado para esta declaração é mostrado no quadro 83.

QUADRO 83 - CÓDIGO GERADO PARA A DECLARAÇÃO DO QUADRO 72

```

MOV      M,0           ; move o valor zero para a var. M

```

O programa do quadro 84 gera o código-alvo apresentado nos quadros 85, 86, 87 e 88. Várias tarefas são realizadas no momento da alocação do *array*. As explicações estão juntamente com o código gerado.

QUADRO 84 - PROGRAMA QUE USA O COMANDO *REDIMENSIONA*

```

programa TesteBasico;

var
  x, y: inteiro;
  m: matriz: inteiro;

inicio
  x := 6;
  y := 5;
  redimensiona(m, x - 5..y * 2, 1..10);
fim.

```

QUADRO 85 - CÓDIGO GERADO PARA O PROGRAMA DO QUADRO 84

```

; definição do segmento de código e de dados
CODIGO    SEGMENT
ASSUME    CS:CODIGO, DS:CODIGO
          ORG    100H

; ponto de entrada do programa
ENTRADA:  JMP    TESTEBASICO

; declaração de variáveis globais
X         DW    ?
Y         DW    ?
M         DW    ?    ; o array dinâmico é um ponteiro

; programa principal
TESTEBASICO  PROC NEAR

; tratamento da pilha para o programa principal
PUSH      BP
MOV       BP,SP

; liberação de memória alocada pelo DOS
MOV       AX,CS
MOV       ES,AX
MOV       BX,4096
MOV       AH,4AH
INT       21H

; x := 6
MOV       AX,6
MOV       X,AX

; y := 5
MOV       AX,5
MOV       Y,AX

; redimensiona(m, x - 5..y * 2, 1..10);
; Salva pilha para armazenar valores temporários
PUSH      BP
MOV       BP,SP

; calcula x - 5 e coloca na pilha
MOV       AX,X
SUB       AX,5
MOV       CX,AX
MOV       BX,CX
PUSH      BX

; calcula y * 2 e coloca na pilha
MOV       DI,2
MOV       AX,Y
MUL      DI
MOV       CX,AX
MOV       BX,CX
PUSH      BX

```

QUADRO 86 - CÓDIGO GERADO PARA O PROGRAMA DO QUADRO 84
(CONTINUAÇÃO DO QUADRO 85)

```

; empilha 1
MOV     CX,1
PUSH   CX

; empilha 10
MOV     BX,10
PUSH   BX

; aloca memória para o descritor do array
MOV     BX,1
MOV     AL,00
MOV     AH,48H
INT     21H

; salva o endereço do descritor na pilha
PUSH   AX

; inicializa os 3 primeiros campos do descritor
MOV     SI,AX
MOV     CX,0           ; zera a ref. da área de dados
MOV     [SI],CX
MOV     CX,2           ; inicializa no. de dimensões
MOV     [SI+2],CX
MOV     CX,0
MOV     [SI+4],CX     ; endereço relat. 1º elemento

; copia os valores dos limites das dimensões
; da pilha para o descritor
MOV     CX,4           ; no. de campos das dimensões (2*2)
MOV     DI,SI         ; endereço descritor
ADD     DI,6           ; pula campos iniciais
MOV     SI,BP         ; inicializa endereço da pilha
SUB     SI,2           ; posiciona no limite inferior
L2:
MOV     AX,[SI]       ; copia valor da pilha para AX
MOV     [DI],AX       ; copiar de AX para o descritor
ADD     DI,2           ; pula para próximo campo
SUB     SI,2           ; pula para próximo campo
DEC     CX             ; um campo a menos
CMP     CX,0           ; acabou?
JNE     L2            ; se não, próximo campo

; retorna o endereço do descritor salvo na pilha
POP     AX

; remove os valores das expressões avaliadas
MOV     SP,BP
POP     BP

```

QUADRO 87 - CÓDIGO GERADO PELO PROGRAMA DO QUADRO 84
(CONTINUAÇÃO DO QUADRO 86)

```

; move o ponteiro do descritor para a variável
MOV     M,AX

; calcula o número de elementos o array
MOV     DX,1           ; inicializa DX
MOV     CX,2           ; número de dimensões
MOV     SI,M           ; endereço do descritor
ADD     SI,6           ; pula campos iniciais
L1:
MOV     BX,[SI]        ; copia valor limite inferior
MOV     AX,[SI+2]      ; copia valor limite superior
ADD     SI,4           ; pula para próxima dimensão
SUB     AX,BX          ; limite superior - limite inferior
INC     AX             ; adiciona 1
MUL     DX             ; multiplica AX por DX
MOV     DX,AX          ; salve o produto em DX
DEC     CX             ; próxima dimensão
CMP     CX,0           ; acabou?
JNE     L1             ; se não, processar próxima dimensão

; calcula o tamanho de memória necessário para o array
MOV     AX,DX          ; move no. elementos para AX
MOV     BX,2           ; move tamanho do elemento
MUL     BX             ; multiplica pelo tamanho do elem.
MOV     BX,16          ; move tamanho do parágrafo
DIV     BX             ; calcula no. parágrafos
INC     AX             ; adiciona 1 ao no. parágrafos

; aloca a área de dados para o array
MOV     BX,AX          ; move tamanho para AX
MOV     AL,0
MOV     AH,48H
INT     21H

; move o endereço da área de dados do array para o
; descritor
MOV     SI,M
MOV     [SI],AX

```

QUADRO 88 - CÓDIGO GERADO PELO PROGRAMA DO QUADRO 84
(CONTINUAÇÃO DO QUADRO 87)

```

; calcula o endereço do primeiro elemento. Este valor é
; diminuído toda vez que se acessa um elemento do array

MOV     SI,M           ; move endereço do descritor
MOV     CX,[SI+2]     ; move no. dimensões
MOV     AX,0          ; inicializa AX
ADD     SI,6          ; pula campos iniciais do descritor
MOV     DX,[SI]       ; move valor limite inicial para DX
L3:
DEC     CX             ; diminui no. dimensões.
CMP     CX,0          ; acabou?
JE      L4            ; se sim, pule para L4
ADD     SI,4          ; pula para próxima dimensão
MOV     AX,[SI+2]     ; move limite superior para AX
MOV     BX,[SI]       ; move limite inferior para AX
SUB     AX,BX         ; limite superior - limite inferior
INC     AX            ; incrementa 1
MUL     DX            ; multiplica por DX
ADD     AX,[SI]       ; adiciona o limite inferior da
                        ; dimensão atual ao valor calculado
MOV     DX,AX         ; move o valor calculado para DX
JMP     L3            ; calcula a próxima dimensão
L4:
; rótulo para fim do cálculo
MOV     AX,DX         ; move o valor para AX
MOV     BX,2          ; move o tamanho do elemento para BX
MUL     BX            ; multiplica (AX contém o valor)

; move o valor calculado para o terceiro campo do
; descritor
MOV     SI,M
MOV     [SI+4],AX

; retorna a pilha e sai do programa
POP     BP
int     20h
TESTEBASICO ENDP
ULTIMO_ROTULO:
CODIGO ENDS
END     ENTRADA

```

O quadro 88 apresenta o código gerado para calcular o endereço relativo ao primeiro elemento do *array* de acordo com a fórmula apresentada no quadro 89. Esta fórmula é a parte final da fórmula genérica do quadro 31 para cálculo do endereço de um elemento em um *array* (2.5.5). O valor calculado por este código é armazenado no terceiro campo do descritor do *array*.

QUADRO 89 - FÓRMULA DO CÁLCULO DO PRIMEIRO ELEMENTO DO ARRAY DINÂMICO

$$((((\text{linf}_1 n_2 + \text{linf}_2) n_3 + \text{linf}_3) \dots) n_k + \text{linf}_k) \times w$$

Para a função *dimensoes*, utilizado no programa do quadro 90, o código gerado fica como o mostrado no quadro 91. Somente o código gerado pela função *dimensoes* é mostrado.

QUADRO 90 - PROGRAMA QUE UTILIZA A FUNÇÃO *DIMENSOES*

```

programa TesteBasico;

var
  x: inteiro;
  m: matriz: inteiro;

inicio
  x := dimensoes(m);
fim.

```

QUADRO 91 - CÓDIGO GERADO PARA A FUNÇÃO *DIMENSOES* DO QUADRO 90

```

MOV     SI,M           ; Obtém o descritor do array
MOV     BX,[SI+2]      ; Move o segundo campo para BX
MOV     AX,BX          ; Move este valor para a var. X
MOV     X,AX

```

Para a função *LimInf* utilizada no programa do quadro 92, o código gerado é aquele apresentado no quadro 93.

QUADRO 92 - PROGRAMA QUE UTILIZA A FUNÇÃO *LIMINF*

```

programa TesteBasico;

var
  x: inteiro;
  m: matriz: inteiro;

inicio
  x := liminf(m, 2);
fim.

```


QUADRO 93 - CÓDIGO GERADO PARA A FUNÇÃO *LIMINF*

MOV	AX,2	; move o valor da expressão para AX
DEC	AX	; decrementa 1
MOV	DI,2	; move o número de campos por
		; dimensão para DI
MUL	DI	; multiplica DI por AX
MUL	DI	; multiplica o tamanho dos campos
MOV	SI,M	; obtém o descritor
ADD	SI,6	; pula os 3 primeiros campos do descritor
ADD	SI,AX	; adiciona o deslocamento da
		; dimensão desejada
MOV	BX,[SI]	; move o valor para BX
MOV	AX,BX	; move o valor do limite inferior
MOV	X,AX	; para a variável X

A única diferença entre a função *LimInf* e *LimSup* em relação ao código gerado pelo programa do quadro 92 é que a instrução *MOV BX,[SI]* (quadro 93) do comando *LimInf* é substituído por *MOV BX,[SI+2]* no comando *LimSup*. Isto acontece porque a primeira instrução acessa o campo que tem o limite inferior no descritor do *array* e a segunda instrução, acessível através de *SI+2* no quadro 93, acessa o campo que tem o limite superior do *array*.

Quando um elemento é acessado, precisa-se calcular o seu endereço para que o valor possa ser atribuído e/ou obtido (seção 2.5.5). O quadro 94 mostra um programa FURBOL onde existe uma atribuição de um valor para um elemento de um *array* dinâmico. O código gerado para esta atribuição é apresentado no quadro 95.

QUADRO 94 - PROGRAMA FURBOL COM ATRIBUIÇÃO A UM ELEMENTO DE UM *ARRAY* DINÂMICO

```

programa AcessoElementos;

var
  M: matriz: Inteiro;
  i, j: inteiro;

inicio
  ...
  i := 2; j := 10;
  M[i,j] := 99;
fim.

```

QUADRO 95 - CÓDIGO GERADO PELA ATRIBUIÇÃO DO PROGRAMA DO QUADRO
94

```

MOV  SI,I           ; move o valor de I para SI
MOV  DI,M           ; obtém o descritor do array
ADD  DI,6+4         ; pula os campos iniciais e a 1º dim.
MOV  AX,[DI+2]      ; AX = limite inferior 2º dimensão
SUB  AX,[DI]        ; AX = AX - limite superior 2º dim.
INC  AX             ; soma 1
MOV  DI,AX
MOV  AX,DI
MOV  DI,SI          ; move o valor de I (SI) para DI
MUL  DI             ; mult. pelo intervalo da 2º dim.
MOV  DI,AX          ; move para DI
ADD  DI,J           ; adiciona o valor de J
MOV  DI,DI
MOV  AX,2           ; move o tamanho do elemento
MUL  DI             ; mult. no. de dimensões pelo tamanho do elemento
MOV  DI,AX          ; move o valor obtido para DI
MOV  SI,M           ; obtém o descritor
SUB  DI,[SI+4]      ; subtrai DI do endereço do 1º elem.
MOV  BX,DI          ; move o valor para BX
PUSH BX            ; e empilha
MOV  AX,99          ; move o valor 99 par AX
POP  DI             ; desempilha end. relat. do elem.
MOV  SI,M           ; obtém o descritor do array
MOV  SI,[SI]        ; obtém end. da área de dados array
ADD  SI,DI          ; adiciona deslocamento ao elemento
MOV  [SI],AX        ; move o valor para o elemento

```

A liberação do *array* dinâmico é feita implicitamente através de geração de código-alvo ao final de cada escopo onde existam declarações de *arrays* dinâmicos. Um procedimento que possua uma declaração de um *array* dinâmico *M* terá o código do quadro 96 ao final do escopo, antes que o procedimento retorne ao chamador.

QUADRO 96 - CÓDIGO GERADO PARA A LIBERAÇÃO DE UM ARRAY DINÂMICO *M*

MOV	DI,M	; obtém o descritor do array <i>M</i>
CMP	DI,0	; verifica se este está alocado
JE	L1	; se não está, não faz nada
MOV	AX,[DI]	; se sim, obtém o end. da área dados
MOV	ES,AX	; move o endereço para ES
MOV	AH,49h	; 49H = liberar memória
INT	21h	; chama função do DOS e libera área
MOV	AH,49h	
MOV	ES,DI	; move endereço do descritor para ES
INT	21h	; libera descritor
MOV	M,0	; limpa a referência da var. <i>M</i>
L1:		

3.4 ESPECIFICAÇÃO DA LINGUAGEM FURBOL

Nesta seção será apresentada a especificação da linguagem FURBOL. Ela consiste na especificação apresentada por Adriano (2001) com a inclusão de construções para acesso a variáveis não-locais e o uso de *arrays* dinâmicos.

O símbolo sustentado (#) na especificação representa um elemento léxico, ou seja, um símbolo terminal (como #ID). O mesmo acontece com as cadeias entre apostrofes ('). Estas cadeias representam símbolos terminais da linguagem. As palavras representam os elementos não terminais (como Matriz, MaisDimensão), os quais podem incluir derivações. O símbolo *ε* representa a palavra vazia.

Os atributos *Codigo* e *CodAsm* representam o código intermediário e o código *Assembly* gerados, respectivamente. O símbolo '||' é usado para concatenar cadeias para a geração deste código. O nome *vSimbolos* usado é um objeto responsável pelo controle dos escopo de nomes.

3.4.1 PROGRAMAS E BLOCOS

A definição para programas e blocos é mostrada no quadro 97. A ação semântica *GerarAsm* é responsável pela criação do programa *Assembly*. A variável *vSimbolos* é a instância do gerenciador de símbolos responsável pelo escopo de nomes da linguagem.

O comando *vSimbolos.Limpar* elimina todas as entradas do gerenciador de símbolos, permitindo que novos identificadores sejam declarados. A função *DeclaraDadosGlobais* gera

código para a declaração de todas as variáveis declaradas no programa principal. O comando *vSimbolos.AbrirEscopo* cria uma tabela de símbolos para o programa principal.

Se forem declarados *arrays* dinâmicos no programa principal, os comandos *CodIniArrays* e *CodLiberarArrays* geram código-alvo para a inicialização e liberação destes *arrays* dinâmicos, respectivamente.

QUADRO 97 - DEFINIÇÃO DE PROGRAMAS E BLOCOS

Programa	::=	'PROGRAMA'	<code>vSimbolos.Limpar; GeraAsm('CODIGO SEGMENT'); GeraAsm('ASSUME CS:CODIGO, DS:CODIGO'); GeraAsm('ORG 100H');</code>
		#ID	<code>ID.nome := #ID; GeraAsm('ENTRADA: JMP' #ID.nome);</code>
		';'	<code>vSimbolos.AbrirEscopo('Principal');</code>
		EstruturaDados	
		EstruturaSubRotinas	
		Ccomposto	<code>GeraAsm(EstruturaSubRotinas.CodAsm ID.nome 'PROC NEAR' 'PUSH BP' 'MOV BP,SP' 'MOV AX,CS' 'MOV ES,AX' 'MOV BX,4096' 'MOV AH,4AH' 'INT 21H' CodIniArrays(vSimbolos.EscopoAtual) CComposto.CodAsm CodLiberarArrays(vSimbolos.EscopoAtual) 'POP BP' 'INT 20H' ID.nome 'ENDP');</code>
'.'	<code>DeclararDadosGlobais; GerarAsm('ultimo_rotulo:'); GerarAsm('CODIGO ENDS'); GerarAsm('END ENTRADA');</code>		
CComposto	::=	'INICIO'	
		Comando	<code>CComposto.Codigo := Comando.Codigo; CComposto.CodAsm := Comando.CodAsm;</code>
		'FIM'	

3.4.2 ESTRUTURA DE DADOS

A definição das estruturas de dados é apresentada no quadro 98. As ações semânticas para a declaração de variáveis impedem que dois identificadores com mesmo nome sejam declarados no mesmo escopo. Esta verificação é realizada através do comando *vSimbolos.SimboloRedeclarado*. O comando *vSimbolos.Instalar* instala o símbolo especificado na tabela de símbolos. Este símbolo é criado através do comando *TSimbolo.Create*. Para atualizar vários identificadores com um mesmo tipo, o comando *vSimbolos.AtualizarUltimosSimbolos* deve ser usado.

No quadro 99 são apresentadas as produções para a declaração de *arrays*. Para a declaração de *arrays* dinâmicos, a produção *LimitesM* foi alterada. A alteração consiste em

tornar opcional a especificação dos limites do *array*. Quando incluídos na declaração, determinam o *array* como semi-estático e incluem os limites deste *array* na tabela de símbolos em tempo de compilação. Se forem omitidos, determinam que o *array* será dinâmico e que suas dimensões e limites somente serão conhecidos em tempo de execução.

QUADRO 98 - DEFINIÇÃO DE ESTRUTURAS DE DADOS

EstruturaDados	::=	'VAR'	
		#ID	Se vSimbolos.SimboloRedeclarado(ID) então Erro('Identificador redeclarado'); vSimbolos.Instalar(Tsimbolo.Create(#ID.nome));
		ListaID	
		(Matriz	
		ε)	vSimbolos.AtualizarUltimosSimbolos(NroVar, ListaId.Tipo);
		','	
Declaracoes	::=	#ID	Se vSimbolos.SimboloRedeclarado(ID) então Erro('Identificador redeclarado'); vSimbolos.Instalar(TSimbolo.Create(ID.nome));
		ListaID	
		(Matriz	
		ε)	vSimbolos.AtualizarUltimosSimbolos(NroVar, ListaId.Tipo);
		','	
		Declaracoes	
ListaID	::=	','	
		ID	Se vSimbolos.SimboloRedeclarado(ID) entao Erro('Símbolo redeclarado'); vSimbolos.Instalar(TSimbolo.Create(ID.nome));
		ListaID	
		(Matriz	
		ε)	vSimbolos.AtualizarUltimosSimbolos(NroVar, ListaId.Tipo); NroVar := 0;
		':'	
Tipo	::=	'INTEIRO'	Tipo.Tipo := tsInteiro;
		'LOGICO'	Tipo.Tipo := tsLogico;
		'MATRIZ'	Tipo.Tipo := tsMatriz;

QUADRO 99 - PRODUÇÕES PARA A DECLARAÇÃO DE ARRAYS

Matriz	::=	LimitesM	Simb:= vSimbolos.ProcurarSimbolo(Matriz.NomeId); se Simb.NumeroDimensoes > 0 entao Tipo := tsMatriz senao Tipo := tsMatrizDinamica; vSimbolos.AtualizarUltimosSimbolos(NroVar, Tipo.Tipo); NroVar := 0;
		':' Tipo	
LimitesM	::=	ε	
		'[' Dimensao ']'	
		ε	

Dimensao	::=	#NUM ₁	Simb:=vSimbolos.SimboloPorNome(Dimensao.NomeId) ;
		'..'	
		#NUM ₂	Simb.AdicionarDimensao(NUM ₁ .valor, NUM ₂ .valor) ;
		MaisDimensao	
MaisDimensao	::=	' , '	
		Dimensao	
			ε

3.4.3 ESTRUTURA DE SUBPROGRAMAS

A estrutura de subprogramas é apresentada no quadro 100. A cada procedimento encontrado o comando *vSimbolos.AbrirEscopo* é chamado para criar um novo escopo. Na entrada do procedimento é reservado espaço para as variáveis locais através das instruções em *Assembly PUSH BP, MOV BP,SP* e *SUB SP,n*; onde *n* é a largura das variáveis locais. Esta informação da largura é obtida através da função *vSimbolos.LarguraVariaveis*. Os parâmetros que estão antes das variáveis na pilha são removidos através da instrução *RET n*; onde *n* é a largura dos parâmetros, obtida através da função *vSimbolos.LarguraParametros*, adicionando-se dois bytes referentes ao tamanho do elo de acesso que está empilhado logo após os parâmetros.

QUADRO 100 - DEFINIÇÃO DE ESTRUTURAS DE SUBPROGRAMAS

EstruturaSubRotinas	::=	EstruturaProcedimento	EstruturaSubRotina.Codigo := EstruturaProcedimento.Codigo EstruturaSubRotina.CodAsm := EstruturaProcedimento.CodAsm;
		ε	
EstruturaProcedimento	::=	'PROCEDIMENTO'	
		#ID	Se vSimbolos.SimboloRedeclarado(ID.nome) entao Erro('Símbolo redeclarado'); Simb:=TSimbolo.Create(ProcName, tsProcedimento); vSimbolos.Instalar(Simb); Simb.TabelaAgregada := vSimbolos.AbrirEscopo(ID.nome);
		ParamFormais	
		','	
		EstruturaDados	
		EstruturaSubRotina	
		CComposto	EstruturaProcedimento.CodAsm := ID.nome 'PROC NEAR' 'PUSH BP' 'MOV BP,SP' 'SUB SP,' vSimbolos. EscopoAtual.LarguraVariaveis CComposto.CodAsm 'MOV SP,BP' 'POP BP' 'RET' vSimbolos. EscopoAtual.LarguraParametros + 2 ID.nome 'ENDP' EstruturaSubRotinas.CodAsm;
		','	vSimbolos.FecharEscopo;
		EstruturaSubRotinas	EstruturaProcedimento.Codigo := EstruturaProcedimento.Codigo EstruturaSubRotina.Codigo; EstruturaProcedimento.CodAsm := EstruturaProcedimento.CodAsm EstruturaSubRotina.CodAsm;

A definição de parâmetros é apresentada no quadro 101. Os símbolos são criados com a informação de parâmetro (valor ou referência). Logo depois são instalados na tabela de símbolos como acontece com as variáveis.

QUADRO 101 - DEFINIÇÃO DA ESTRUTURA DE PARÂMETROS FORMAIS

ParamFormais	::=	'('	
		(ParamValor	
		ParamRef)	
		SecaoParam	
)'	
			ϵ
SecaoParam	::=	';'	
		(ParamValor	
		ParamRef)	
		SecaoParam	
			ϵ
ParamValor	::=	#ID	<pre>Inc(NroVar); Se vSimbolos.SimboloRedeclarado(ID.nome) entao Erro('Símbolo redeclarado'); Simb:=TSimbolo.Create(ID.nome); Simb.Parametro := tpValor; vSimbolos.Instalar(Simb);</pre>
		ListaID	<pre>vSimbolos. AtualizarUltimosSimbolos(NroVar,ListaId.Tipo); NroVar := 0;</pre>
ParamRef	::=	'REF'	
		#ID	<pre>Inc(NroVar); Se vSimbolos.SimboloRedeclarado(ID.nome) entao Erro('Símbolo redeclarado'); Simb:=TSimbolo.Create(ID.nome); Simb.Parametro := tpReferencia vSimbolos.Instalar(Simb);</pre>
		ListaID	<pre>vSimbolos. AtualizarUltimosSimbolos(NroVar,ListaId.Tipo); NroVar := 0;</pre>

3.4.4 ESTRUTURA DE COMANDOS

A definição da estrutura de comandos é apresentada no quadro 102. Quando uma chamada de procedimento é encontrada, precisa-se empilhar o elo de acesso antes de realizar a chamada. O código para empilhar encontra-se na variável *CodigoEloAcesso*. Quando a chamada é para um procedimento aninhado em um nível menor que o chamador, deve-se determinar qual elo de acesso empilhar. Isto é realizado pela função *GerarEloAcesso*. Para a alocação de *arrays* dinâmicos foi incluído o comando *redimensiona*.

QUADRO 102 - DEFINIÇÃO DE ESTRUTURA DE COMANDOS

Comando	::=	#ID	se nao vSimbolos.ProcurarSimbolo(ID) entao Erro('Símbolo não declarado');
		(Atribuição	Comando.Codigo:=Atribuição.Codigo; Comando.CodAsm:=Atribuição.CodAsm;
	ChamProc)	NivelChamador := vSimbolos.EscopoAtual.Nivel; NivelChamado := ID.Nivel; se NivelChamador<NivelChamado entao CodigoEloAcesso := 'PUSH BP' senao CodigoEloAcesso := GerarEloAcesso(NivelChamador - NivelChamado + 1) 'PUSH DI'; Comando.Codigo := Comando.Codigo ChamProc.Codigo CodigoEloAcesso 'CHAMADA' ChamProc.Nome; Comando.CodAsm := Comando.CodAsm ChamProc.CodAsm CodigoEloAcesso 'CALL' ChamProc.Nome;	
	Virgula		
		CCondicional	
	Virgula	Comando.Codigo := CCondicional.Codigo; Comando.CodAsm := CCondicional.CodAsm;	
		CRepetição	
	Virgula	Comando.Codigo := CRepetição.Codigo; Comando.CodAsm := CRepetição.CodAsm;	
		CEntrada	
	Virgula	Comando.Codigo := CEntrada.Codigo; Comando.CodAsm := CEntrada.CodAsm;	
		CSaida	
	Virgula	Comando.Codigo := CSaida.Codigo; Comando.CodAsm := Comando.CodAsm;	
		CInc	Comando.Codigo := CInc.Codigo; Comando.CodAsm := CInc.CodAsm;
	Virgula		
		CDec	Comando.Codigo := CDec.Codigo; Comando.CodAsm := CDec.CodAsm;
	Virgula		
		'NL'	
	Virgula	Comando.Codigo := CNL.Codigo; Comando.CodAsm := Comando.CodAsm;	
		CRedimensiona	Comando.Codigo := Credimensiona.Codigo; Comando.CodAsm := Credimensiona.CodAsm;
	Virgula		
	Virgula		
Virgula	::=	' ; '	
		Comando	Virgula.Codigo:=Comando.Codigo Virgula.CodAsm:=Comando.CodAsm
		ε	

A definição da estrutura de atribuição é apresentada no quadro 103. Quando a atribuição é feita para um elemento de um *array*, deve-se primeiro obter a área de dados onde o valor deve ser armazenado. Quando esta situação é reconhecida, é gerado código para obter a área de dados do *array* em questão antes que se atribuir o valor ao elemento.

QUADRO 103 - DEFINIÇÃO DA ESTRUTURA DE ATRIBUIÇÃO

Atribuicao	::=	' := '	<pre> se Atribuição.Deslocamento <> '' então inicio RX:=novo_t; IndCodAsm := 'MOV RX,' Ldesloca ' PUSH RX'; PreIdAt := 'POP DI'; end; </pre>
		Expressao	<pre> Atribuicao.Local := Expressao.local; se Expressao.tipo <> Atribuicao.tipo entao Erro; se Expressão.Deslocamento <> '' então Atribuição.Codigo := idAT '[' Expressao.Deslocamento ']:= ' Expressao.Local; senão Atribuição.Codigo := IdAT ' := ' Expressao.Local); se Simb.Tipo = tsMatrizDinamica entao inicio PreIdAt := PreIdAt 'MOV SI,[' IdAt ']' 'MOV SI,[SI] 'ADD SI,DI'; IdAt := '[SI]'; fim; Atribuição.CodAsm := IndCodAsm Expressao.CodAsm AtPrelocal 'MOV RX,' aTlocal PreIdAT 'MOV' IdAT ',' RX; </pre>

No quadro 104 é apresentada a estrutura de chamada de procedimentos.

QUADRO 104 - DEFINIÇÃO DA ESTRUTURA DE CHAMADA DE PROCEDIMENTOS

ChamProc	::=	' ('	
		ParamAtuais	ChamProc.CodAsm := ParamAtuais.CodAsm;
)'	
		ε	
ParamAtuais	::=	#ID	<pre> SimbVar := vSimbolos.SimboloPorNome(lexema); SimbPar := SimbProc.Simbolos[parN - 1] se SimbVar.Tipo <> SimbPar.Tipo entao Erro('Tipos incompatíveis'); se SimbVar.Parametro = tpReferencia entao ParamAtuais.CodAsm := 'LEA DI,' ID.nome ' PUSH DI' ParamAtuais.CodAsm; se SimbVar.Parametro = tpValor entao ParamAtuais.CodAsm := 'PUSH' ID.nomeAsm ParamAtuais.CodAsm; </pre>
		(',' ParamAtuais	
		ε)	

A estrutura de comando de repetição é mostrada no quadro 105.

QUADRO 105 - DEFINIÇÃO DA ESTRUTURA DE COMANDO DE REPETIÇÃO

CRepetição	::=	'ENQUANTO'	CRepetição.Inicio := novo_l; Expressao.v := novo_l; Expressao.f := CRepetição.Proximo;
		Expressao	se Expressao.Tipo <> tsLogico entao Erro('Tipos incompatíveis');
		'FACA'	
		CComposto	CComposto.Proximo := CRepetição.inicio; CRepetição.Codigo := CRepetição.Inicio ':' Expressão.Codigo E.v ':' CComposto.Codigo GOTO CRepetição.Inicio; CRepetição.CodAsm := CRepetição.Inicio ':' Expressão.CodAsm E.v ':' CComposto.CodAsm 'JMP' CRepetição.inicio;

No quadro 106 é apresenta a estrutura de comandos condicionais.

QUADRO 106 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS CONDICIONAIS

Ccondicional	::=	'SE' Expressao 'ENTAO'	
		CComposto	se Expressao.Tipo <> tsLogico entao Erro('Tipo incompatível'); CComposto.Proximo:=CCondicional.Proximo; CCondicional.Codigo:=Expressão.Codigo E.v ':' CComposto.Codigo; CCondicional.CodAsm:=Expressão.CodAsm E.v ':' CComposto.CodAsm;
		CCondicional2	CCondicional.codigo:=CCondicional.Codigo CCondicional2.Codigo; CCondicional.CodAsm:=CCondicional.CodAsm CCondicional2.CodAsm;
CCondicional2	::=	'SENÃO'	
		CComposto	CCondicional2.Codigo := Expressao.Codigo 'goto' proximo CCondicional2.f ':' CComposto.Codigo; CCondicional2.CodAsm := Expressao.CodAsm 'JMP' proximo CCondicional2.f ':' CComposto.CodAsm;
		ε	CCondicional2.Codigo := Ccondicional2.Codigo f ':' ; Ccondicional2.CodAsm := Ccondicional2.Codasm f ':' ;

O quadro 107 apresenta a estrutura para comandos de entrada. O quadro 108 apresenta a estrutura para comandos de saída.

QUADRO 107 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS DE ENTRADA

CEntrada	::=	'LEITURA'	
		'('	
		#ID	se nao vSimbolos.ProcurarSimbolo(ID.nome) entao Erro('Identificador não declarado');
		LeituraListaId	
)'	CEntrada.Codigo := 'LEITURA(ID1' LeituraListaID.Codigo)';
LeituraListaID	::=	' , '	
		#ID	se nao vSimbolos.ProcurarSimbolo(ID.nome) entao Erro('Identificador não declarado'); LeituraListaID.Codigo:= LeituraListaID.Codigo ' , ' ID;
		LeituraListaId	
			ε

QUADRO 108 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS DE SAÍDA

Csaida	::=	'IMPRIME'	
		'('	
		Expressao	
		ListaExpressao	
)'	CSaida.Codigo := 'Imprime' Expressao.Codigo ' , ' ListaExpressao.Codigo;
ListaExpressao	::=	' , '	
		Expressao	ListaExpressao.Codigo := ListaExpressao.Codigo Listaexpressao.Codigo;
		ListaExpressao	
			ε

A estrutura de incremento e decremento de variáveis é apresentada no quadro 109.

QUADRO 109 - DEFINIÇÃO DA ESTRUTURA DE INCREMENTO E DECREMENTO

CInc	::=	'INC'	
		'('	
		#ID	se vSimbolos.ProcurarSimbolo(ID.nome) entao Erro('Símbolo não declarado'); se ID.Tipo <> 'inteiro' entao Erro('Tipos incompatíveis'); CInc.Codigo := 'INC(' ID.nome ')'; CInc.CodAsm := 'INC' ID.nome;
)'	
CDec	::=	'DEC'	
		'('	
		#ID	se vSimbolos.ProcurarSimbolo(ID.nome) entao Erro('Símbolo não declarado'); se ID.Tipo <> 'inteiro' entao Erro('Tipos incompatíveis'); CDec.Codigo := 'DEC(' ID.nome ')'; CDec.CodAsm := 'DEC' ID.nome;
)'	

A estrutura de comandos para *array* dinâmico é apresentado no quadro 110. O código gerado pelo comando para redimensionamento do *array* dinâmico é muito extenso. Por este

motivo este foi substituído pelo comando *CodigoAsmMatriz*. Maiores detalhes sobre o código gerado para a alocação de *arrays* dinâmicos podem ser vista na seção 3.3.4.

Quando as dimensões são especificadas no comando redimensiona (quadro 110), as expressões que determinam os limites inferior e superior são avaliadas e colocadas na pilha para serem processadas posteriormente na determinação do tamanho do *array* dinâmico a ser alocado.

QUADRO 110 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS PARA *ARRAYS* DINÂMICOS

CRedimensiona	::=	'REDIMENSIONA'	
		' ('	
		#ID	<pre> Simb:=vSimbolos. ProcurarSimbolo(#ID.nome); se nao Simb <> nil entao Erro('Identificador não declarado'); se Simb.Tipo<>tsMatrizDinamica entao erro('Identificador não é uma matriz dinâmica');</pre>
		','	
		DimensaoDinamica	<pre> RotuloDimensoes:=novo_l; RotuloInicializa:=novo_l; RotuloComecoCalcula:=novo_l; RotuloFimCalcula:=novo_l; se (Simb.Escopo.Nivel>0) and (Simb.Escopo.Nivel< vSimbolos.EscopoAtual. Nivel) então inicio AcessoVariavelInicio := 'PUSH DI' GerarEloAcesso(vSimbolos. EscopoAtual.Nivel-Simb. Escopo.Nivel); AcessoVariavelFim := 'POP DI'; RegistradorVariavel:='DI'; fim; senao inicio AcessoVariavelInicio:=''; AcessoVariavelFim:=''; RegistradorVariavel := 'BP'; fim; CodAsm := CodigoAsmMatriz;</pre>
)'	

QUADRO 111 - DEFINIÇÃO DA ESTRUTURA DE COMANDOS PARA ARRAYS DINÂMICOS (CONTINUAÇÃO DO QUADRO 110)

DimensaoDinamica	::=	Expressao	<pre> se Expressao.Tipo<>tsInteiro entao Erro('Tipos incompatíveis'); R := novo_t; DimensaoDinamica.CodAsm:= Expressao.CodAsm PreLocal 'MOV' R ',' Local 'PUSH' R; </pre>
		'..'	
		Expressao	<pre> se Expressao.Tipo<>tsInteiro entao Erro('Tipos incompatíveis'); R := novo_t; DimensaoDinamica.CodAsm:= Expressao.CodAsm PreLocal 'MOV' R ',' Local 'PUSH' R; </pre>
MaisDimensaoDinamica		Dimensao.CodAsm:=Dimensao.CodAsm + MaisDimensaoDinamica.CodAsm;	
MaisDimensaoDinamica	::=	','	
		DimensaoDinamica	
		ε	

3.4.5 ESTRUTURA DE EXPRESSÕES

As estruturas de expressões são apresentadas nos quadros 112, 113, 114, 115, 116, 117 e 118. O quadro 117 apresenta o acréscimo de produções para a implementação das funções *dimensoes*, *liminf* e *limsup*. Os quadros 119 e 120 mostram as produções e as ações semânticas para estas funções.

QUADRO 112 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES

Expressão	::=	Expressão2	<pre> Relação.v:=Expressão2.v; Relação.f:=Expressão2.f; Relação.local:=Expressão2.local; Relação.codigo:=Expressão2.codigo; Relação.codAsm:=Expressão2.codAsm; </pre>
		Relação	<pre> Expressão.local :=Relação.local; Expressão.Codigo:=Relação.Codigo; Expressão.CodAsm:=Relação.CodAsm; Expressão.v:=Relação.v Expressão.f:=Relação.f </pre>

QUADRO 113 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 112)

Relação	::=	'='	
		Expressao2	Relacao.Codigo := 'SE' Relação.local '=' Expressão2.Local 'goto' E.v 'goto' E.f; Relacao.CodAsm := 'MOV R0,' Relacao.local 'CMP R0,' local 'JE' RV 'JMP Rf';
		'<>'	
		Expressao2	Relacao.Codigo:= 'SE' Relação.local '<>' Expressão2.local 'goto' E.v 'goto' E.f; Relação.CodAsm := 'MOV R0,' Relacao.local 'CMP R0,' local 'JNE' Rv 'JMP Rf';
		'<'	
Expressao2	Relacao.Codigo := 'SE' Relação.local '<' Expressão2.local 'goto' E.v 'goto' E.f; Relacao.CodAsm := 'MOV R0,' Relacao.local 'CMP R0,' local 'JB' Rv 'JMP Rf';		
'>'			
Expressao2	Relacao.Codigo := 'SE' Relação.local '>' Expressão2.local 'goto' E.v 'goto' E.f; Relacao.CodAsm := 'MOV R0,' Relacao.local 'CMP R0,' local 'JA' Rv 'JMP' Rf;		
		ε	
Expressão2	::=	TC,	EL _i .v:=TC.v EL _i .f:=TC.f EL _i .Local := TC.local EL _i .Codigo := TC.Codigo EL _i .CodAsm := TC.CodAsm
		EL;	Expressao2.local := EL.local Expressao2.Codigo := EL.Codigo Expressao2.CodAsm := EL.CodAsm

QUADRO 114 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 113)

EL	::=	'+'	
		TC	<code>EL₁.Local := novo_t;</code> <code>EL₁.Codigo := EL.Codigo TC.Codigo EL₁.local :=' EL.local '+'</code> <code> TC.local;</code> <code>EL₁.CodAsm := EL.Codigo TC.Codigo 'MOV RX,' EL.local </code> <code>'ADD RX,' Tc.Local 'MOV' EL₁.local ',RX';</code>
	EL ₁	<code>EL.local := EL₁.local;</code> <code>EL.Codigo := EL₁.Codigo;</code> <code>EL.CodAsm := EL₁.CodAsm;</code>	
	ε		
TC	::=	'-'	
		TC	<code>EL₁.local:= novo_t;</code> <code>EL₁.Codigo:= EL.Codigo TC.Codigo EL₁.local :=' </code> <code>EL.local '-' TC.local;</code> <code>EL₁.CodAsm := EL.Codigo TC.Codigo 'MOV 'RX,' EL.local</code> <code> 'SUB RX,' Tc.Local 'MOV' EL₁.local ',RX';</code>
	EL ₁	<code>EL.local := EL₁.local;</code> <code>EL.Codigo := EL₁.Codigo;</code> <code>EL.CodAsm := EL₁.CodAsm;</code>	
	ε		
TC	::=	F	<code>TL.v := F.v; TL.f := F.f;</code> <code>TL.local := F.local; TL.Codigo := F.Codigo; TL.CodAsm := F.CodAsm;</code>
		TL	<code>T.v := TL_s.v;</code> <code>T.f := TL_s.f; TC.local := TL.local;</code> <code>TC.CodAsm := TL.CodAsm; TC.Codigo := TL.Codigo;</code>

QUADRO 115 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 114)

TL	::=	'*'		
		F	<code>TL₁.local := novo_t;</code> <code>TL₁.Codigo := TL.Codigo F.Codigo TL₁.local :=' TL.local</code> <code>'*' F.local;</code> <code>TL₁.CodAsm := TL.Codigo F.Codigo 'MOV AX,' TLLOCAL </code> <code>'MUL ' Local 'MOV' TliLocal ',AX;</code>	
	TL ₁	<code>TL.local := TL_{1s}.local; TL.Codigo := TL_{1s}.Codigo;</code> <code>TL.CodAsm := TL_{1s}.CodAsm;</code>		
		::=	'/'	
			F	<code>TL₁.local := novo_t; TL₁.Codigo:=TL.Codigo F.Codigo </code> <code>'TL₁.local :=' TL.local '/' F.local;</code> <code>TL₁.CodAsm := TL.Codigo F.Codigo 'MOV AX,' TLLOCAL </code> <code>'DIV' Local 'MOV' TliLocal ',AX;</code>
	TL ₁	<code>TL.local := TL₁.local; TL.Codigo := TL_s.Codigo;</code> <code>TL.CodAsm := TL₁.CodAsm;</code>		
		::=	'E'	
			F	<code>TL.v := novo_l; TL.f := TL₁.f; F.v := TL₁.v; F.f :=TL₁.f;</code> <code>TL₁.Codigo := TL.Codigo TL₁.v ':' F.Codigo;</code> <code>TL₁.CodAsm := TL.CodAsm TL₁.v ':' F.CodAsm;</code>
			TL ₁	<code>TL.v := TL₁.v; TL.f := TL₁.f; TL.CodAsm := TL_{1s}.CodAsm;</code> <code>TL.Codigo := TL_{1s}.Codigo;</code>
	ε			

QUADRO 116 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 115)

L	:=	Lista_E]	<pre>TipoInt:= Integer(vSimbolos.SimboloPorNome(Lista_Earray).SimboloElemento.Tipo); L.local := novo_t; L.Deslocamento := novo_t; L.Codigo := gerar(L.local := c(Lista_Earray); L.Codigo := gerar(L.deslocamento ':' Lista_E.local '*' Largura(Lista_E.array)); L.CodAsm:=Lista_E.CodAsm 'MOV DX,' Lista_Elocal 'MOV AL,' IntToStr(TtipoInt) 'MUL DX' 'MOV ' Ldesloca ',AX'; se Simb.Tipo = tsMatriz então CodigoAsm := CodigoAsm 'ADD' LDesloca ',' IntToStr(Lc) //LC negativo senao // Matriz Dinâmica inicio se (Simb.Escopo.Nivel>0) and (Simb.Escopo.Nivel< vSimbolos.EscopoAtual.Nivel) então CodigoAsm := CodigoAsm 'PUSH DI' GerarEloAcesso(vSimbolos.EscopoAtual.Nivel - Simb.Escopo.Nivel) 'MOV SI,' Simb.ObtemCodigoAssembly(True,'DI','') 'POP DI' senao CodigoAsm := CodigoAsm 'MOV SI,' Simb.ObtemCodigoAssembly(True,'BP',''); CodigoAsm := CodigoAsm 'SUB' Ldesloca ',[SI+4]'; // O valor está positivo; Fim;</pre>
Lista_E	::=	#ID	L.local := ID.local; L.deslocamento := '';
		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 117 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 116)

F	::=	'('	Expressao.v := F.v; Expressao.f := F.f
		Expressao	
)'	F.local := Expressao.local; F.Codigo := Expressao.Codigo; F.CodAsm := Expressao.CodAsm;
		'_'	
		Expressao	F.local := novo_t; F.Codigo := Expressao.Codigo gerar(F.local ':' 'uminus' E.local);
		'NAO'	Expressao.v := F.f; Expressao.f := F.v;
		Expressao	
		L	<pre> Simb := vSimbolos.ProcurarSimbolo(Evar); se Simb <> nil and (Simb.Tipo=tsMatrizDinamica) then inicio F.Local := NOVO_t; se (Simb.Escopo.Nivel>0) and (Simb.Escopo.Nivel< vSimbolos.EscopoAtual.Nivel) entao F.CodAsm := F.CodAsm 'PUSH DI' GerarEloAcesso(vSimbolos.EscopoAtual.Nivel - Simb.Escopo.Nivel) 'MOV SI,' Simb.ObtemCodigoAssembly(True,'DI',LDesloca) 'POP DI; senao F.CodAsm := F.CodAsm 'MOV SI,' Simb.ObtemCodigoAssembly(True,'BP',LDesloca); F.CodAsm := F.CodAsm 'MOV SI,[SI]' 'ADD SI,' LDesloca 'MOV' F.Local ',[SI]'; Fim; senao se (Simb <> nil) entao se L.deslocamento := 0 então F.local := L.local senão F.local := novo_t; gerar(F.local ':' L.local '[' L.deslocamento '); </pre>
		NUM	F.local := NUM; F.Codigo := ''; F.CodAsm := '';
		FDimensoes	F.Local := FDimensoes.Local; F.Codigo := FDimensoes.Codigo; F.CodAsm := FDimensoes.CodAsm;
		FLimInf	F.Local := FLimInf.Local; F.Codigo := FLimInf.Codigo; F.CodAsm := FLimInf.CodAsm;
		FLimSup	F.Local := FLimSup.Local; F.Codigo := FLimSup.Codigo; F.CodAsm := FLimSup.CodAsm;
		ε	

QUADRO 118 - DEFINIÇÃO DA ESTRUTURA DE EXPRESSÕES (CONTINUAÇÃO DO QUADRO 117)

R	::=	Expressao	<pre> t:='DI'; m:=ndim+1; Simb := vSimbolos.SimboloPorNome(L_EArray); se Simb.Tipo = tsMatriz entao ValorDimensao := IntToStr(limite(L_Earray,m)); senao inicio // Matriz Dinâmica se (Simb.Escopo.Nivel>0) and (Simb.Escopo.Nível< vSimbolos.EscopoAtual.Nivel) entao R.CodAsm := R.CodAsm GerarEloAcesso(vSimbolos.EscopoAtual.Nivel - Simb.Escopo.Nivel) 'MOV DI,' Simb.ObtemCodigoAssembly(True,'DI',''); senao R.CodAsm := R.CodAsm 'MOV DI,' Simb.ObtemCodigoAssembly(True,'BP',''); R.CodAsm := R.CodAsm 'ADD DI,6+ IntToStr((m - 1) * 4) 'MOV AX,[DI+2]' 'SUB AX,[DI]' 'INC AX' 'MOV DI,AX'; ValorDimensao := 'DI'; fim; gerar(t + ':=' + local + '*' + ValorDimensao); R.CodAsm := R.CodAsm Expressao.CodAsm 'MOV AX,' ValorDimensao 'MOV DI,SI' 'MUL DI' 'MOV' t ',AX' 'ADD' t ',' Expressao.local; R.local:=t; gerar(t ':=' t '+' Expressao.local); </pre>
		R ₁	<pre> R_s.matriz := R₁.matriz; R_s.local := R₁.local; R_s.ndim := R₁.ndim; R_s.CodAsm := R₁.CodAsm; </pre>
		ε	

QUADRO 119 - DEFINIÇÃO DAS ESTRUTURAS PARA AS FUNÇÕES DE ARRAYS DINÂMICOS

FDimensoes	::=	'DIMENSOES'	
		'('	
		#ID	<pre> Simb := vSimbolos.ProcurarSimbolo(Nome); se nao Simb <> nil então Erro('Identificador não declarado'); Se Simb.Tipo <> tsMatrizDinamica então Erro('Identificador não é uma matriz dinâmica'); </pre>
)'	<pre> FDimensoes.Local := novo_t; se (Simb.Escopo.Nivel > 0) and (Simb.Escopo.Nivel < vSimbolos.EscopoAtual.Nivel) então FDimensoes.CodAsm := 'PUSH DI' GerarEloAcesso(vSimbolos.EscopoAtual.Nivel - Simb.Escopo.Nivel) 'MOV SI,' Simb.ObtemCodigoAssembly(True, 'DI', '') 'POP DI' senao FDimensoes.CodAsm := 'MOV SI,' Simb.ObtemCodigoAssembly(True, 'BP', ''); FDimensoes.CodAsm := FDimensoes.CodAsm 'MOV' Local ',',[SI+2]'; </pre>
FLimInf	::=	'('	
		#ID	<pre> Simb := vSimbolos.ProcurarSimbolo(Nome); se nao Simb <> nil então Erro('Identificador não declarado'); se Simb.Tipo <> tsMatrizDinamica então Erro('Identificador não é uma matriz dinâmica'); </pre>
		','	
		Expressao	<pre> se Expressao.Tipo <> tsInteiro então Erro('Tipo incompatível'); Local := novo_t; FLimInf.CodAsm := FLimInf.CodAsm Expressao.CodAsm Expressao.PreLocal 'MOV AX,' Expressao.Local 'DEC AX' 'MOV DI,2' 'MUL DI' 'MUL DI'; se (Simb.Escopo.Nivel > 0) and (Simb.Escopo.Nivel < vSimbolos.EscopoAtual.Nivel) então FLimInf.CodAsm := FLimInf.CodAsm GerarEloAcesso(vSimbolos.EscopoAtual.Nivel - Simb.Escopo.Nivel) 'MOV SI,' Simb.ObtemCodigoAssembly(True, 'DI', ''); senao FLimInf.CodAsm := FLimInf.CodAsm 'MOV SI,' Simb.ObtemCodigoAssembly(True, 'BP', ''); FLimInf.CodAsm := FLimInf.CodAsm + 'ADD SI,6'#10 'ADD SI,AX'#10 'MOV' Local ',',[SI]'; </pre>
)'	

QUADRO 120 - DEFINIÇÃO DAS ESTRUTURAS PARA AS FUNÇÕES DE ARRAYS DINÂMICOS (CONTINUAÇÃO DO QUADRO 119)

FLimSup	::=	' ('	
		#ID	<pre> Simb := vSimbolos.ProcurarSimbolo(Nome); se nao Simb <> nil entao Erro('Identificador não declarado'); se Simb.Tipo <> tsMatrizDinamica entao Erro('Identificador não é uma matriz dinâmica'); </pre>
		','	
	Expressao	<pre> se Expressao.Tipo <> tsInteiro então Erro('Tipo incompatível'); Local := novo_t; FLimSup.CodAsm := FLimSup.CodAsm Expressao.CodAsm Expressao.PreLocal 'MOV AX,' Expressao.Local 'DEC AX' 'MOV DI,2' 'MUL DI' 'MUL DI'; se (Simb.Escopo.Nivel > 0) and (Simb.Escopo.Nivel < vSimbolos.EscopoAtual.Nivel) entao FLimSup.CodAsm := FLimSup.CodAsm GerarEloAcesso(vSimbolos.EscopoAtual.Nivel - Simb.Escopo.Nivel) 'MOV SI,' Simb.ObtemCodigoAssembly(True, 'DI', ''); senao FLimSup.CodAsm := FLimSup.CodAsm 'MOV SI,' Simb.ObtemCodigoAssembly(True, 'BP', ''); FLimSup.CodAsm := FLimSup.CodAsm + 'ADD SI,6'#10 'ADD SI,AX'#10 'MOV' Local ',[SI+2]'; </pre>	
)'			

3.5 APRESENTAÇÃO DO PROTÓTIPO

O protótipo deste trabalho, referenciado como ambiente FURBOL, é um ambiente capaz de compilar, gerar código executável e disparar a execução de programas escritos na linguagem FURBOL, especificada na seção 3.4. O ambiente FURBOL herdou características de ambientes já implementados por Silva (1993), Bruxel (1996), Radloff (1997), Schimt (1999) e André (2000).

Após a extensão realizada por este trabalho, as características gerais do ambiente continuaram as mesmas apresentadas por André (2000) como a tela de edição e funções gerais do ambiente disponíveis através dos menus da tela de edição. Foram realizados alguns melhoramentos na formatação do código *Assembly* apresentado após o processo de compilação.

A principal alteração deste trabalho foi a inclusão de construções para a implementação de *arrays* dinâmicos na linguagem FURBOL e o acesso a variáveis não-locais.

3.5.1 PRINCIPAIS CARACTERÍSTICAS DO AMBIENTE APÓS A EXTENSÃO

Este trabalho estendeu a definição formal da linguagem FURBOL através da implementação de *arrays* dinâmicos e acesso a variáveis não-locais. O quadro 121 apresenta um programa exemplo onde o novo recurso de *arrays* dinâmicos é utilizado.

QUADRO 121 - EXEMPLO DE PROGRAMA FURBOL QUE UTILIZA ARRAY DINÂMICO

```

programa Exemplo;

var
  m: Matriz: Inteiro;
  i, j: Inteiro;

inicio
  Redimensiona(M, 1..10, 1..10);
  i := LimInf(M, 1);
  enquanto i < LimSup(M, 1) faca
  inicio
    j := LimInf(M, 2);
    enquanto j < LimSup(M, 2) faca
    inicio
      M[i,j] := i * j;
      Inc(j);
    fim;
    M[i,j] := dimensoes(M);
    Inc(i);
  fim;
  M[i,j] := dimensoes(M);
fim.

```

O programa do quadro 121 atribui aos elementos o valor do índice da linha multiplicado pelo índice da coluna onde o elemento está localizado. Nos últimos elementos de cada linha e coluna são atribuídos o número de dimensões do *array* dinâmico *M*. Neste exemplo são apresentados todos os recursos implementados com relação ao *array* dinâmico: a declaração, o comando *redimensiona* e as funções *dimensoes*, *liminf* e *limsup*.

Neste trabalho também foi revisto a implementação de escopo de Adriano (2001). A partir deste trabalho é possível o acesso a variáveis não-locais declaradas em procedimentos aninhados. O quadro 122 apresenta um exemplo de programa FURBOL que utiliza este recurso.

QUADRO 122 - EXEMPLO DE PROGRAMA FURBOL QUE ACESSA VARIÁVEIS NÃO-LOCAIS

```

programa Exemplo;

var
  Valor: Inteiro;

  procedimento Calcula(ref Numero: Inteiro);
  var
    Temporario: Inteiro;

    procedimento Soma;
    inicio
      Temporario := Temporario + 12;
    fim;

    procedimento Multiplica;

      procedimento Multiplica3;
      inicio
        Temporario := Temporario * 3;
      fim;

      inicio
        Temporario := Temporario * 2;
        Multiplica3;
      fim;

    inicio
      Temporario := Numero;
      Soma;
      Multiplica;
      Numero := Temporario;
    fim;

  inicio
    Valor := 10;
    Calcula(Valor);
  fim.

```

No programa do quadro 122 a variável *temporario* é local ao procedimento *calcula* e é não-local aos procedimentos *soma*, *multiplica* e *multiplica3*. Estes três últimos alteram o valor desta variável no escopo do procedimento *calcula*. Este assunto é apresentado na seção 2.6 deste trabalho.

3.5.2 INTERFACE DO PROTÓTIPO

Características gerais como a edição de programas-fonte (fig. 25) e funções como *compilar*, *gerar código* e *executar* não foram alteradas. Foram somente realizados alguns

melhoramentos na formatação do código *Assembly* (fig. 26) apresentado após o processo de compilação.

FIGURA 25 - TELA DE EDIÇÃO DE PROGRAMAS-FONTE DO AMBIENTE FURBOL

The screenshot shows a window titled 'FURBOL - C:\Documents and Settings\stephan\Desktop\FURBOL\FURBOL\Atual\Sam...'. The menu bar includes 'Arquivo', 'Editar', 'Projeto', and 'Ajuda'. The toolbar contains icons for file operations and execution. The 'Código Fonte' tab is active, displaying the following code:

```

programa Exemplo;

var
  Valor: Inteiro;

procedimento Calcula(ref Numero: Inteiro);
var
  Temporario: Inteiro;

procedimento Soma;
inicio
  Temporario := Temporario + 12;
fim;

procedimento Multiplica;

```

The status bar at the bottom indicates 'Lin:36' and 'Col:7'.

FIGURA 26 - TELA DE APRESENTAÇÃO DO CÓDIGO *ASSEMBLY* GERADO

The screenshot shows a window titled 'FURBOL - D:\TCC\FURBOL\Atual\Samples_Exemplo_Escopo.fur'. The menu bar includes 'Arquivo', 'Editar', 'Projeto', and 'Ajuda'. The toolbar contains icons for file operations and execution. The 'Código Assembly' tab is active, displaying the following assembly code:

```

POP    BP
RET    4
CALCULA ENDP

SOMA   PROC NEAR
PUSH   BP
MOV    BP,SP
MOV    DI,[BP+4]
MOV    AX,WORD PTR [DI-2]
MOV    DI,[BP+4]
ADD    AX,12
MOV    BX,AX
MOV    AX,BX
MOV    DI,[BP+4]
MOV    WORD PTR [DI-2],AX

```

The status bar at the bottom indicates 'Lin:32' and 'Col:2'.

4 CONCLUSÕES

Os objetivos deste trabalho foram atingidos. A revisão da definição de escopo da linguagem, o qual torna possível o acesso a variáveis não-locais, e a implementação de *arrays* dinâmicos, que oferecem uma grande flexibilidade na construção de programas, foram implementadas.

A implementação do acesso a variáveis não-locais foi baseada em Aho (1995) e em programas compilados e depurados no ambiente Borland Delphi 5, verificando-se o código objeto gerado pelo mesmo. A base teórica para a implementação de *arrays* dinâmicos foi baseada em Aho (1995). Estas construções foram especificadas utilizando a notação BNF e gramática de atributos, as quais mostraram-se muito eficientes para a definição de uma linguagem de programação como o FURBOL.

A maior dificuldade foi a adaptação da especificação existente ao novo controle de escopo implementado. Uma estrutura a parte foi implementada com a finalidade de facilitar este controle e todas as produções foram alteradas para utilizar esta estrutura. O controle de dimensões de *arrays* semi-estáticos, implementado por Adriano (2001), também foi adaptado a esta nova estrutura.

Os *arrays* dinâmicos podem ser utilizados em expressões numéricas e atribuições. As funções construídas para a obtenção de informações acerca dos *arrays* podem ser utilizadas em expressões, atribuições, comandos de repetição e comandos condicionais. Na alocação de *arrays* dinâmicos, onde são determinados os limites das dimensões, podem ser usadas variáveis e expressões que são avaliadas antes da execução do comando de alocação, permitindo assim maior flexibilidade na construção de programas.

A limitação sobre os *arrays* dinâmicos está no fato de que não é possível atribuir um *array* dinâmico a outro diretamente. Esta cópia somente pode ser realizada copiando-se o valor de cada elemento de um *array* para outro. Também não é possível a atribuição de um *array* dinâmico para um *array* semi-estático e vice-versa. No acesso aos elementos de *arrays* dinâmicos, os subscritos não são verificados em tempo de execução quanto ao acesso indevido fora dos limites especificados, ou seja, pode-se acessar elementos que não foram alocados para o *array* dinâmico. Esta verificação poderia ser realizada comparando-se os

valores dos subscritos com os valores dos limites inferior e superior da dimensão, contidos no descritor do *array* dinâmico.

4.1 EXTENSÕES

A implementação de unidades do tipo função (*function* em Pascal), comandos para entrada e saída de dados, comando de repetição *Para* (*for* em Pascal), comando de seleção *Caso* (*case* em Pascal), ampliação das funções para a manipulação de *arrays* semi-estáticos e dinâmicos, geração de código executável no formato *EXE* e otimização do código gerado pelo compilador são sugestões para trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

ADRIANO, Anderson. **Implementação de mapeamento finito (*arrays*) no ambiente FURBOL**. 2001. 94 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

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

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

BORLAND INTERNATIONAL INC. **Turbo Assembler** : reference guide, version 1.0. Scotts Valley: Borland International, 1988.

BORLAND INTERNATIONAL INC. **Object Pascal Reference**. Scotts Valley: Borland International, 2001.

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

CANTU, Marco. **Dominando o Delphi 5: a bíblia**. Tradução João Eduardo Nobrega Tortello. São Paulo: Makron Books, 2000.

FURLAN, Jose Davi. **Modelagem de objetos através da UML - the unified modeling language**. São Paulo: Makron Books do Brasil, 1998.

GHEZZI, Carlo, JAZAYERI, Mehdi. **Conceitos de linguagens de programação**. 2. ed. Rio de Janeiro: Campus, 1991.

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

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

MOON, Raymond. **Common reason why memory allocation fails**, Vienna, 1999. Disponível em <<http://www.faqs.org/faqs/assembly-language/x86/general/part3/section-2.html>>. Acesso em: 13 jun. 2002

QUADROS, Daniel G. A. **PC ASSEMBLER usando DOS**. Rio de Janeiro: Campus, 1988. 174p.

RADLOFF, Marcelo. **Protótipo de um ambiente para programação em uma linguagem bloco estruturada com vocabulário na língua portuguesa**. 1997. 73 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SCHIMT, Hélió. **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, Joilson Marcos da. **Desenvolvimento de um ambiente de programação para a linguagem Portugol**. 1993. 82 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

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

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**. 1993. 80 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

YEUNG, Bik-Cheung. **8086/8088 assembly language programming**. Chichester: John Wiley, 1984. 265p.