

UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA

Departamento de Ciências Exatas

Ambiente de Apoio ao Ensino de
Linguagem de Programação I

João Paulo Vieira Bahia



Vitória da Conquista - BA

Ambiente de Apoio ao Ensino de Linguagem de Programação I

João Paulo Vieira Bahia

Orientadora: Cátia Khouri

Monografia de conclusão de curso apresentada ao Departamento de Ciências Exatas – DCE-UESB - para obtenção do título de Bacharel em Ciência da Computação.

Área de Concentração: Linguagem de Programação e Compiladores.

UESB – Vitória da Conquista
Fevereiro de 2008

**Dedico o presente trabalho a Deus por sempre
está ao meu lado me ajudando.**

**A minha família que me incentivou, me apoiou
e me deu a oportunidade dessa conquista.**

AGRADECIMENTOS

À Diogo Souza, colega de curso e amigo, que me ajudou neste passando um pouco do seu conhecimento, contribuindo com o desenvolvimento desse trabalho.

À minha orientadora Cátia Khouri que muito contribuiu no desenvolvimento desse trabalho.

Aos meus amigos e colegas de curso Adriana Almeida, Diogo Souza, Francisco Fonseca, Igor Xavier, Karina Moreira, Pablo Matos, Rodrigo Nery, Sandoelton Santana, Vagner Dias e aos demais colegas que me ajudaram durante o curso e por estarem em todos os momentos.

RESUMO

Aprender a programar é um processo difícil e exigente para maioria dos alunos. Entre as dificuldades estão a complexidade que as linguagens de programação usuais apresentam, possuindo sintaxes complexas, e o elevado nível de abstração envolvido. Com o intuito de diminuir ou até sanar essas dificuldades, o Ambiente Virtual de Linguagem de Programação (AVLP) foi desenvolvido.

O AVLP simula o que ocorre dentro do computador quando ele está executando um programa escrito na linguagem de programação C++. Ele auxilia o estudante não somente na compreensão dessa linguagem, como também pode ampliar esse conhecimento para demais, visto que a lógica de programação não difere muito de uma linguagem para outra. Sua interface é simples e de fácil compreensão, assim o estudante terá uma melhor visão e maior facilidade em assimilar a linguagem.

ABSTRACT

Learning to program is a difficult and challenging process for most students. Among the difficulties are the complexity of the usual programming languages presents, with complex syntax, and the high level of abstraction involved. In order to reduce or even solve these difficulties, the Ambiente Virtual de Linguagem de Programação (AVLP) was developed.

The AVLP simulates what happens inside the computer when it is running a program written in the C++ programming language. It helps the student not only in understanding that language, but also can expand that knowledge to others, because the logic of programming does not differ greatly from one language to another. Its interface is simple and easy to understand, so the student will have a better view and it will be easier to assimilate the language.

LISTA DE FIGURAS

Figura 1: Execução de um programa fonte.....	14
Figura 2: Estrutura de um compilador.....	16
Figura 3: Um diagrama de estados para reconhecer nomes	17
Figura 4: Análise Preditiva Tabular.....	20
Figura 5: Ilustração de procedimentos em C++	26
Figura 6: Ambiente típico C++	28
Figura 7: Programa que soma dois números em C++	29
Figura 8: Representando em fluxograma a estrutura de seleção if.....	35
Figura 9: Representando em fluxograma a estrutura de seleção if/else	37
Figura 10: Representando em fluxograma a estrutura de repetição while ...	38
Figura 11: O uso do break	39
Figura 12: Um array com 11 elementos	40
Figura 13: Exemplo de função	42
Figura 14: Código executado	43
Figura 15 - Código da Torre de Hanói no AMBAP.....	45
Figura 16: Execução do código da Torre de Hanói.....	46
Figura 17: Execução do código da Torre de Hanói passo a passo.....	46
Figura 18: Autômato Finito do analisador léxico	48
Figura 19: Trecho do código da classe AnaliseLexica.....	49
Figura 20: Trecho do código da classe AnaliseSintatica.....	50
Figura 21: Trecho do código da classe AnalisadorSemantico	51
Figura 22: Exibição de erros no AVL P	52
Figura 23: AVL P executando um código em C++	53
Figura 24: Criação de variável	54
Figura 25: Valor da variável modificado	54
Figura 26: Array sendo criado no AVL P	55
Figura 27: Chamada de função com parâmetros	55
Figura 28: Console do AVL P	56
Figura 29: Entrada de dados através do teclado	56

LISTA DE TABELAS

Tabela 1-Símbolos e lexemas.....	17
Tabela 2-Tabela de análise preditiva	22
Tabela 3-Movimentos de um reconhecedor preditivo analisando uma sentença.....	23
Tabela 4-Tabela de análise preditiva com correção de erros	24
Tabela 5- Tipos de variáveis em C++	31
Tabela 6- Precedência de operadores	34

Sumário

1. INTRODUÇÃO	11
1.1. MOTIVAÇÃO.....	11
1.2. OBJETIVO DO TRABALHO	12
1.3. ORGANIZAÇÃO DA MONOGRAFIA	12
2. FERRAMENTAS EXISTENTES	14
2.1. TRADUTORES DE LINGUAGEM DE PROGRAMAÇÃO.....	14
2.1.1. ESTRUTURAS DE UM TRADUTOR.....	15
2.1.2. ANÁLISE LÉXICA	15
2.1.2.1 TOKENS	18
2.1.3. TABELA DE SIMBOLOS.....	18
2.1.4. ANÁLISE SINTÁTICA	18
2.1.4.1. ANÁLISE PREDITIVA TABULAR	19
2.1.4.1.1. DEFINIÇÃO DE FIRST(α).....	20
2.1.4.1.2. DEFINIÇÃO DO FOLLOW(α)	21
2.1.4.1.3. ALGORITMO PARA CONSTRUÇÃO UMA TABELA DE ANÁLISE PREDITIVA.....	21
2.1.4.2. RECUPERAÇÃO DE ERRO.....	22
2.1.5. ANÁLISE SEMÂNTICA.....	24
2.2. PARADIGMAS DE PROGRAMAÇÃO	24
2.2.1. PROGRAMAÇÃO ORIENTADA A PROCEDIMENTOS	25
2.2.2. PROGRAMAÇÃO ORIENTADA A OBJETOS.....	25
2.3. C++.....	26
2.3.1. FUNDAMENTOS DE UM AMBIENTE TÍPICO C++	27
2.3.2. PROGRAMAÇÃO EM C++	27
2.3.2.1. VARIÁVEIS.....	31
2.3.2.2. O QUALIFICADOR CONST.....	32
2.3.2.3. TIPOS DE OPERADORES.....	32
2.3.2.3.1. OPERADOR DE ATRIBUIÇÃO.....	32
2.3.2.3.2. OPERADORES ARITMÉTICOS	32
2.3.2.3.3. OPERADOR DE EXTRAÇÃO.....	33
2.3.2.3.4. OPERADORES RELACIONAIS	33
2.3.2.3.5. OPERADORES LÓGICOS.....	33
2.3.2.3.6. PRECEDÊNCIA DE OPERADORES	34

2.3.2.4. ESTRUTURAS DE CONTROLE.....	34
2.3.2.4.1. ESTRUTURAS DE SELEÇÃO IF	34
2.3.2.4.2. ESTRUTURAS DE SELEÇÃO IF/ELSE.....	36
2.3.2.5. ESTRUTURA DE SELEÇÃO WHILE	36
2.3.2.6. O COMANDO BREAK	38
2.3.2.7. ARRAYS.....	39
2.3.2.7.1. DECLARAÇÃO DE ARRAYS	39
2.3.2.8. FUNÇÕES	40
2.3.2.8.1. TIPOS DE FUNÇÃO	41
2.3.2.8.2. O COMANDO RETURN.....	41
2.3.2.8.3. PARÂMETROS DA FUNÇÃO	41
2.3.2.8.4. PASSAGEM POR VALOR	42
2.3.2.8.5. EXEMPLO DE UMA FUNÇÃO EM C++	42
2.3.2.9. ESCOPO DE VARIÁVEIS	43
2.3.2.9.1. VARIÁVEIS LOCAIS	43
2.3.2.9.2. VARIÁVEIS GLOBAIS	43
2.3.2.9.3. PARÂMETROS FORMAIS	43
2.4. AMBIENTE DE SIMULAÇÃO DE MÁQUINAS VIRTUAIS (ASIMAV)	44
2.5. AMBIENTE DE APRENDIZADO DE PROGRAMAÇÃO (AMBAP)	45
3. AMBIENTE VIRTUAL DE LINGUAGEM DE PROGRAMAÇÃO (AVLP)....	47
3.1. A ANÁLISE LÉXICA DO AVLP	48
3.2. A ANÁLISE SINTÁTICA DO AVLP.....	49
3.3. A ANÁLISE SEMÂNTICA DO AVLP	50
3.4. TRATAMENTO DE ERROS NO AVLP	51
3.5. A EXECUÇÃO DO AVLP	52
4. CONCLUSÕES.....	57
5. REFERÊNCIAS	58
APÊNDICE A - GRAMÁTICA RESTRINGIDA DO C++.....	59
APÊNDICE B – CÁLCULO DO FIRST E DO FOLLOW	61
APÊNDICE C – TABELA DE ERROS	68
APÊNDICE D – DIAGRAMA DE ATIVIDADE	70
APÊNDICE E – DIAGRAMA SEQUÊNCIA.....	71
APÊNDICE F – DIAGRAMA DE CLASSES.....	72

APÊNDICE A - Gramática Restringida do C++

S-> #include <iostream>S'A

S'-> using namespace std; | λ

A-> void id (A') D | int id(A') D | float id(A') D | string id(A') D | char id (A') D | bool id (A') D | double id (A') D

A'-> B | λ

B-> int id C B' | float id C B' | string id C B' | char id C B' | bool id C B' | double id C B'

B'-> B | λ

B2-> ch | strB2'

B2'-> +strB2' | λ

C-> []C | λ

D-> {FE}D'

D'-> A | λ

E -> return E2; | λ

E2-> B2 | H

F-> GF'

F'-> F | λ

G-> I; | id O; | Q; | R; | U | W | X; | break;

H-> -HH' | (H)H' | idOH' | numH'

H'-> +HH' | -HH' | *HH' | /HH' | %HH' | λ

I -> int J | float J | string J | char J | double J | bool J

J-> id K

K-> , J | L | =E2 | λ

L-> [L'] M | λ

L'-> num | id | λ

M-> L | = {N}

N-> str N2 | num N3 | chN4

N2-> ,strN2 | λ

N3-> ,numN3 | λ

N4> ,chN4 | λ

O-> (P) | = E2 | [L'] = E2 | λ

P-> idP' | numP' | strP' | chP' | trueP' | falseP' | λ

P'-> ,P | λ

Q-> cin >> id

R-> cout << T

T->idT' | str T' | num T' | chT'

T'-> << T | λ

U-> if(Z){F}V

V-> else{F}| λ

W-> while(Z){F}

X-> const Y| static Y

Y-> int id=num | float id=num | double id=num | string id=str | char id=ch | bool id=Y'

Y'-> true | false

Z-> ch Z2 | str Z2 | Z3 | (Z)Z2

Z2-> A2 Z | λ

Z3-> - Z3Z4 | ! Z3 | idZ4 | numZ4

Z4-> +Z3Z4 | -Z3Z4 | * Z3Z4 | /Z3Z4 | %Z3Z4 | A2Z | λ

A2-> < | > | <= | >= | == | || | &&

APÊNDICE B – Cálculo do First e do Follow

S-> #include <iostream>S'A

First(#include <iostream> A) = {#} **M**[S,#]= S-> #include <iostream>S' A

S'-> using namespace std;| λ

First(using namespace std;) = {using} **M**[S',using] = S'-> using namespace std;

Follow(S') = First(A)={void, int, float, string, char, bool, double}

M[S',void]=**M**[S', int]=**M**[S', float]=**M**[S',string]=**M**[S',char]=**M**[S', bool] =

M[S',double]= S'-> λ

A-> void id (A') D | int id(A') D | float id(A') D | string id(A') D | char id (A') D | bool

id (A') D | double id (A') D

First(void id (A') D) = {void}

M[A,void] = A->void id (A') D

First(int id (A') D) = {int}

M[A,int] = A->int id (A') D

First(float id (A') D) = {float}

M[A,float] = A->float id (A') D

First(string id (A') D) = {string}

M[A,string] = A->string id (A') D

First(char id (A') D) = {char}

M[A,char] = A->char id (A') D

First(bool id (A') D) = {bool}

M[A,bool] = A->bool id (A') D

First(double id (A') D) = {double}

M[A,double] = A->double id (A') D

A'-> B| λ

First(A') = First(B) = {int, float, string, char, bool, double} **M**[A,int] = **M**[A,float] =

M[A,string] = **M**[A,char] = **M**[A,bool] = **M**[A,double] = A'->B

Follow(A')={ } **M**[A',)]= A'-> λ

B->int id C B' | float id C B' | string id C B' | char id C B' | bool id C B' |double id C B'

First (int id C B') = {int}

M[B,int] = B->int id C B'

First (float id C B') = {float}

M[B,float] = B->float id C B'

First (string id C B') = {string}

M[B,string] = B->string id C B'

First (char id C B') = {char}

M[B,char] = B->char id C B'

First (bool id C B') = {bool}

M[B,bool] = B->bool id C B'

First (double id C B') = {double}

M[B,double] = B->double id C B'

B' -> B | λ

First(B) = { , }

M[B', ,] = B' -> B

Follow(B') = Follow(B) = Follow(A') = { } } **M[B',)] = B' -> λ**

B2 -> ch | strB2'

First(ch) = { ch }

M[B2, ch] = B2 -> ch

First(strB2') = { str }

M[B2, str] = B2 -> strB2'

B2' -> +strB2' | λ

First(+strB2') = { + }

M[B2', +] = B2' -> +strB2'

Follow(B2') = Follow(B2) = Follow(K) = Follow(J) = Follow(I) = { ; }

M[B2', ;] = B2' -> λ

C -> []C | λ

First([]C) = { [] }

M[C, []] = C -> []C

Follow(C) = First(B') + Follow(B) = { , ,) }

M[C, ,] = M[C,)] = C -> λ

D -> {FE}D'

First({FE}) = { { }

M[D, {] = D -> {FE}D'

D' -> A | λ

First(A) = { void, int, float, string, char, bool e double } **M[D', void] = M[D', int] = M[D',**

float] = M[D', string] = M[D', char] = M[D', bool] = [D', double] = D' -> A

Follow(D') = Follow(D) = Follow(A) = Follow(S) = { \$ } **M[D', \$] = D' -> λ**

E -> return E2; | λ

First(return E2;) = { return }

M[E, return] = E -> return E2;

Follow(E) = { }

M[E,] = E -> λ

E2 -> B2 | H

First(B2) = { ch, str }

M[E2, ch] = M[E2, str] = E2 -> B2

First(H) = { -, (, id, num }

M[E2, -] = M[E2, (] = M[E2, id] = M[E2, num] = E2 -> H

H -> -HH' | (H)H' | idOH' | numH'

First(-HH') = { - }

M[H, -] = H -> -HH'

$\text{First}((H)H') = \{ (\}$ $\mathbf{M[H, (] = H \rightarrow (H)H'}$
 $\text{First}(\text{id}OH') = \{ \text{id} \}$ $\mathbf{M[H, \text{id}] = H \rightarrow \text{id}OH'}$
 $\text{First}(\text{num}H') = \{ \text{num} \}$ $\mathbf{M[H, \text{num}] = H \rightarrow \text{num}H'}$

$\mathbf{H' \rightarrow +HH' \mid -HH' \mid *HH' \mid /HH' \mid \%HH' \mid \lambda}$

$\text{First}(+HH') = \{ + \}$ $\mathbf{M[H', +] = H' \rightarrow +HH'}$
 $\text{First}(-HH') = \{ - \}$ $\mathbf{M[H', -] = H' \rightarrow -HH'}$
 $\text{First}(*HH') = \{ * \}$ $\mathbf{M[H', *] = H' \rightarrow *HH'}$
 $\text{First}(/HH') = \{ / \}$ $\mathbf{M[H', /] = H' \rightarrow /HH'}$
 $\text{First}(\%HH') = \{ \% \}$ $\mathbf{M[H', \%] = H' \rightarrow \%HH'}$

$\text{Follow}(H') = \text{Follow}(H) = \text{Follow}(E2) = \{ ; ,) \}$
 $\mathbf{M[H', ;] = M[H',)] = H' \rightarrow \lambda}$

$\mathbf{I \rightarrow \text{int } J \mid \text{float } J \mid \text{string } J \mid \text{char } J \mid \text{double } J \mid \text{bool } J}$

$\text{First}(\text{int } J) = \{ \text{int} \}$ $\mathbf{M[I, \text{int}] = I \rightarrow \text{int } J}$
 $\text{First}(\text{float } J) = \{ \text{float} \}$ $\mathbf{M[I, \text{float}] = I \rightarrow \text{float } J}$
 $\text{First}(\text{string } J) = \{ \text{string} \}$ $\mathbf{M[I, \text{string}] = I \rightarrow \text{string } J}$
 $\text{First}(\text{char } J) = \{ \text{char} \}$ $\mathbf{M[I, \text{char}] = I \rightarrow \text{char } J}$
 $\text{First}(\text{double } J) = \{ \text{double} \}$ $\mathbf{M[I, \text{double}] = I \rightarrow \text{double } J}$
 $\text{First}(\text{bool } J) = \{ \text{bool} \}$ $\mathbf{M[I, \text{bool}] = I \rightarrow \text{bool } J}$

$\mathbf{J \rightarrow \text{id } K}$

$\text{First}(\text{id } K) = \{ \text{id} \}$ $\mathbf{M[J, \text{id}] = J \rightarrow \text{id } K}$

$\mathbf{K \rightarrow , J \mid L \mid =B2 \mid \lambda}$

$\text{First}(, J) = \{ , \}$ $\mathbf{M[K, ,] = K \rightarrow , J}$
 $\text{First}(L) = \{ [\}$ $\mathbf{M[K, [] = K \rightarrow L}$
 $\text{First}(= K') = \{ = \}$ $\mathbf{M[K, =] = K \rightarrow = B2}$

$\text{Follow}(K) = \{ ; \}$ $\mathbf{M[K, ;] = K \rightarrow \lambda}$

$\mathbf{L \rightarrow [L'] M \mid \lambda}$

$\text{First}([L'] M) = \{ [\}$ $\mathbf{M[L, [] = L \rightarrow [L'] M}$

$\text{Follow}(L) = \text{Follow}(K) = \text{Follow}(J) = \text{Follow}(I) = \{ ; \}$

$\mathbf{M[L, ;] = L \rightarrow \lambda}$

$\mathbf{L' \rightarrow \text{num} \mid \text{id} \mid \lambda}$

$\text{First}(\text{num}) = \{ \text{num} \}$ $\mathbf{M[L', \text{num}] = L' \rightarrow \text{num}}$

$\text{First}(\text{id}) = \{ \text{id} \}$ $\text{M}[\text{L}', \text{id}] = \text{L}' \rightarrow \text{id}$

$\text{Follow}(\text{L}') = \{ \lambda \}$ $\text{M}[\text{L}', \lambda] = \text{L}' \rightarrow \lambda$

$\text{M} \rightarrow \text{L} \mid \{ \text{N} \}$

$\text{First}(\text{L}) = \{ \lambda \}$ $\text{M}[\text{M}, \lambda] = \text{M} \rightarrow \text{L}$

$\text{First}(\{ \text{N} \}) = \{ \text{N} \}$ $\text{M}[\text{M}, \text{N}] = \text{M} \rightarrow \{ \text{N} \}$

$\text{N} \rightarrow \text{str N2} \mid \text{num N3} \mid \text{chN4}$

$\text{First}(\text{str N2}) = \{ \text{str} \}$ $\text{M}[\text{N}, \text{str}] = \text{N} \rightarrow \text{strN2}$

$\text{First}(\text{num N3}) = \{ \text{num} \}$ $\text{M}[\text{N}, \text{num}] = \text{N} \rightarrow \text{numN3}$

$\text{First}(\text{ch N4}) = \{ \text{ch} \}$ $\text{M}[\text{N}, \text{ch}] = \text{N} \rightarrow \text{chN4}$

$\text{N2} \rightarrow \text{,strN2} \mid \lambda$

$\text{First}(\text{,strN2}) = \{ \text{,} \}$ $\text{M}[\text{N2}, \text{,}] = \text{N2} \rightarrow \text{,strN2}$

$\text{Follow}(\text{N2}) = \text{Follow}(\text{N}) = \{ \lambda \}$ $\text{M}[\text{N2}, \lambda] = \text{N2} \rightarrow \lambda$

$\text{N3} \rightarrow \text{,numN3} \mid \lambda$

$\text{First}(\text{,numN3}) = \{ \text{,} \}$ $\text{M}[\text{N3}, \text{,}] = \text{N3} \rightarrow \text{,numN3}$

$\text{Follow}(\text{N3}) = \text{Follow}(\text{N}) = \{ \lambda \}$ $\text{M}[\text{N3}, \lambda] = \text{N3} \rightarrow \lambda$

$\text{N4} \rightarrow \text{,chN4} \mid \lambda$

$\text{First}(\text{,chN4}) = \{ \text{,} \}$ $\text{M}[\text{N4}, \text{,}] = \text{N4} \rightarrow \text{,chN4}$

$\text{Follow}(\text{N4}) = \text{Follow}(\text{N}) = \{ \lambda \}$ $\text{M}[\text{N4}, \lambda] = \text{N4} \rightarrow \lambda$

$\text{O} \rightarrow (\text{P}) \mid =\text{H} \mid [\text{L}'] = \text{H} \mid \lambda$

$\text{First}(\text{(P)}) = \{ (\text{)} \}$ $\text{M}[\text{O}, (\text{)}] = \text{O} \rightarrow (\text{P})$

$\text{First}(\text{=H}) = \{ = \}$ $\text{M}[\text{O}, =] = \text{O} \rightarrow =\text{H}$

$\text{First}([\text{L}'] = \text{H}) = \{ [\text{ }] \}$ $\text{M}[\text{O}, [\text{ }]] = \text{O} \rightarrow [\text{L}'] = \text{H}$

$\text{Follow}(\text{O}) = \text{First}(\text{;}) + \text{Follow}(\text{H}) + \text{First}(\text{H}') \{ \text{;}, \text{+}, \text{-}, \text{*}, \text{/}, \text{\%} \}$

$\text{M}[\text{O}, \text{;}] = \text{M}[\text{O}, \text{+}] = \text{M}[\text{O}, \text{-}] = \text{M}[\text{O}, \text{*}] = \text{M}[\text{O}, \text{/}] = \text{M}[\text{O}, \text{\%}] = \text{O} \rightarrow \lambda$

$\text{P} \rightarrow \text{idP}' \mid \text{numP}' \mid \text{strP}' \mid \text{chP}' \mid \text{trueP}' \mid \text{falseP}'$

$\text{First}(\text{idP}') = \{ \text{id} \}$ $\text{M}[\text{P}, \text{id}] = \text{P} \rightarrow \text{idP}'$

$\text{First}(\text{numP}') = \{ \text{num} \}$ $\text{M}[\text{P}, \text{num}] = \text{P} \rightarrow \text{numP}'$

$\text{First}(\text{strP}') = \{ \text{str} \}$ $\text{M}[\text{P}, \text{str}] = \text{P} \rightarrow \text{strP}'$

$\text{First}(\text{chP}') = \{ \text{ch} \}$ $\text{M}[\text{P}, \text{ch}] = \text{P} \rightarrow \text{chP}'$

$\text{First}(\text{trueP}') = \{ \text{true} \}$ $\text{M}[\text{P}, \text{true}] = \text{P} \rightarrow \text{trueP}'$

$\text{First}(\text{falseP}') = \{ \text{false} \}$ $\text{M}[\text{P}, \text{false}] = \text{P} \rightarrow \text{falseP}'$

$\text{Follow}(\text{P}) = \{ \text{)} \}$ $\text{M}[\text{P}, \text{)}] = \text{P} \rightarrow \lambda$

$\text{P}' \rightarrow \text{,P} \mid \lambda$

$\text{First}(\text{,P}) = \{ \text{,} \}$ $\text{M}[\text{P}', \text{,}] = \text{P}' \rightarrow \text{,P}$

$\text{Follow}(\text{P}') = \text{Follow}(\text{P}) = \{ \text{)} \}$ $\text{M}[\text{P}', \text{)}] = \text{P}' \rightarrow \lambda$

Q-> cin >> id

First(cin>>id) = { cin } **M[Q, cin] = Q-> cin >> id**

R-> cout<<T

First(cout<<T) = { cout } **M[R, cout] = R-> cout<<T**

T->idT' | str T' | num T' | chT'

First(idT') = { id } **M[T, id] = T->idT'**
First(strT') = { str } **M[T, str] = T->strT'**
First(numT') = { num } **M[T, num] = T->numT'**
First(chT') = { ch } **M[T, ch] = T->chT'**

T'-> << T | λ

First(<< T) = { << } **M[T', <<] = T'-><<T**
Follow(T') = Follow(P) = Follow(R) = { ; } **M[T', ;] = T'-> λ**

U-> if(Z){F}V

First(if(Z){F}V) = { if } **M[U, if] = U-> if(Z){F}V**

V-> else{F}| λ

First(else{F}) = { else } **M[V, else] = V-> else{F}**
Follow(V) = first(F') = First(F) = First(G) + Follow(F) = { int, float, string, char, double, bool, id, cin, cout, if, while, const, static, break, }, return, } }
M[V, int] = M[V, float] = M[V, string] = M[V, char] = M[V, double] = M[V, bool] =
M[V, id] = M[V, cin] = M[V, cout] = M[V, if] = M[V, while] = M[V, const] = M[V,
static] = M[V, break] = M[V,)] = M[V, return] = = M[V, }] =

V-> λ

W-> while (Z){F}

First(while(Z){F}) = { while } **M[W, while] = W-> while(Z){F}**

X-> const Y| static Y

First(const Y) = { const } **M[X, const] = X-> const Y**
First(static Y) = { static } **M[X, static] = X-> static Y**

Y-> int id=num | float id=num | double id=num | string id=str | char id=ch | bool id=Y'

First(int id=num) = { int } **M[Y, int] = Y-> int id=num**
First(float id=num) = { float } **M[Y, float] = Y-> float id=num**
First(double id=num) = { double } **M[Y, double] = Y-> double id=num**
First(string id=str) = { string } **M[Y, string] = Y-> string id=str**
First(char id=ch) = { char } **M[Y, char] = Y-> char id=ch**
First(int bool id=Y') = { bool } **M[Y, bool] = Y-> bool id=Y'**

Y'-> true | false

First(true) = { true }
First(false) = { false }

M[Y', true] = Y'-> true
M[Y', false] = Y'-> false

Z-> ch Z2 | str Z2 | Z3 | (Z)Z2

First(ch Z2) = { ch }
First(str Z2) = { str }
First(Z3) = { -, id, num, ! }
First((Z)Z2) = { (}

M[Z,ch] = Z-> chZ2
M[Z,str] = Z-> strZ2
M[Z,-] = M[Z,id] = M[Z,num] = M[Z,!] = Z-> Z3
M[Z,(] = Z-> (Z)Z2

Z2-> A2 Z | λ

First(A2 Z) = { <, >, <=, >=, ==, ||, && }

M[Z2,<] = M[Z2,>] = M[Z2,<=] = M[Z2,>=] = M[Z2,==] = M[Z2,||] = M[Z2,&&] = Z2-> A2 Z

Follow(Z2)=Follow(Z)={) } **M[Z2,)] = Z2 -> λ**

Z3-> - Z3Z4 | ! Z3 | idZ4 | numZ4

First(- Z3Z4) = { - }
First(! Z3Z4) = { ! }
First(id Z3Z4) = { id }
First(num Z3Z4) = { num }

M[Z3,-] = Z3 -> -Z3Z4
M[Z3,!] = Z3 -> !Z3
M[Z3,id] = Z3 -> idZ4
M[Z3,num] = Z3 -> numZ4

Z4-> +Z3Z4 | -Z3Z4 | * Z3Z4 | /Z3Z4 | %Z3Z4 | A2Z | λ

First(+ Z3Z4) = { + }
First(- Z3Z4) = { - }
First(* Z3Z4) = { * }
First(/ Z3Z4) = { / }
First(% Z3Z4) = { % }

M[Z4,+] = Z4 -> +Z3Z4
M[Z4,-] = Z4 -> -Z3Z4
M[Z4,*] = Z4 -> *Z3Z4
M[Z4,/] = Z4 -> /Z3Z4
M[Z4,%] = Z4 -> %Z3Z4

First(A2Z) = { <,>,<=,>=,==,||,&& }

M[Z4,<] = M[Z4,>] = M[Z4,<=] = M[Z4,>=] = M[Z4,==] = M[Z4,||] = M[Z4,&&] = Z4 -> A2Z

Follow(Z4)={) } **M[Z4,)] = Z4-> λ**

A2-> < | > | <= | >= | == | || | &&

First(<) = { < }
First(>) = { > }
First(<=) = { <= }
First(>=) = { >= }
First(==) = { == }
First(||) = { || }
First(&&) = { && }

M[A2,<] = A2-> <
M[A2,>] = A2-> >
M[A2,<=] = A2-> <=
M[A2,>=] = A2-> >=
M[A2,==] = A2-> ==
M[A2,||] = A2-> ||
M[A2,&&] = A2-> &&

G-> I; | id O; | Q; |R; | U | W | X; | break;

First(I;) = { int, float, string, char, double, bool }

M[G, int]= M[G, float]= M[G, string]= M[G, char]= M[G, double]= M[G, bool] = G->I;

First(idO;) = { id } **M[G, id] = G->idO;**

First(Q;) = { cin }	M[G, cin] = G->Q;
First(R;) = { cout }	M[G, cout] = G->R;
First(U) = { if }	M[G, if] = G->U
First(W) = { while }	M[G, while] = G->W
First(X;) = { const, static }	M[G, const] = M[G, static] = G->X;
First(break;) = { break }	M[G, break] = G-> break;

F-> GF'

First(F) = First(G) = { int, float, string, char, double, bool, id, cin, cout, if, while, const, static, break }

M[F, int] = M[F, float] = M[F, string] = M[F, char] = M[F, double] = M[F, bool] = M[F, id] = M[F, cin] = M[F, cout] = M[F, if] = M[F, while] = M[F, const] = M[F, static] = M[F, break] = F-> GF'

F'->F | λ

First(F') = First(F) = First(G) = { int, float, string, char, double, bool, id, cin, cout, if, while, const, static, break }

M[F', int] = M[F', float] = M[F', string] = M[F', char] = M[F', double] = M[F', bool] = M[F', id] = M[F', cin] = M[F', cout] = M[F', if] = M[F', while] = M[F', const] = M[F', static] = M[F', break] = F'-> F

Follow(F') = Follow(F) = First() + First(E) = { }, return }

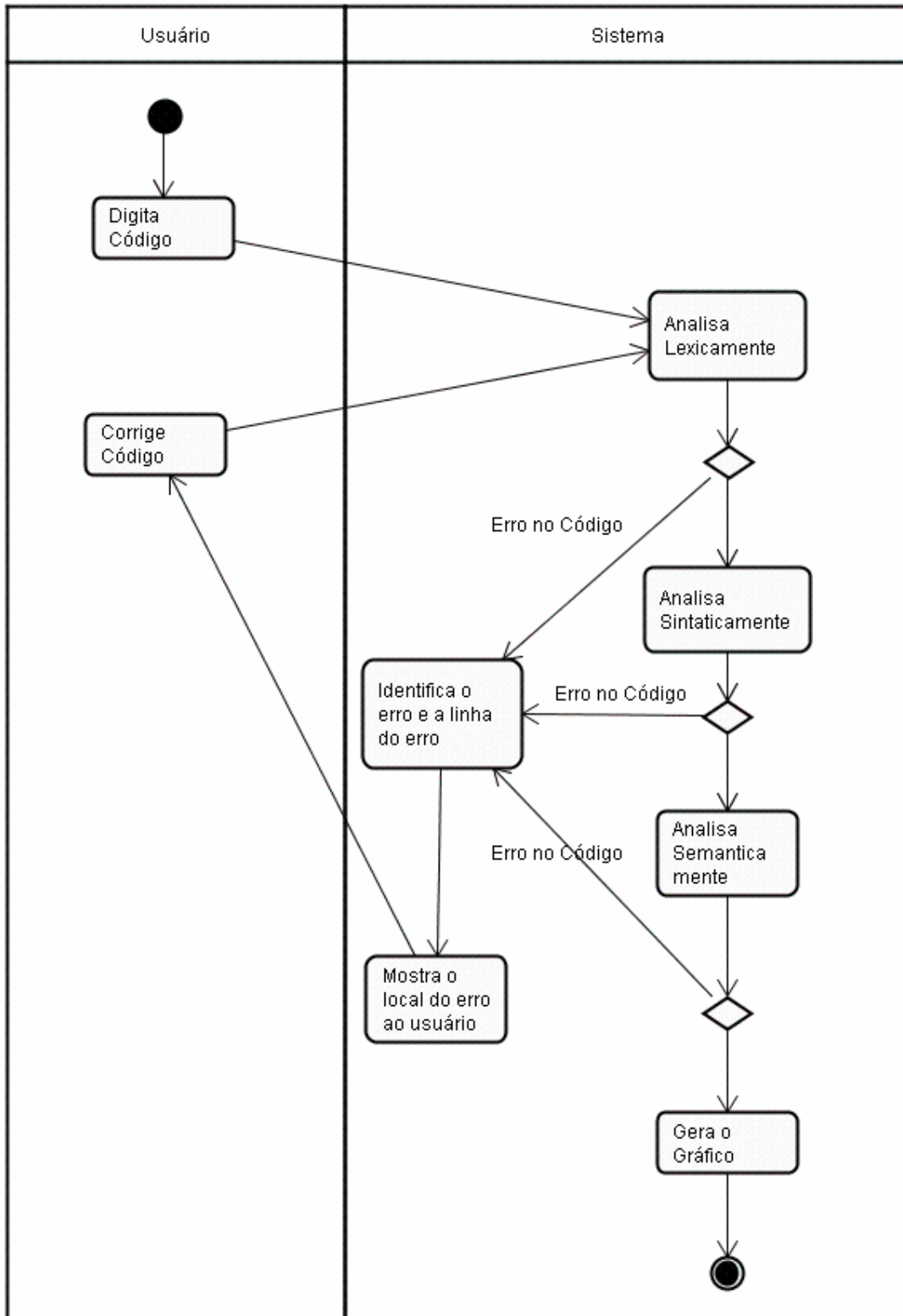
M[F', }] = M[F', return] = F'-> λ

APÊNDICE C – TABELA DE ERROS

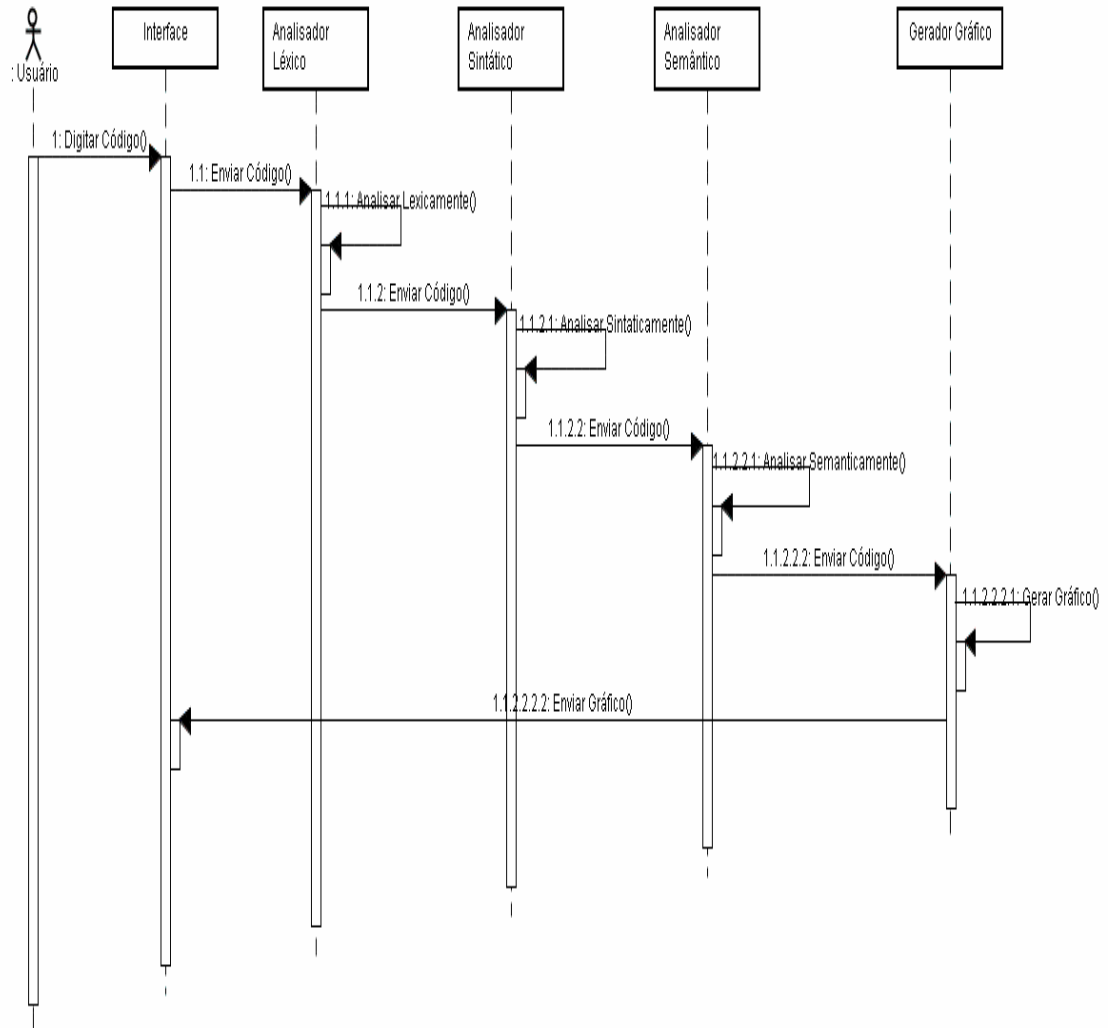
Número	Tipo de erro	Ação
1	# esperado	Inserir #
2	# include esperados	Inserir # include
3	# include < esperados	Inserir # include <
4	# include < iostream esperados	Inserir # include < iostream
5	# include < iostream > esperados	Inserir #include<iostream>
6	# include < iostream > using esperados	Inserir #include<iostream> using
7	# include < iostream > using namespace esperados	Inserir #include<iostream> using namespace
8	# include < iostream > using namespace std esperados	Inserir #include<iostream> using namespace std
9	# include < iostream >	Inserir #include<iostream>
10	Token inesperado	Descartar token
11	Token inesperado	Descartar token
12	using esperado	Inserir using
13	using namespace esperados	Inserir using namespace
14	Using namespace std	Inserir using namespace std
15	Token inesperado	Descartar token
16	Declarador de função esperado	Inserir declarador
17	Declarador de função e id esperados	Inserir declarador e id
18	Token inesperado	Descartar token
19	Declarador de parâmetro esperado	Inserir declarador
20	Token inesperado	Descartar token
21	Declarador de parâmetro esperado	Inserir declarador
22	Token inesperado	Descartar token
23	, esperada	Inserir ,
24	Token inesperado	Descartar token
25	[esperada	Inserir [
26	, esperada	Inserir ,
27	Token inesperado	Descartar token
28	{ esperada	Inserir {
29	Token inesperado	Descartar token
30	Declarador de função esperado	Inserir declarador
31	Declarador de função e id esperados	Inserir declarador e id
32	str esperado	Inserir str e descartar token
33	Token inesperado	Descartar token
34	Expressão, id, num, ch ou str esperados	Inserir
35	Token inesperado	Descartar token
36	cout esperado	Inserir cout
37	cin esperado	Inserir cin
38	Token inesperado	Descartar token
39	id esperado	Inserir id
40	Token inesperado	Descartar token
41	= esperado	Inserir =
42	= { esperado	Inserir = {
43	Atribuições esperadas	Sincronizar

44	Token inesperado	Descartar token
45	Atribuições e } esperadas	Sincronizar
46	Token inesperado	Descartar token
47	, esperada	Inserir ,
48	} esperada	Inserir }
49	Token inesperado	Descartar token
50	(esperado	Inserir (
51	Token inesperado	Descartar token
52) esperado	Inserir)
53	Token inesperado	Descartar token
54) esperado	Inserir)
55	Token inesperado	Descartar token
56	Token inesperado	Descartar token
57	else esperado	Inserir else
58	Token inesperado	Descartar token
59	Atribuição de booleano esperado	Inserir atribuição
60	Token inesperado	Descartar token
61	Condição esperada	Sincronizar
62	Token inesperado	Descartar token
63	Operador de comparação esperado	Sincronizar
64	Token inesperado	Descartar token
65	include esperado	Inserir include
66	iostream esperado	Inserir iostream
67	Include> esperado	Inserir include>
68	namespace esperado	Inserir namespace
69	namespace std; esperado	Inserir namespace std;
70	namespace std esperado	Inserir namespace std
71	std esperado	Inserir std
72	std; esperado	Inserir std;
73	Token inesperado	Descartar token
74	(esperado	Inserir (
75) esperado	Inserir)
76	[esperado	Inserir [
77] esperado	Inserir]
78	{ esperado	Inserir {
79	}esperado	Inserir }
80	+ - * / ou % esperados	Sincronizar
81	= esperado	Inserir =
82	id ou num esperados	Sincronizar
83	str ou ch esperados	Sincronizar
84	true ou false esperado	Sincronizar
85	, esperado	Inserir ,
86	; esperado	Inserir ;
87		
88		
89		

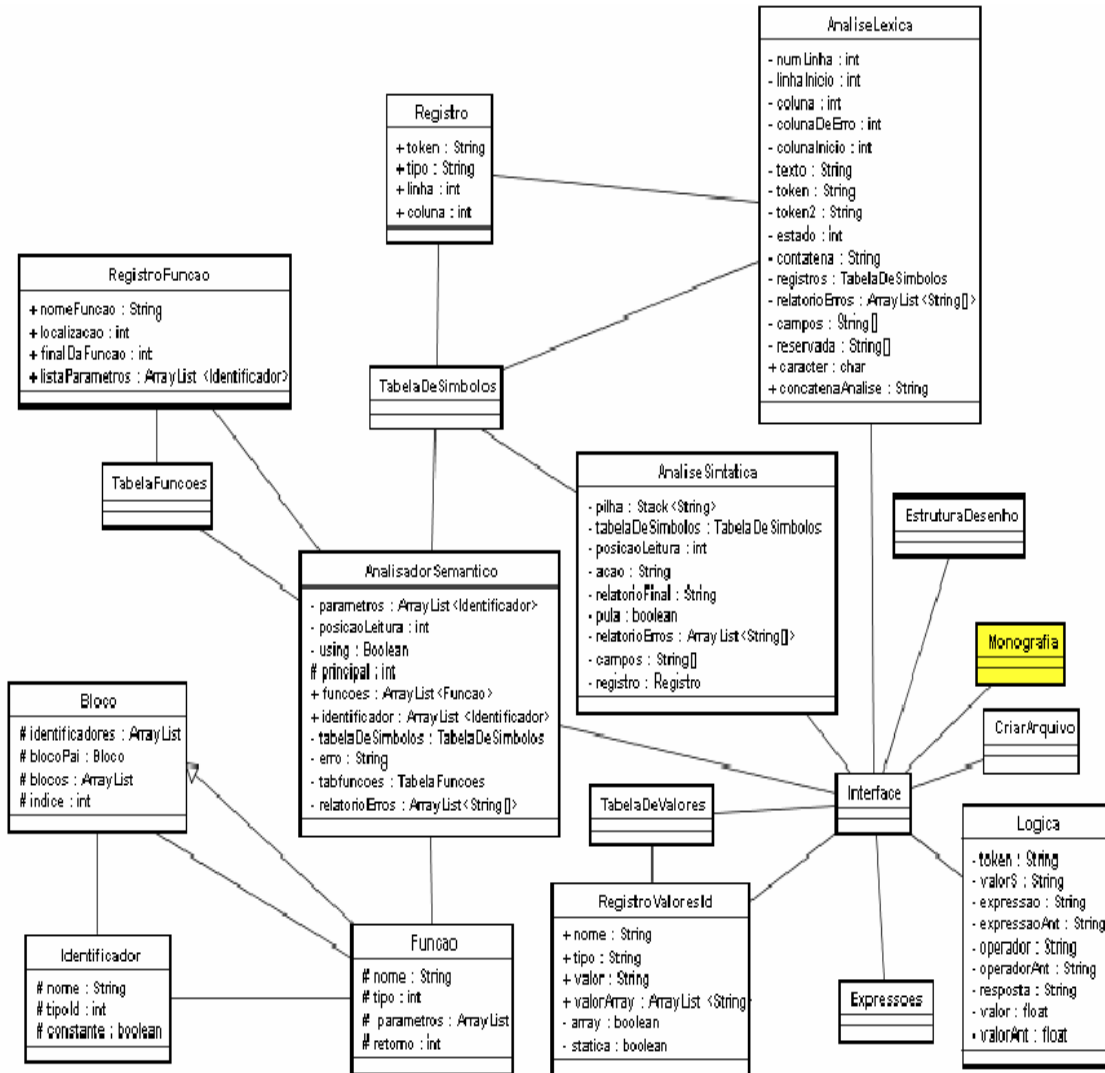
APÊNDICE D – DIAGRAMA DE ATIVIDADE



APÊNDICE E – DIAGRAMA SEQUÊNCIA



APÊNDICE F – DIAGRAMA DE CLASSES



1. Introdução

O presente documento tem como propósito apresentar a Monografia de conclusão do curso de Ciência da Computação do graduando João Paulo Vieira Bahia. Este trabalho se propõe a construir um ambiente computacional de aprendizagem, para auxiliar os alunos de Linguagem de Programação I (LPI), que terá o nome de AVL P (Ambiente Virtual de Linguagem de Programação). Esse ambiente mostrará graficamente como o computador “interpreta” a seqüência de instruções escrita pelo programador por meio do código escrito em uma linguagem de alto nível, para que os alunos visualizem e entendam esse processo, tornando mais fácil a aprendizagem. Esse ambiente permite ao aluno construir seu próprio programa, executando-o e tendo a oportunidade de entender conceitos, tais como criação e escopo das variáveis, comandos, funções e arrays através de um processo de simulação.

1.1. Motivação

O meio mais eficaz de comunicação entre as pessoas é a linguagem natural (idioma). Na programação de computadores, uma linguagem de programação serve de meio de comunicação entre o indivíduo que deseja resolver um determinado problema e o computador escolhido para resolvê-lo ou ajudar a resolvê-lo (Price, 2005, p. 1).

Aprender a programar é um processo difícil e exigente para maioria dos alunos. Até mesmo a aprendizagem de conceitos básicos e a sua aplicação na resolução de problemas concretos coloca problemas difíceis a muitos estudantes. Estas dificuldades são comuns a muitos países e instituições de ensino. As causas destas dificuldades são, por certo, variadas e específicas de cada caso, mas há algumas que são referidas com grande frequência:

- a necessidade de um bom nível de conhecimentos e prática de técnicas de resolução de problemas;
- a existência de turmas com alunos com *backgrounds* diversificados acerca das matérias em questão, traduzindo-se em ritmos de aprendizagem muito diferenciado;
- a impossibilidade de um acompanhamento individualizado ao aluno, devido à existência de turmas demasiado grandes e ao tempo dentro do currículo escolar ser insuficiente;

- a exigência de um estudo aplicado baseado na prática e, por isso, bastante diferente do requerido pela maioria das disciplinas (mais baseado em noções teóricas, implicando muita leitura e alguma memorização);
- o elevado nível de abstração envolvido;
- as metodologias tradicionais de ensino que privilegiam a aprendizagem de conceitos dinâmicos utilizando principalmente abordagens e materiais de índole estática;
- as linguagens de programação usuais apresentam sintaxes complexas e não têm representações visuais dos algoritmos, o que não contribui para uma melhor compreensão(Mendes, 2007);

As conseqüências dessas dificuldades têm levado educadores a procurar estratégias e materiais que possam ajudar a minorar essas dificuldades sentidas tanto por professores como por alunos. Uma das estratégias encontradas é o desenvolvimento de ferramentas que auxiliem os alunos e professores no processo ensino e aprendizagem das disciplinas de informática.

Foi com esse intuito que o AVLPL foi criado, pois mostra ao aluno o que computador faz internamente quando está executando um código e é o próprio aluno quem escreve o código a ser executado, isso dá uma dinâmica maior ao aprendizado, já que o aluno poderá escrever inúmeros códigos e cada um desses códigos terá uma simulação diferente.

1.2. Objetivos do Trabalho

O AVLPL é um ambiente que facilita o aprendizado da linguagem de programação. Seu principal objetivo é motivar os estudantes de linguagem de programação através de um processo de simulação, isto é, o aluno irá digitar o código e o AVLPL irá compilá-lo e simulará sua execução. Com isso o estudante verá passo a passo o que cada linha do código digitado faz internamente no computador.

1.3. Organização da Monografia

No Capítulo 2 são apresentadas as tecnologias utilizadas na construção do Ambiente Virtual de Linguagem de Programação (AVLPL) e algumas ferramentas com o propósito semelhante ao do AVLPL.

No capítulo 3 será apresentado o AVLPL, seus propósitos, suas funcionalidades e o modo como foi planejado e construído.

No Capítulo 4 discutem-se as conclusões do trabalho e possibilidades de trabalhos futuros.

2. Ferramentas existentes

Nesta seção, serão mostradas as ferramentas utilizadas para a construção do AVLPL, bem como as ferramentas existentes criadas para o ensino de linguagem de programação.

2.1 Tradutores de linguagem de programação

Tradutor, no contexto de linguagem de programação, é um sistema que aceita como entrada um programa escrito em uma linguagem de programação (linguagem fonte) e produz como resultado um programa equivalente em outra linguagem (linguagem objeto).

Um tipo de tradutor é o compilador, que mapeia programas escritos em linguagem de alto nível para programas equivalentes em linguagem simbólica ou linguagem de máquina.

A Figura 1 mostra um processo de execução de um programa escrito em uma linguagem de alto nível. Esse processo tem basicamente dois passos.

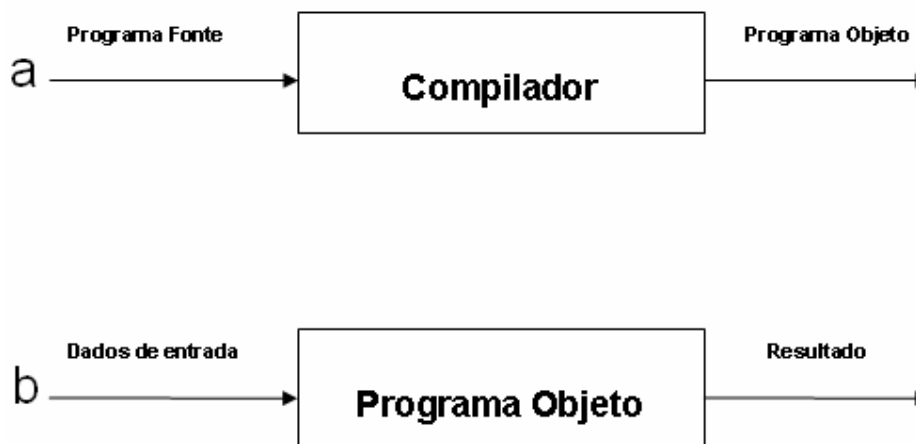


Figura 1: Execução de um programa fonte

Fonte: Price, 2005.

O intervalo de tempo no qual ocorre a conversão de um programa fonte para um programa objeto é chamado de tempo de compilação, como mostra a figura 1(a). O código objeto é executado no intervalo de tempo chamado de tempo de execução. Como está representado na Figura 1(b), o programa fonte e os dados são

processados em momentos distintos, respectivamente, tempo de compilação e tempo de execução.

2.1.1 Estruturas de um tradutor

Em geral, os tradutores de linguagem de programação são bastante complexos. Porém, devido à experiência acumulada ao longo dos anos, e principalmente, ao desenvolvimento de teorias relacionadas às tarefas de análise e síntese de programas, existe um consenso sobre a estrutura básica desses processadores.

O processo de tradução é comumente estruturado em fases, no qual cada fase se comunica com a seguinte, através de uma linguagem intermediária adequada. Essas fases podem ser vistas na figura 2, e serão melhor detalhadas nas próximas seções.

2.1.2 Análise Léxica

O analisador léxico lê caractere a caractere do texto fonte, verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando tokens, e desprezando comentários e espaços em branco desnecessários. Os tokens constituem classes de símbolos tais como palavras reservadas, delimitadores, identificadores, etc(Prive, 2005, 7).

O analisador léxico coleta caracteres em grupamentos lógicos e atribue códigos internos aos grupamentos de acordo com sua estrutura. Os grupamentos de caracteres são chamados de lexemas. Os códigos internos são chamados tokens. Os lexemas são identificados pelo casamento da cadeia de caracteres de entrada com os padrões. Considere o seguinte exemplo de uma instrução da atribuição: recebe = valor – numero/ 10; (SEBESTA, 2003, 155)

Na tabela 1 estão os símbolos e os lexemas dessa instrução.

Os analisadores léxicos são, usualmente, especificados através de notações para a descrição de linguagens regulares tais como os autômatos finitos, expressões regulares ou gramáticas regulares.

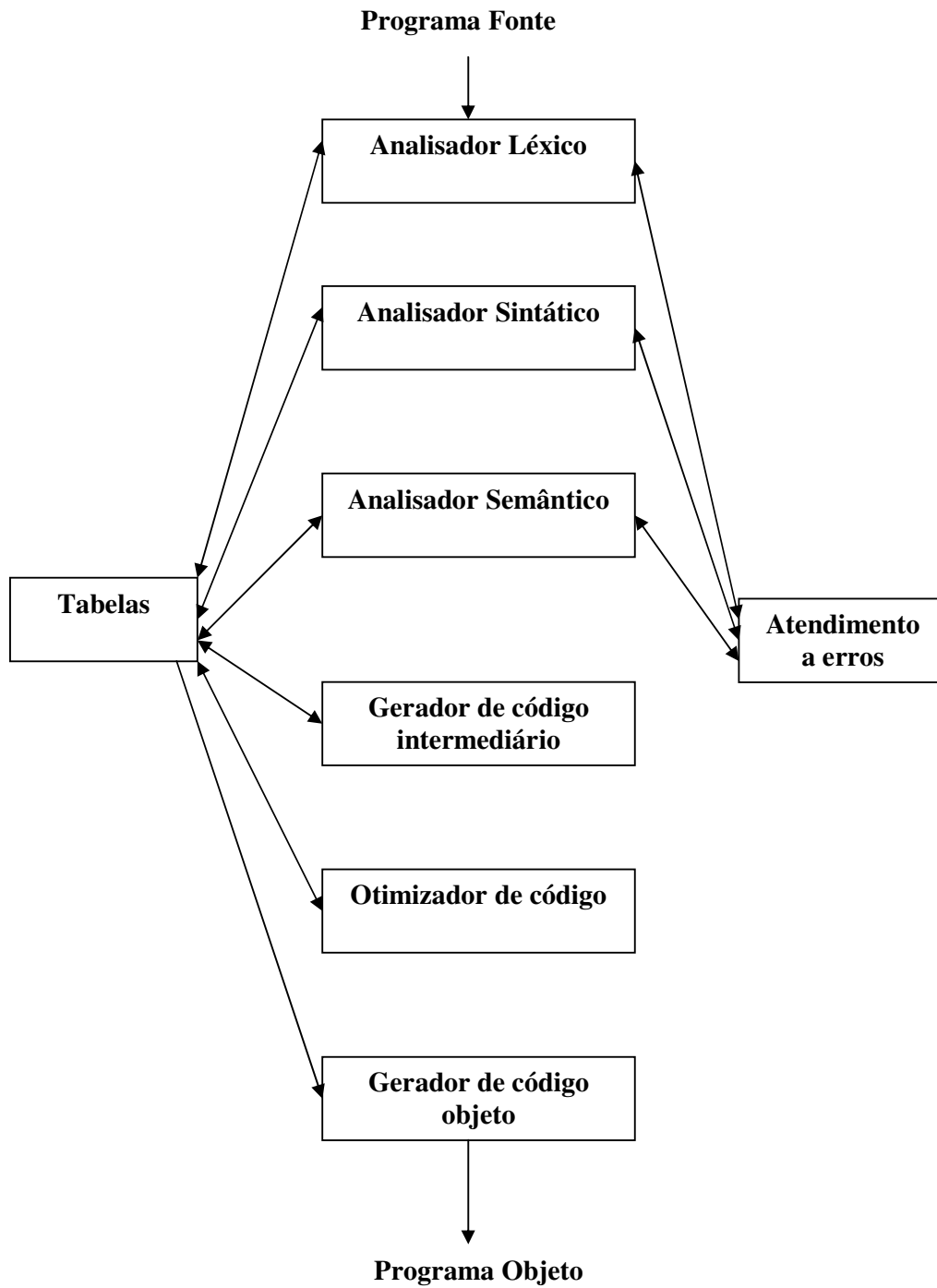


Figura 2: Estrutura de um compilador

Fonte: (Price, 2005)

Tabela 1-Símbolos e lexemas

Símbolo	Lexema
Identificador	recebe
Op_Atribuição	=
Identificador	valor
Op_Subtração	-
Identificador	numero
Op_Divisão	/
Literal int	10
Ponto e vírgula	;

Fonte: (SEBESTA, 2003, 155)

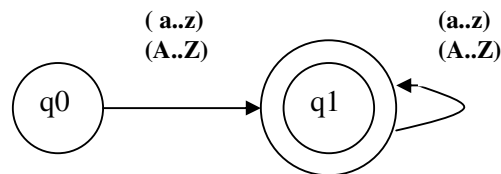


Figura 3: Um diagrama de estados para reconhecer nomes

Fonte: (SEBESTA, 1999)

Os vértices do diagrama de estado são rotulados com os nomes dos estados. Os arcos são rotulados com os caracteres de entrada que causam as transições. Cada vez que um caractere é inserido, o estado muda. O início é no vértice **q0** e o vértice com dois arcos(**q1**) representa um estado de reconhecimento. Se ao final das transições o ponteiro parar no vértice de reconhecimento, quer dizer que a palavra é aceita pela linguagem, caso contrário ela não é aceita pela linguagem.

Durante o processo de análise léxica, são desprezados caracteres não significativos como espaços em branco. Além de reconhecer os símbolos léxicos, o analisador também realiza outras funções, como armazenar alguns desses símbolos em tabelas internas e indicar a ocorrência de erros léxicos. A seqüência de tokens reconhecida pelo analisador léxico é utilizada como entrada pelo módulo seguinte do simulador que é o analisador sintático (Araújo, 2005).

2.1.2.1 Tokens

Os tokens, ou símbolos léxicos, são as unidades básicas do texto do programa. Cada token é responsável por três informações:

- Classe do token: representa o tipo do token reconhecido. Como por exemplo, os identificadores, constantes numéricas, cadeias de caracteres, palavras reservadas, operadores e separadores.
- Valor do token: depende da classe. Para tokens da classe constante inteira, por exemplo, o valor do token pode ser o número inteiro representado pela constante.
- Posição do token: Indica a linha e a coluna em que o token se encontra no texto. Essa informação é utilizada, principalmente, para indicar o local onde ocorre um erro.

2.1.3 Tabela de símbolos

A tabela de símbolos é uma estrutura de dados gerada pelo compilador com o objetivo de armazenar informações sobre os nomes (identificadores de variáveis, de parâmetros, de funções, de procedimentos, etc.) definidos no programa fonte. Ela começa a ser construída durante a análise léxica, quando os identificadores são reconhecidos. Quando o identificador é encontrado, o analisador léxico armazena-o na tabela. Toda vez que um identificador é reconhecido no programa fonte, a tabela de símbolos é consultada, a fim de verificar se o nome já está registrado; caso não esteja, é feita uma inserção na tabela.

Essa tabela serve como um banco de dados para o processo de compilação. Seu principal conteúdo são informações sobre tipos e atributos de cada nome definido pelo usuário no programa. Essas informações são colocadas na tabela de símbolos pelos analisadores léxico e sintático e usadas pelo analisador semântico e pelo gerador de código (Oliveira, 2007).

2.1.4 Análise Sintática

A análise sintática é a segunda fase de um tradutor. Sua função é verificar se a estrutura gramatical do programa está correta, isto é, se essa estrutura foi formada usando as regras gramaticais da linguagem.

O Analisador sintático identifica seqüências de símbolos que constituem estruturas sintáticas (por exemplo, expressões, comandos), através de uma varredura da seqüência de tokens do programa fonte. Ele produz uma estrutura em

árvore, chamada árvore de derivação, que exibe a estrutura sintática do texto fonte resultante da aplicação das regras gramaticais da linguagem. Outra função do reconhecedor sintático é a detecção de erros de sintaxe identificando a posição e o tipo de erro ocorrido. Mesmo que erros tenham sido encontrados, o analisador sintático deve tentar recuperá-los prosseguindo a análise do texto restante.

2.1.4.1 Análise Preditiva Tabular

Esse tipo de analisador sintático implementa um autômato de pilha controlado por uma tabela de análise. O princípio do reconhecimento preditivo é a determinação da produção a ser aplicada, cujo lado direito irá substituir o símbolo não-terminal que se encontra no topo da pilha. O analisador busca a produção a ser aplicada na tabela de análise, levando em conta o não-terminal no topo da pilha e o token sob o cabeçote de leitura.

O analisador é controlado por um programa que se comporta conforme três ações:

- 1) se $X = a = \$$, o analisador pára, aceitando a sentença
- 2) se $X = a \neq \$$, o analisador desempilha a e avança o cabeçote de leitura para o próximo símbolo na fita.
- 3) se X é um símbolo não-terminal, o analisador consulta a entrada $M[X, a]$ da tabela de análise. Essa entrada poderá conter uma produção da gramática ou ser vazia. Supondo $M[X, a] = \{X \rightarrow UVW\}$, o analisador substitui X (que está no topo da pilha) por WVU (ficando U no topo) e retorna a produção aplicada. Se $M[X, a]$ é vazia, isso corresponde a uma situação de erro; nesse caso, o analisador chama uma rotina de tratamento de erro.

Considerando X como símbolo no topo da pilha e a como terminal da fita de entrada sob o cabeçote de leitura.

O símbolo da base da pilha é $\$$, estado inicial do analisador. O analisador é dirigido pela Tabela de análise, cuja estrutura é mostrada na Figura 4.

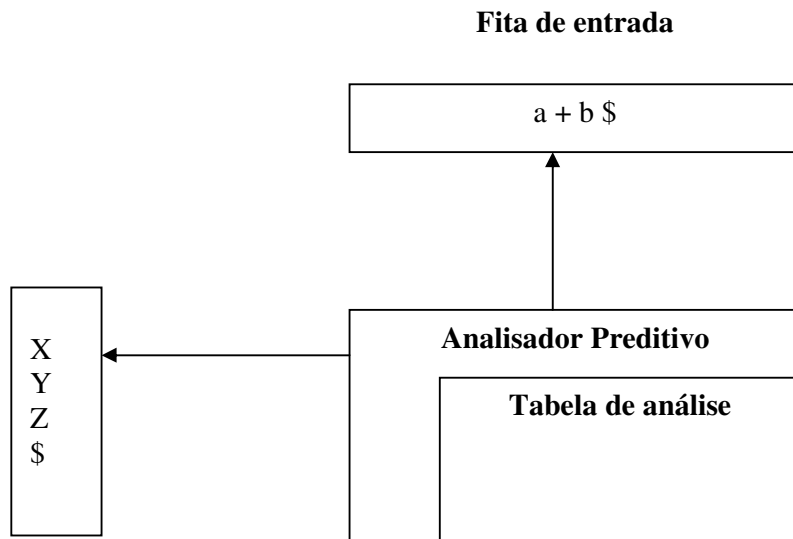


Figura 4: Análise Preditiva Tabular

Fonte: (Price, 2005)

Para melhor explicar observe o exemplo:

Considere a gramática não ambígua abaixo que gera expressões lógicas:

$E \rightarrow E \vee T \mid T$
 $T \rightarrow T \& F \mid F$
 $F \rightarrow \neg F \mid id$

Eliminando-se a recursividade à esquerda das produções que definem E e T, obtém-se:

$E \rightarrow TE'$
 $E' \rightarrow \vee TE' \mid \lambda$
 $T \rightarrow FT'$
 $T' \rightarrow \& FT' \mid \lambda$
 $F \rightarrow \neg F \mid id$

A partir da gramática deve-se construir a tabela de análise. Para construção dessa tabela, é necessário computar duas funções associativas à gramática: as funções **FIRST** e **FOLLOW**. (Price, 2005, 45)

2.1.4.1.1 Definição de FIRST(α)

Se α é uma seqüência de símbolos da gramática, então **FIRST**(α) é o conjunto de terminais que iniciam formas seqüenciais derivadas a partir de α . Se $\alpha \rightarrow^* \lambda$, então a palavra vazia também faz parte do conjunto.

Para computar **FIRST(X)** para um símbolo **X** da gramática, aplicam-se as regras abaixo, até que não se possa adicionar mais terminais ou λ ao conjunto em questão.

- 1) Se **a** é terminal, então **FIRST(a)={a}**.
- 2) Se $X \rightarrow \lambda$ é uma produção, então adicione λ a **FIRST(X)**.
- 3) Se $X \rightarrow Y_1 Y_2 \dots Y_k$ é uma produção e, para algum **i**, todos $Y_1 Y_2 \dots Y_{i-1}$ derivam λ , então **FIRST(Y_i)** está em **FIRST(X)**, juntamente com todos os símbolos não- λ de **FIRST(Y₁)**, **FIRST(Y₂)**,..., **FIRST(Y_{i-1})**. O símbolo λ será adicionado a **FIRST(X)** apenas se todo $Y_j(j=1,2,\dots,k)$ derivar λ . (Price, 2005, 48)

2.1.4.1.2 Definição de FOLLOW(α)

A função **FOLLOW** é definida para símbolos não-terminais. Sendo **A** um não-terminal, **FOLLOW(A)** é o conjunto de terminais **a** que podem aparecer imediatamente à direita de **A** em alguma forma sentencial. Isto é, o conjunto de terminais **a**, tal que existe uma derivação da forma $S \rightarrow^* \alpha A a \beta$ para α e β quaisquer.

Para computar **FOLLOW(X)**, aplicam-se as regras abaixo até que não se possa adicionar mais símbolos ao conjunto.

- 1) Se **S** é símbolo inicial da gramática e $\$$ é o marcador de fim da sentença, então $\$$ está em **FOLLOW(S)**.
- 2) Se existe produção do tipo $A \rightarrow \alpha A X \beta$, então todos os símbolos de **FIRST(β)**, exceto λ , fazem parte do **FOLLOW(X)**.
- 3) Se existe produção do tipo $A \rightarrow \alpha X$, ou $A \rightarrow \alpha X \beta$, sendo que $\beta \rightarrow^* \lambda$, então todos os símbolos que estiverem em **FOLLOW(A)** fazem parte de **FOLLOW(X)**. (Price, 2005, 48)

2.1.4.1.3 Algoritmo para construção uma tabela de análise preditiva.

Entrada: gramática **G**.

Saída: Tabela de Análise **M**.

Método:

- 1) Para cada produção $A \rightarrow \alpha$ de **G**, execute os passos 2 e 3 (para criar a linha **A** da tabela **M**).
- 2) Para cada terminal **a** de **FIRST(α)**, adicione a produção $A \rightarrow \alpha$ a **M[A,a]**.
- 3) Se **FIRST(α)** inclui a palavra vazia, então adicione $A \rightarrow \alpha$ a **M[A,b]** para cada **b** em **FOLLOW(A)**.

Aplicando o algoritmo à gramática, obtêm-se as seguintes entradas para a tabela de análise.

Para $E \rightarrow TE'$	tem-se $\text{First}(TE') = \{\neg, \text{id}\}$	$M[E, \neg] = M[E, \text{id}] = E \rightarrow TE'$
Para $E' \rightarrow v TE'$	tem-se $\text{First}(v TE') = \{v\}$	$M[E', v] = E' \rightarrow v TE'$
Para $E' \rightarrow \lambda$	tem-se $\text{Follow}(E') = \{\$\}$	$M[E', \$] = E' \rightarrow \lambda$
Para $T \rightarrow FT'$	tem-se $\text{First}(FT') = \{\neg, \text{id}\}$	$M[T', \neg] = M[T, \text{id}] = T \rightarrow \& FT'$
Para $T \rightarrow \& FT'$	tem-se $\text{First}(\& FT') = \{\&\}$	$M[T', \&] = T \rightarrow \& FT'$
Para $T' \rightarrow \lambda$	tem-se $\text{Follow}(T') = \{v, \$\}$	$M[T', \$] = M[T', v] = T' \rightarrow \lambda$
Para $F \rightarrow \neg F$	tem-se $\text{First}(\neg F) = \{\neg\}$	$M[F, \neg] = M[T', \$] = F \rightarrow \neg F$
Para $F \rightarrow \text{id}$	tem-se $\text{First}(\text{id}) = \{\text{id}\}$	$M[F, \text{id}] = F \rightarrow \text{id}$

Tabela 2-Tabela de análise preditiva

	id	v	&	\neg	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E'		$E' \rightarrow v TE'$			$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T'		$T' \rightarrow \lambda$	$T \rightarrow \& FT'$		$T' \rightarrow \lambda$
F	$F \rightarrow \text{id}$			$F \rightarrow \neg F$	

Fonte: (Price, 2005, 51)

Se em cada entrada da tabela, existe apenas uma produção, então a gramática que originou a tabela é dita ser do tipo LL(1).

Para a sentença **id v id & id**, o reconhecedor preditivo realiza os movimentos mostrados na figura a seguir (Price, 2005, 47).

2.1.4.2 Recuperação de erro

Na tabela LL, as lacunas representam situações de erro e devem ser usadas para chamar rotinas de recuperação. Um modo de recuperação de erro é chamado de recuperação local, nele o analisador tenta recuperar o erro, fazendo alterações sobre um símbolo apenas: desprezando o token da entrada, ou substituindo-o por outro, ou inserindo um novo token, ou ainda, removendo um símbolo da pilha.

Na recuperação local, a tabela LL deve ser expandida para incluir as situações em que ocorre discrepância entre o token do topo da pilha e o da fita de entrada. A tabela 4 é uma ampliação da tabela de análise preditiva mostrada anteriormente. A tabela anterior é aumentada com linhas para os símbolos terminais.

As lacunas que permanecem vazias representam situações que jamais ocorrerão durante a análise.

As rotinas de erro poderiam ser as seguintes:

Erro1: insere token id na entrada e emite “operando esperado”;

Erro2: Descarta token id na entrada e emite “operando descartado”;

Tabela 3-Movimentos de um reconhecedor preditivo analisando uma sentença

Pilha	Entrada	Ação
\$E	id v id & id\$	$E \rightarrow TE'$
\$E'T	id v id & id\$	$T \rightarrow FT'$
\$E'T'F	id v id & id\$	$F \rightarrow id$
\$E'T'id	id v id & id\$	Desempilha e lê símbolo
\$E'T'	v id & id\$	$T' \rightarrow \lambda$
\$E'	v id & id\$	$E' \rightarrow v TE'$
\$E'Tv	v id & id\$	Desempilha e lê símbolo
\$E'T	id & id\$	$T \rightarrow FT'$
\$E'T'F	id & id\$	$F \rightarrow id$
\$E'T'id	id & id\$	Desempilha e lê símbolo
\$E'T'	& id\$	$T \rightarrow \& FT'$
\$E'T'F&	& id\$	Desempilha e lê símbolo
\$E'T'F	id\$	$F \rightarrow id$
\$E'T'id	id\$	Desempilha e lê símbolo
\$E'T'	\$	$T' \rightarrow \lambda$
\$E'	\$	$E' \rightarrow \lambda$
\$	\$	Aceita a sentença!

Fonte: (Price, 2005, 47)

Muitas vezes, o analisador sintático opera conjuntamente com o analisador semântico, cuja principal atividade é determinar se as estruturas sintáticas analisadas fazem sentido ou não, esse tipo de técnica é chamado de Tradução Dirigida por Sintaxe. As ações semânticas são associadas às regras de produção da gramática de modo que, quando uma dada produção é processada, essas ações são executadas. A execução dessas ações pode gerar ou interpretar código, armazenar informações na tabela de símbolos, emitir mensagens de erro, etc. (Price, 2005, 75)

Tabela 4 -Tabela de análise preditiva com correção de erros

	Id	v	&	¬	\$
E	$E \rightarrow TE'$	Erro 1	Erro 1	$E \rightarrow TE'$	
E'		$E' \rightarrow v TE'$			$E' \rightarrow \lambda$
T	$T \rightarrow FT'$	Erro 1	Erro 1	$T \rightarrow FT'$	
T'	Erro 2	$T' \rightarrow \lambda$	$T \rightarrow \& FT'$	Erro 2	$T' \rightarrow \lambda$
F	$F \rightarrow id$			$F \rightarrow \neg F$	
Id	Desempilha				
V		Desempilha			
&			Desempilha		
¬				Desempilha	
\$					Aceita

Fonte: (Price, 2005, 76)

2.1.5 Análise Semântica

A análise semântica tem a função principal de determinar se as estruturas sintáticas analisadas fazem sentido ou não. Por exemplo, o analisador semântico verifica se um identificador declarado como variável é usado como tal; se existe compatibilidade entre os operandos e operadores em expressões e etc (Price, 2005, 9).

2.2 Paradigmas de programação

Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa.

O significado da palavra paradigma quer dizer um "Conjunto de unidades susceptíveis de aparecer num mesmo contexto. No paradigma, as unidades têm, pelo menos, um traço em comum que as relaciona, formando conjuntos abertos ou fechados, segundo a natureza das unidades".

As linguagens de programação são agrupadas em quatro diferentes paradigmas:

- **Procedural ou imperativo:** é baseado na perspectiva do computador. Isso se reflete na execução seqüencial dos comandos e no uso de armazenamento de dados variável, conceitos que são baseados na maneira em que os computadores executam os programas no nível da máquina. Um programa

deste modelo consiste numa seqüência de modificações nos registradores/acumuladores do computador.

- **Orientação à lógica:** é mais relacionado à perspectiva da pessoa. O problema é focado do ponto de vista lógico. O programa é uma descrição lógica do problema expresso de maneira formal, similar ao modo em que o cérebro humano raciocina para obter uma solução.
- **Funcional:** o modelo funcional focaliza o processo de solução de um problema. A visão funcional resulta num programa que descreve as operações que devem ser executadas para resolver um problema.
- **Orientada a objetos:** reflete o problema corrente. O programa consiste de objetos que trocam mensagens entre si. Estes objetos correspondem diretamente aos objetos correntes, tais como pessoas, máquinas, departamentos, documentos e assim por diante (Oliveira, 2007).

O C++ é uma linguagem multi-paradigma. Ela é, ao mesmo tempo, uma linguagem procedural e orientada a objeto, por isso esses dois paradigmas serão explicados com mais detalhes nas seções seguintes.

2.2.1 Programação Orientada a Procedimentos (Procedural)

O foco é no processamento, o algoritmo necessário para executar a programação desejada. Linguagens suportam esse paradigma providenciando facilidades para passar argumentos para funções e retornando valores de argumentos, maneiras de distinguir entre diferentes tipos de argumentos, diferentes espécies de funções (Stroustrup, 1999).

Um programa é visto como uma seqüência de procedimentos locais que se “comunicam” através de dados locais, denominados parâmetros.

Assim, tem-se, um único programa que é dividido em pequenas partes, chamadas “procedures”.

A figura 5 ilustra a programação orientada a procedimentos.

2.2.2 Programação Orientada a Objetos

É um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos. Nesse tipo de paradigma, o programador divide conceitualmente o “problema” em partes independentes (objetos), quem podem conter atributos que os descrevem, e que implementam o comportamento do sistema através de funções

definidas nestes objetos (métodos). Objetos (e seus métodos) fazem referência a outros objetos e métodos.

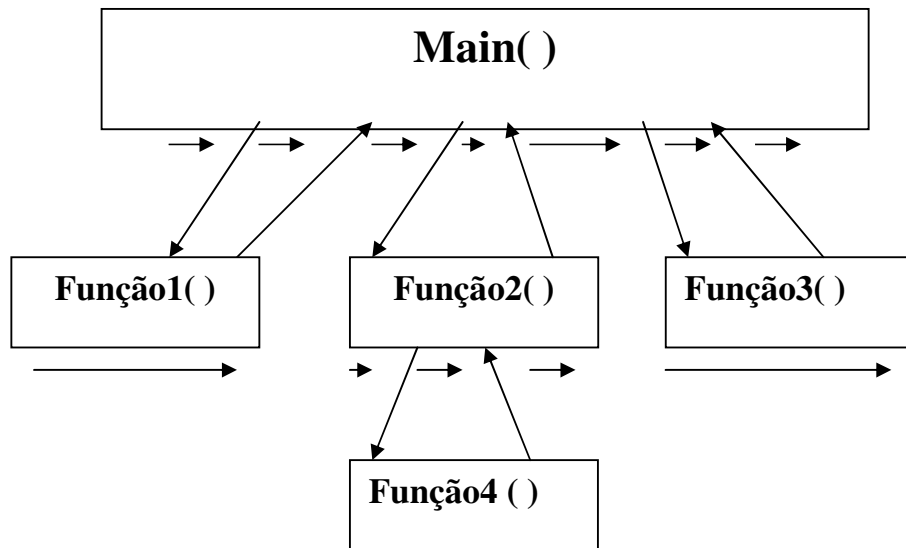


Figura 5: Ilustração de procedimentos em C++

Fonte: Banco de imagens do autor

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com os outros objetos.

2.3. C++

A linguagem C++ foi desenvolvida por Bjarne Stroustrup durante a década de 1980, com o objetivo de melhorar a linguagem de programação C, mantendo a compatibilidade com aquela linguagem. Esta linguagem foi escolhida porque possuía uma proposta de uso genérico, era rápida e também suportada por várias plataformas.

Algumas linguagens que serviram de inspiração para o desenvolvimento do C++ foram as linguagens ALGOL 68, Ada, CLU, ML e Simula[3].

Por muito tempo foi encarada muitas vezes como um superconjunto do C, entretanto em 1999 o novo padrão ISO para a linguagem C, conhecido como C99, tornou as duas linguagens diferentes entre si.

A linguagem C++ continua evoluindo de forma a possuir novos requisitos. Atualmente o comitê de padronização do C++ está trabalhando para estender a

linguagem em uma nova especificação, conhecida informalmente por *C++0x*. Esse nome é uma referência ao ano no qual o padrão será lançado, possivelmente em 2009 (Mendes, 2007).

2.3.1 Fundamentos de um ambiente típico de C++

Os programas em C++ passam tipicamente por seis passos, até que possam ser executados. São os seguintes: editar, pré-processar, compilar, “ligar”(link), carregar e executar.

A primeira fase consiste em editar um arquivo em um programa editor, depois esse programa é armazenado em um dispositivo de armazenamento secundário, como um disco. Os nomes dos arquivos em C++ freqüentemente terminam com as extensões .cpp, cxx ou C. Em seguida, o programador executa o comando para compilar o programa. O compilador traduz o programa em C++ para código em linguagem de máquina (também chamado de código objeto). A próxima fase é chamada ligação. Os programas em C++ contêm tipicamente referências a funções definidas em outro lugar, como nas bibliotecas padrão ou nas bibliotecas privadas de grupos de programadores que trabalham em um projeto particular. O código objeto produzido pelo compilador C++ contém tipicamente “buracos” devido a essas partes que estão faltando. Um editor de ligação (linker) liga o código objeto com o código das funções que estão faltando, para produzir uma imagem executável (Deitel, 2001).

A figura 6 ilustra as fases que o programa em C++ passa até ser executado.

2.3.2 Programação em C++

Para entender uma linguagem, nada melhor do que analisá-la. Na figura 7 é mostrado o código de um programa que soma dois números inteiros em C++.

O símbolo // indica que o restante da linha é um comentário. Os comentários ajudam a outras pessoas lerem e entenderem o programa. Os comentários são ignorados pelo compilador e não causam qualquer geração de código objeto, com isso não levam o computador a executar qualquer ação quando o programa é executado. Existe também o comentário de várias linhas. Eles começam com /* e terminam com */.

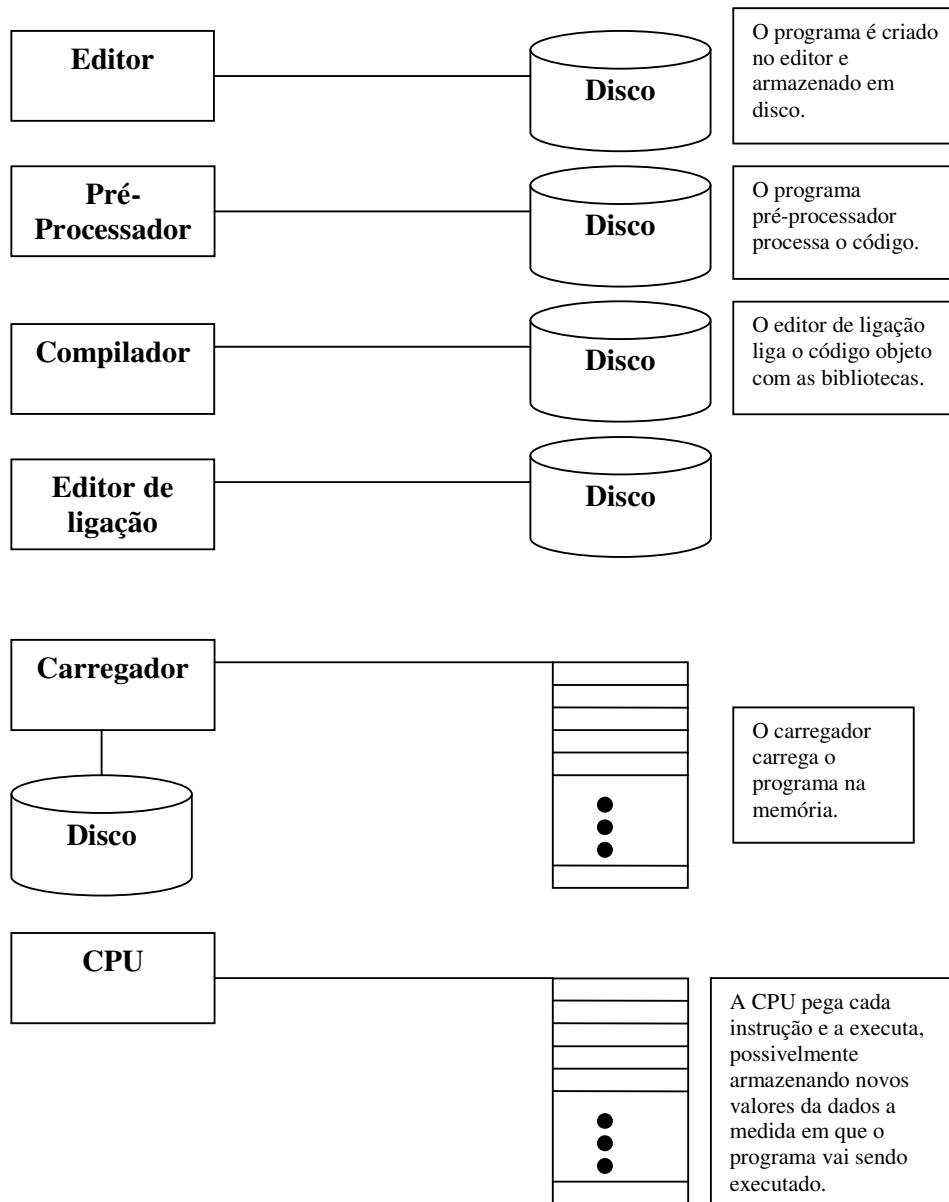


Figura 6: Ambiente típico C++

Fonte: (Deitel, 2001)

```
[*] soma.cpp
/*Programa que soma dois números digitados pelo usuário*/
#include <iostream>
using namespace std;

int main() //Função Principal
{
    int a, b, c; //Declaração dos inteiros a, b e c
    cout <<"Digite o valor de a: "; //Imprime a string na tela de saída
    cin>>a; //lê o inteiro
    cout <<"\nDigite o valor de b: "; //Imprime a string na tela de saída
    cin >> b; //lê o inteiro
    c = a + b; //atribuição da soma
    cout << "\nA soma e: "<< c; //Imprime a soma na tela de saída
    cout<<"\n";

    return 0; //Indica que o programa terminou com sucesso
}
```

Figura 7: Programa que soma dois números em C++

Fonte: Banco de imagens do autor

A linha **# include<iostream>** é uma diretiva do pré-processador, isto é, é uma mensagem para o pré-processador C++. As linhas iniciadas com **#** são processadas pelo pré-processador, antes do programa ser compilado. Essa linha específica diz ao pré-processador para incluir no programa o conteúdo do arquivo de cabeçalho de stream de entrada/saída, **<iostream>**. Este arquivo deve ser incluído em qualquer programa que envia dados de saída para a tela ou recebe dados de entrada do teclado usando o estilo entrada/saída para a tela. O **iostream** é uma biblioteca que armazena funções de entrada e saída de dados. O **cin** e o **cout** utilizam esta biblioteca para fazerem a saída e a entrada de dados respectivamente.

using namespace std; informa ao compilador que o **namespace std** está sendo usado. Os conteúdos **<iostream>** são todos definidos como sendo parte do **namespace std**. O **std** é um dos ambientes de nomes do C++.

A linha **int main** deve constar em todo programa C++. Os parênteses depois de **main** indicam que **main** é um bloco de construção de programa chamado função. Os programas em C++ possuem uma ou mais funções, umas delas é a função **main**, pois os programas em C++ começam a executar na função **main()**. A palavra **int**, à esquerda de **main**, indica que **main** devolve um valor inteiro (Este tópico será explicado em seções posteriores).

O símbolo { (chave) delimita o início do corpo de qualquer função. Da mesma forma, o símbolo } correspondente deve terminar o corpo de cada função.

A linha `int a, b, c;` indica que as variáveis “a”, “b” e “c” estão sendo declaradas. Variável é um espaço de memória reservado para armazenar um certo tipo de dado, isso será explicado com mais detalhes na próxima seção.

O `cout << “Digite o valor de a”;` instrui o computador a imprimir na tela a string contida entre aspas duplas. A linha inteira é chamada de comando. Todo comando deve ter um ponto-e-vírgula no fim (terminador de comando). O `cout` pertence ao “ambiente de nomes” `std`. Ambientes de nomes são um recurso avançado de C++, por isso não serão discutidos aqui, pois vão além do escopo desse trabalho.

O operador `<<` é chamado de operador de inserção `stream`. Quando esse programa for executado, o operando à direita do operador, é inserido no `stream` de saída. Os caracteres do operando à direita são normalmente impressos exatamente como eles aparecem entre as aspas duplas. A barra invertida (`\`) é chamada de caractere de escape. Indica que um caractere especial deve ser enviado para saída. Quando a barra invertida for encontrada em uma string de caracteres, o próximo caractere é combinado com a barra invertida para formar uma seqüência de escape. A tabela 5 mostra esses códigos.

Tabela 5 - Seqüências de escape em C++

Códigos especiais	Significado
<code>\n</code>	Nova linha
<code>\t</code>	Tab
<code>\b</code>	Retrocesso
<code>\f</code>	Salta página de formulários
<code>\a</code>	Beep – Toca o alto falante
<code>\r</code>	CR – Cursor para o início da linha
<code>\\</code>	<code>\</code> - Barra invertida
<code>\0</code>	Null – Zero
<code>\'</code>	Aspas simples
<code>\”</code>	Aspa dupla
<code>\xdd</code>	Representação hexadecimal

Fonte: (Mizrahi, 1994, 7)

O comando `return 0;` no final da função `main`, indica que o programa terminou com sucesso. Esse comando será explicado com mais detalhes nas seções subseqüentes (Deitel, 2001).

2.3.2.1 Variáveis

As variáveis são fundamentais em qualquer linguagem de programação.

Uma variável em C++ é um espaço de memória reservado para armazenar um certo tipo de dado em um determinado tempo e tendo um nome para referenciar o seu conteúdo (Mizrahi, 1994,10).

As declarações de variáveis podem ser colocadas em qualquer lugar dentro de uma função. Porém, a declaração de uma variável deve aparecer antes da mesma ser referenciada (Deitel, 2001).

Uma declaração de variável é uma instrução que serve para reservar um espaço na memória apropriada a fim de armazenar um tipo especificado de dado. Quando se define uma variável em C++, deve-se informar não apenas o nome, mas também o tipo de informação que ela armazenará. No caso da Figura 11, foram declaradas três variáveis do tipo **int** e para usar cada variável criada, basta referenciá-la pelo nome dado a ela (Mizrahi, 1994,10).

O tipo de variável informa a quantidade de memória, em Bytes, que a variável ocupará e a forma como um valor deverá ser armazenado e interpretado.

Em C++ existem cinco tipos básicos de variáveis, são elas:

Tabela 5_ Tipos de variáveis em C++

Tipo	Descrição	Tamanho*	Faixa de valores
Char	Armazena um único caracter	1 byte	Todos os caracteres disponíveis na tabela ASCII
Bool	Valores booleanos podem ser true ou false	1 byte	True ou False
Int	Números inteiros	2 bytes	-2147483648 a 2147483647
Float	Números reais	4 bytes	3.4e +/- 38 (7 dígitos)
Double	Números reais com dupla precisão	8 bytes	1.7e +/- 308 (15 dígitos)

*Os valores da coluna Tamanho dependem da arquitetura do sistema no qual o programa é compilado e executado. Os valores mostrados são aqueles encontrados na maioria dos sistemas de 32 bits.

2.3.2.2 O qualificador const

O tipo **const** associa a um endereço de memória um valor fixo(constante).

Exemplo:

```
const float pi = 3.14;
```

2.3.2.3 Tipos de Operadores

2.3.2.3.1 Operador de atribuição

Em C++ o sinal de igual não tem a mesma interpretação dada em matemática. Representa a atribuição da expressão à sua direita à variável à sua esquerda. Por exemplo:

`x = 5;` o valor 5 é atribuído a x

`y = x = 5;` o valor 5 é atribuído a x primeiro e depois é atribuído a y, então x passa a ter o valor de 5 e y passa a ter o mesmo valor de x.

2.3.2.3.2 Operadores aritméticos

C++ oferece cinco operadores aritméticos binários e um operador aritmético unário. Como pode ser visto na tabela:

Binários

+ Soma
- Subtração
* Multiplicação
/ Divisão
% Módulo

Estes operadores representam as operações aritméticas básicas de soma, subtração, multiplicação, divisão e módulo(resto da divisão inteira).

Unário

Menos unário

O operador menos unário é usado somente para indicar a troca de sinal algébrico do valor.

2.3.2.3.3 Operador de extração

O objeto **cin** manipula toda entrada do teclado por meio do operador de extração (>>) que conecta a entrada de dados à variável que a armazenará.

O comando **cin** faz com que o programa aguarde até que o usuário digite os dados e aperte *enter* para finalizar a entrada.

2.3.2.3.4 Operadores relacionais

Os operadores relacionais fazem comparações entre valores. Como pode ser visto na tabela XXX:

> maior
>= maior ou igual
< menor
<= menor ou igual
== igual
! diferente

Esses operadores são usados em laços e comandos de decisão, que serão vistos mais adiante.

2.3.2.3.5 Operadores lógicos

O C++ tem três tipos de operadores lógicos, são eles:

Binários:

&& “e” lógico
“ou” lógico

Unários

! “não” lógico

Operadores lógicos também fazem comparações. A diferença entre comparações lógicas e relacionais está na forma como os operadores avaliam seus operandos. Operandos de operadores lógicos são avaliados como lógicos (0 ou 1), e não como numéricos.

2.3.2.3.6 Precedência de operadores

A tabela seguinte mostra a precedência de operadores. Os operadores são mostrados de cima para baixo, obedecendo à ordem decrescente de precedência. Todos estes operadores, com exceção do operador de atribuição, =, se associam da esquerda para a direita. A adição é associativa à esquerda, de modo que uma expressão como $x+y+z$ é calculada como se estivesse sido escrita como $(x+y)+z$. O operador de atribuição = se associa da direita para a esquerda, de modo que uma expressão como $x = y = 0$ é resolvida como se tivesse sido escrita como $x = (y=0)$.

Tabela 6_ Precedência de operadores

Operadores	Associatividade	Tipo
()	Esquerda para direita	parênteses
* / %	Esquerda para direita	multiplicativos
+ -	Esquerda para direita	aditivos
<< >>	Esquerda para direita	Inserção em / extração de stream
< <= > >=	Esquerda para direita	relacional
== !=	Esquerda para direita	igualdade
=	Direita para a esquerda	atribuição

Fonte: (Deitel, 2001).

2.3.2.4 Estruturas de controle

Normalmente, os comandos em um programa são executados um depois do outro, na seqüência em que estão escritos. Isto é chamado de execução seqüencial. Eventualmente pode-se mudar a seqüência de execução dos comandos de um programa, para isto usam-se estruturas denominadas “Estruturas de controle”, estas serão melhor definidas nas próximas seções(Deitel, 2001).

2.3.2.4.1 A estrutura de seleção if

É usada para executar uma instrução ou bloco de instruções somente se uma condição for satisfeita.

O corpo de um **if** pode ter uma única instrução terminada por ponto-e-vírgula ou várias instruções entre chaves. Sua forma é:

```
if(condição) conteúdo
```


onde condição é a expressão que está sendo avaliada. Se a condição for verdadeira (true), o conteúdo é executado. Se for falso (false), o conteúdo não é executado e o programa continua na próxima instrução depois da estrutura condicional.

Exemplo:

Suponha que a nota para um aluno passar em uma prova seja 7,0 ou mais. Em pseudocódigo:

Se a nota do aluno for maior que 7,0

Imprima "Aprovado"

Se a nota for maior ou igual a 7,0 será verdadeira, ou seja, **true**. Se a condição é **true**, será impresso "Aprovado" e o próximo comando da sequência é executado. Se a condição for falsa (false), o comando de impressão é ignorado e o próximo comando na sequência é executado.

O comando em pseudocódigo pode ser escrito em C++ como:

```
if(nota >= 7,0)
```

```
    cout<<"Aprovado";
```

A Figura 8 representa este código.

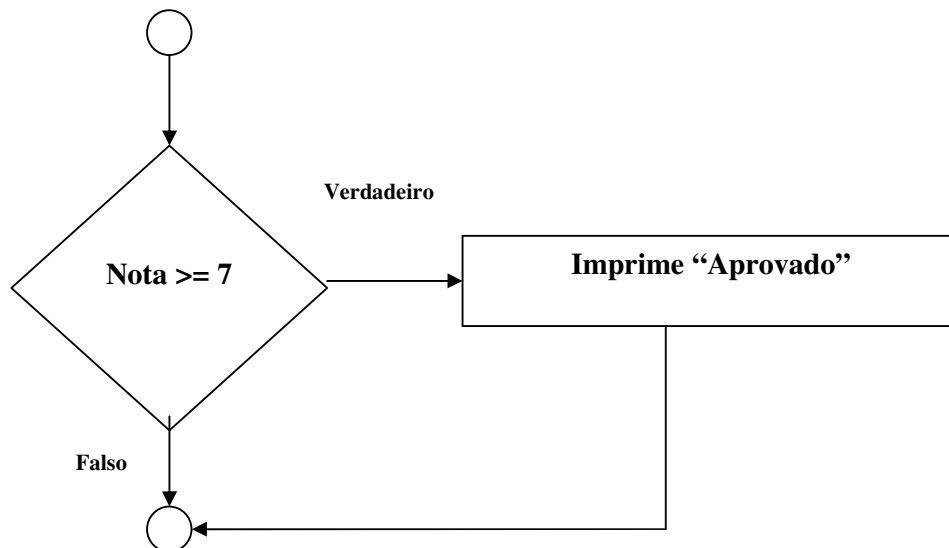


Figura 8: Representando em fluxograma a estrutura de seleção if

Fonte: (Deitel, 2001)

2.3.2.4.2 A estrutura de seleção if/else

A estrutura de seleção **if** executa uma ação indicada só quando a condição é verdadeira (**true**); caso contrário, a ação é saltada. A estrutura **if/else** permite ao programador especificar que uma ação deve ser executada quando a condição é verdade e uma ação diferente quando a condição é falsa(**false**). Por exemplo, o comando de pseudo código:

Se a nota de aluno é maior ou igual a 7,0

Imprime "Aprovado"

Senão

Imprime "Reprovado"

O comando em pseudocódigo pode ser escrito em C++ como:

```
if(nota >= 7,0)
```

```
    cout<<"Aprovado";
```

```
else
```

```
    cout<<"Reprovado";
```

Se a nota do aluno for igual ou maior que 7,0, ou seja, verdade, será impresso "**Aprovado**", caso contrário, o comando do **else** será executado, no caso imprime "**Reprovado**". A Figura 9 representa esse código.

O corpo de um **else**, como o do **if**, pode ter uma única instrução terminada por ponto-e-vírgula ou várias instruções entre chaves.

2.3.2.5 A Estrutura de repetição while

Uma estrutura de repetição que permite ao programador especificar que uma ação deve ser repetida enquanto alguma condição for verdadeira, onde condição é a expressão que está sendo avaliada.

O corpo de um **while** pode ter uma única instrução terminada por ponto-e-vírgula ou várias instruções entre chaves. Sua forma é:

```
while(condição) conteúdo
```

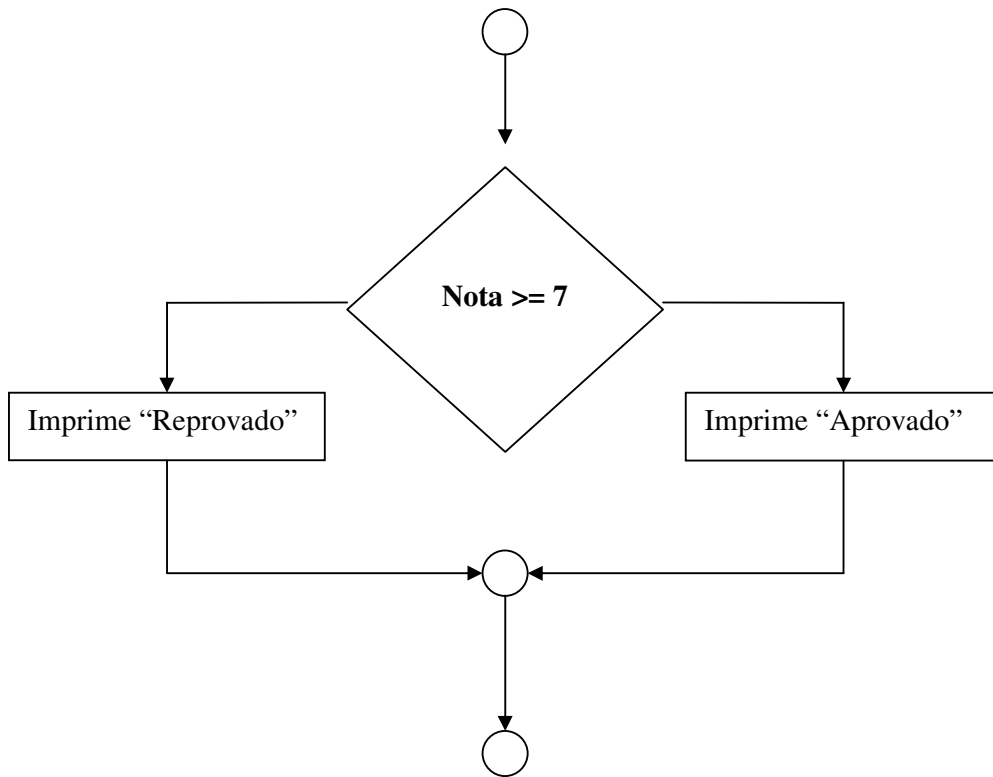


Figura 9: Representando em fluxograma a estrutura de seleção if/else

Fonte: (Deitel, 2001)

Se a condição for verdadeira (**true**), o conteúdo é executado e a condição volta a ser testada e assim vai se repetindo até que a condição seja falsa (**false**), então o conteúdo não é executado e o programa continua na próxima instrução depois da estrutura de repetição.

Exemplo:

*O valor de **a** é zero*

*Enquanto **a** for menor do que três*

*Adicione 1 a **a***

Repete-se primeira a instrução enquanto a condição “*Enquanto **a** for menor do que três*” for verdadeira. Se for verdadeira, a ação “*adicione 1 a **a***” é executada. Os comandos a serem executados, constituem o corpo do **while**. O corpo (conteúdo) pode ser um comando único ou um comando composto. Em algum momento, a condição se tornará falsa (quando o **a** for igual a três). Neste momento, a repetição termina e o primeiro comando de pseudocódigo após a estrutura de repetição é executado.

O comando em pseudocódigo pode ser escrito em C++ como:

```
a=0;
while(a<3)
  a=a+1;
```

A Figura 10 representa esse código.

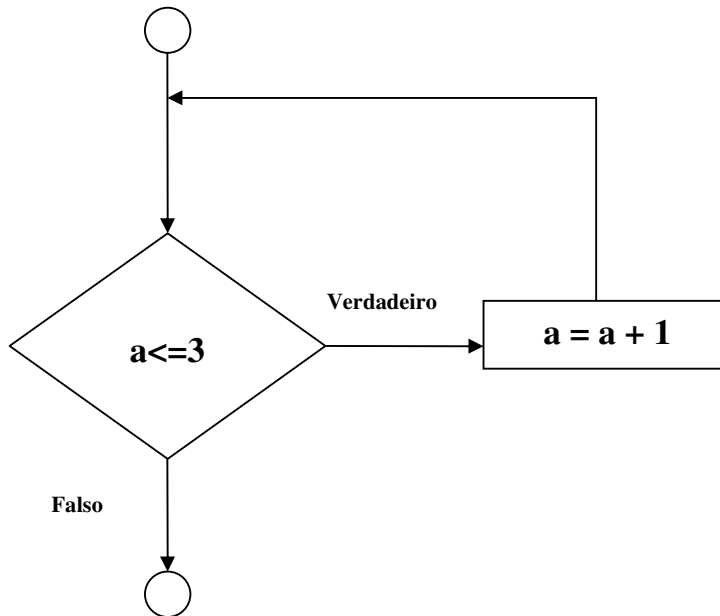


Figura 10: Representando em fluxograma a estrutura de repetição while

Fonte: (Deitel, 2001)

2.3.2.6 O Comando break

O comando **break** causa a saída imediata do laço, o controle passa para a próxima instrução após o laço.

O código da figura 11 mostra um exemplo do uso do **break** e de outras estruturas comentadas anteriormente.

Nesse programa, são criadas duas variáveis, são elas **a**; e **b** com o valor um. Na próxima linha o while verifica se a condição **b <= 10** é verdadeira, caso positivo, o laço é executado. Dentro do laço existe um **cin** que espera a entrada de dados por parte do usuário. Feito isso, o dado digitado pelo usuário fica armazenado na variável **a**, logo depois o comando de decisão **if** verifica se o conteúdo da variável **a** é igual a oito, caso seja verdade, o **break** é executado e o controle do programa sai do laço indo direto para a próxima instrução, no caso, o **return**. Caso a condição do **if** seja falsa, o controle passa para o **else** e o seu corpo é executado, no caso, imprime a string “**Numero nao aceito!**”, logo depois a variável **b** é incrementada em um, e o controle volta para a instrução **while** para testar a condição, refazendo o ciclo.

```
[*] oi.cpp
// Programa que pede para o usuário digitar até dez número entre um a dez.
// Caso o usuário digite o número oito, o programa termina antes sua execução.

# include <iostream>
using namespace std;
void main()
{
    int a; // Variável declarada
    int b=1; // Variável declarada e inicializada
    while(b <= 10) //Enquanto b menor ou igual a dez
    {
        cout << "\n Digite um numero entre um e dez \t";
        cin >> a; // Entrada do teclado
        if(a==8.3) // Se a igual a 8, o dorpo do if é executado
        {
            break; // Se condição verdadeira o break é executado
        }
        else // Se a diferente de 8, o corpo else é executado
        {
            cout << "\n Numero nao aceito!\n";
        }
        b=b+1;
    }
}
```

Figura 11: O uso do break

Fonte: Banco de imagens do autor

2.3.2.7 Arrays

Um array é uma estrutura em C++ usado para armazenar uma coleção de variáveis de mesmo tipo. Essa estrutura na verdade é um grupo de posições de memória consecutivas, todas de mesmo nome e mesmo tipo. Para fazer referência a uma posição particular ou elemento no array, deve-se especificar o nome do array e o número da posição daquele elemento particular no array, entre colchetes ([]).

A Figura 12 mostra um array de valores inteiros chamado **a**. Esse array possui 11 elementos. Cada elemento pode ser referenciado com o nome do array seguido da posição entre colchetes, chamado de subscrito. O primeiro subscrito em qualquer array C++ é zero. Assim, o sexto elemento no array representado na figura é o **a[5]** que armazena o valor **95** (Deitel, 2001).

2.3.2.7.1 Declaração de Arrays

Os arrays, como qualquer variável, precisam ser declarados. A diferença é que os arrays ocupam mais espaço de memória e, além do tipo, o programador tem que especificar o número de elementos exigidos no array, de forma que o compilador possa reservar a quantidade apropriada de memória. A declaração do array da figura 16 pode ser feita da seguinte forma:

```
int a[11];
```

ou já inicializando seus elementos explicitamente da seguinte forma:

```
int a [11] = { 3,45,-5,0,9,95,74,3212,10,8,153 };
```

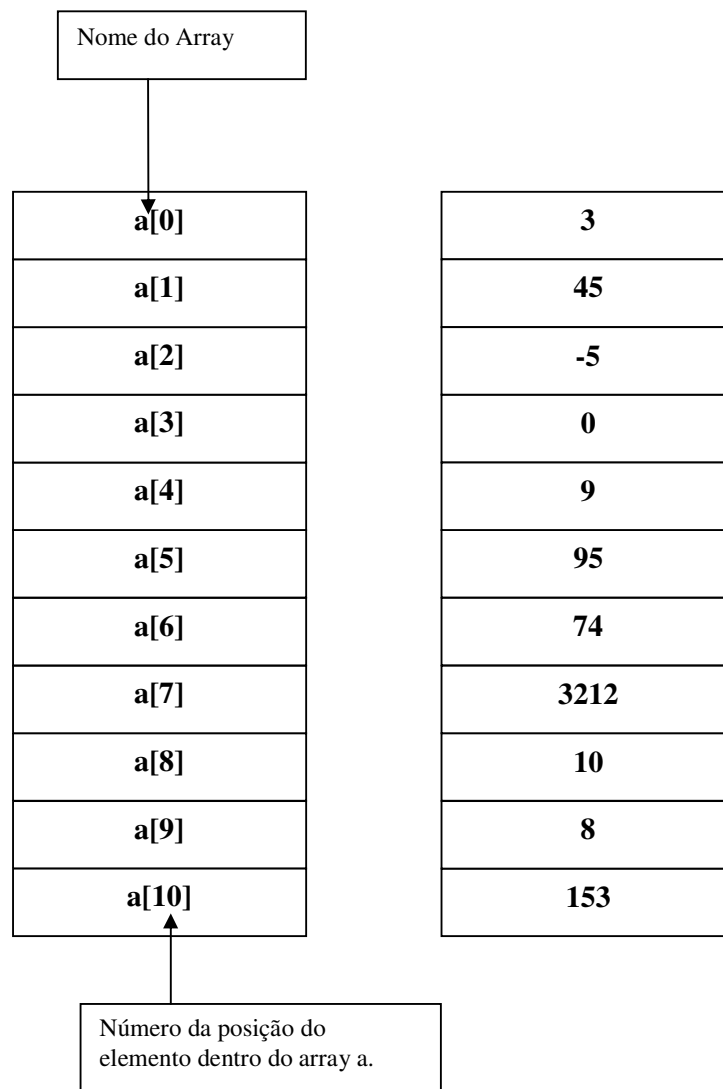


Figura 12: Um array com 11 elementos

Fonte: (Deitel, 2001)

Os elementos do array são guardados em uma seqüência contínua de memória, isto é, um seguido do outro, o que não ocorre quando são criados variáveis separadas. (Deitel, 2001).

2.3.2.8 Funções

Uma função é um conjunto de instruções criadas com o intuito de cumprir uma tarefa particular, agrupadas em uma unidade com nome para referenciá-la.

A principal razão para usar funções é a de dividir a tarefa original em pequenas tarefas que simplificam, organizam e reduzem o tamanho do programa, já que um trecho do código pode ser reaproveitado diversas vezes.

Um programa em C++ pode conter inúmeras funções, das quais uma delas é **main()**. A execução do programa sempre começa em **main**, e sempre que o controle do programa encontra uma instrução que inclui o nome de uma função, a função é chamada. Ao chamar uma função o controle é desviado da seqüência do código da função e passa para a função chamada que executa suas instruções e depois volta o controle à instrução seguinte à chamada da função chamada (Mizrahi1994, p.117).

2.3.2.8.1 Tipos de função

O tipo de uma função é definido pelo tipo de valor que ela retorna por meio do comando **return**.

Os tipos de funções existentes em C++ são os mesmos tipos das variáveis, exceto quando a função não retorna nada. Neste caso, ela é do tipo **void**.

Uma função é dita do tipo **int**, quando ela retorna um valor do tipo **int**. (Mizrahi1994,123).

2.3.2.8.2 O Comando return

O comando **return** termina a execução de uma função retornando o controle para a instrução seguinte à do código de chamada.

A expressão ou valor após a palavra **return** é retornado para a função que a chama. Este valor é convertido para o tipo da função, especificado no seu protótipo. Apenas um único valor pode ser retornado para a função que a chama.

Uma função declarada como **void** não terá o **return**. Quando isso acontecer, a função irá terminar quando a chave que indica o final do corpo da função for atingida (Mizrahi1994, p.123).

2.3.2.8.3 Parâmetros da função

As informações transmitidas a uma função são chamadas parâmetros.

A função deve declarar essas informações entre parênteses, no cabeçalho de sua definição.

Os parâmetros podem ser utilizados livremente no corpo da função (Mizrahi1994,125).

2.3.2.8.4 Passagem por valor

Na passagem por valor uma cópia do valor do argumento é passado para a função. Neste caso a função que recebe este valor ao fazer modificações no parâmetro não estará alterando o valor original que somente existe na função que chamou. Quando a função termina sua execução, os parâmetros são destruídos a menos que sejam static (UFRJ, 2007)

2.3.2.8.5 Exemplo de uma função em C++

A Figura 13 mostra um exemplo de função em C++. A execução programa inicia na função **main()**, que tem como primeira instrução a declaração de três variáveis. A próxima instrução é a impressão da string **“Insira o primeiro valor\t”**, logo depois, o programa irá esperar que o usuário digite um valor e aperte **“Enter”**. Isso irá ocorrer mais uma vez, então a função **soma** será chamada, e dois argumentos serão passados. A função irá criar dois parâmetros para receber os argumentos e na sua primeira instrução irá criar uma variável para receber a soma dos dois parâmetros recebidos (Esse tipo de passagem é conhecido como passagem por valor). Logo depois, o valor da variável **c** será retornado através do comando **return**, nesse momento os parâmetros e a variável **c** serão destruídas.

```
[*] soma.cpp
/* Programa que soma dois números digitados pelo usuário
   usando funções */

# include <iostream>
using namespace std;

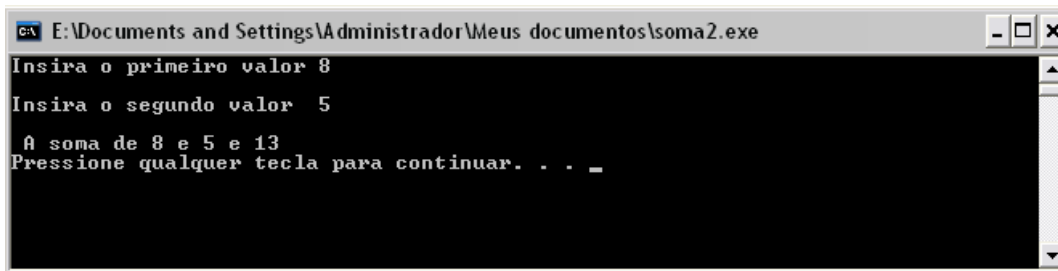
int soma(int a, int b) // Função soma com dois parâmetros declarados
{
    int c; // Variável declarada
    c = a + b;
    return c; // O valor de c é retornado a função que chamou a função soma
}

void main()
{
    int a,d,e; // Variável declarada
    cout<<"Insira o primeiro valor\t";
    cin >> a;
    cout<<"\nInsira o segundo valor\t";
    cin >> d;
    // Chamando a função soma e atribuindo o seu resultado à variável e.
    e = soma(a,d);
    cout <<"\n A soma de " << a << " e " << d << " e " << e << "\n";
}
```

Figura 13: Exemplo de função
Fonte: Banco de imagens do autor

O controle, então, retornará para a linha onde a função foi chamada e atribuirá o valor retornado pela função a variável `e`. A próxima instrução será a impressão dos valores das variáveis `a` e `d`, digitados pelo usuário, e da variável `e` retornado pela função `soma`.

O programa da figura 13 executado ficará da seguinte como a figura 14:



```
E:\Documents and Settings\Administrador\Meus documentos\soma2.exe
Insira o primeiro valor 8
Insira o segundo valor 5
A soma de 8 e 5 e 13
Pressione qualquer tecla para continuar. . . _
```

Figura 14: Código executado

Fonte: Banco de imagens do autor

2.3.2.9 Escopo de variáveis

As variáveis podem ser declaradas em três lugares em um programa; dentro das funções (variáveis locais), fora das funções (variáveis globais) e na lista de parâmetros das funções (parâmetros formais) (UFRJ, 2007).

2.3.2.9.1 Variáveis Locais

São aquelas declaradas dentro de uma função. Elas passam a existir logo que são declaradas no bloco de comandos ou funções e são destruídas ao final da execução do bloco. Uma variável local só pode ser referenciada dentro dos blocos onde foi declarada (UFRJ, 2007).

2.3.2.9.2 Variáveis Globais

São definidas fora de qualquer função e são disponíveis a qualquer função.

2.3.2.9.3 Parâmetros Formais

São variáveis criadas no início da execução de uma função e destruídas no final. Elas recebem valores das funções que as chamou. Portanto, permitem que uma função passe valores para outra. Dentro das funções a que pertencem, elas podem ser modificadas e manipuladas sem nenhuma restrição (UFRJ, 2007).

2.4. Ambiente de Simulação de Máquinas Virtuais (ASIMAV, 1994)

O ASIMAV foi desenvolvido por Evandro de Barros Costa e o Professor Dr. Edmundo Tojal Donato Junior, do Departamento de Informática e Matemática Aplicada de Universidade Federal de Alagoas; por Robério José Rogério dos Santos, do Centro de Processamento de Dados da Fundação de Amparo à Pesquisa do Estado de Alagoas; e por Walfredo da Costa Cirne Filho, do Parque Tecnológico da Paraíba.

É um ambiente computacional que visa prover os estudantes com ferramentas que objetivam favorecer as atividades de aprendizagem e promover o interesse dos mesmos em temas como Arquitetura e Organização de Computadores, Construção de Compiladores e Linguagens de Programação. Sendo assim, o principal objetivo do ASIMAV é atacar a falta de motivação observada nos estudantes dos cursos de informática, quando estes estão envolvidos com temas que fundamentam os sistemas de computação. Uma alternativa para amenização do problema citado é colocar disponível aos estudantes ferramentas de software que sirvam para aproximar o abstrato do concreto. A proposta de ASIMAV representa uma contribuição nesta direção.

A idéia de construir um simulador do funcionamento interno de um computador objetiva justamente aproximar a teoria da prática, fornecendo ferramentas para os alunos "experimentarem" os conceitos que são apresentados em sala de aula.

Apesar da boa proposta, esta ferramenta e suas imagens não estão disponíveis.

2.5. Ambiente de Aprendizado de Programação (AMBAP)

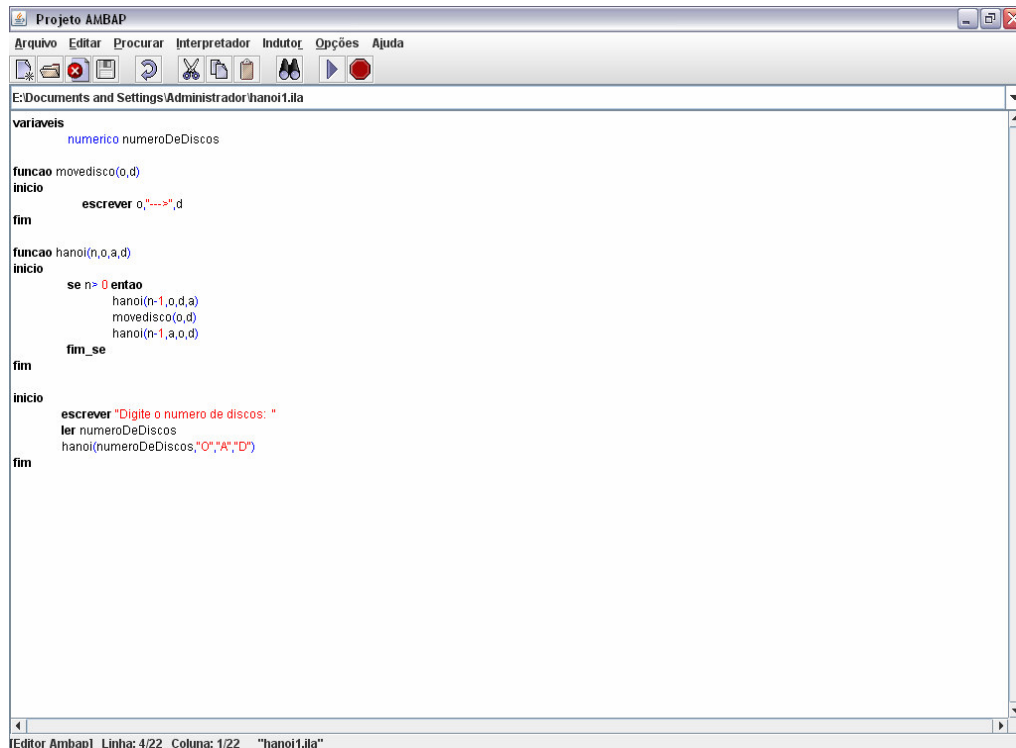
O projeto está sendo desenvolvido pelo Centro de Ciências Exatas e Naturais do Departamento de Tecnologia da Informação da Universidade Federal de Alagoas. A professora Dra. Eliana Silva de Almeida (UFAL), orienta o trabalho dos executores André Atanásio Maranhão Almeida (UFAL), Klebson dos Santos Silva (UFAL) e Rodrigo de Barros Paes (UFAL) com a colaboração dos Professores Dr. Evandro de Barros Costa (UFAL) e Dr. Sergio Crespo Coelho da Silva Pinto (Universidade do Vale do Rio dos Sinos - UNISINOS).

Tendo como base o ASIMAV, o AMBAP é um projeto de pesquisa voltado para desenvolvimento de um ambiente computacional de aprendizagem orientado a

dois propósitos básicos, que são: o ensino de programação para iniciantes e o entendimento do funcionamento de um computador.

Este projeto desenvolveu um ambiente para facilitar o aprendizado de lógica de programação, no que diz respeito a construção de algoritmos, que permita minimizar o esforço do estudante em aplicar corretamente os conceitos e técnicas de programação, através de um processo de simulação. Isto é, este ambiente irá auxiliar o estudante na compreensão de todas as etapas de desenvolvimento de um programa, independente da linguagem de programação utilizada, contando com recursos de indução. Entre estas etapas, podemos incluir a definição da seqüência de ações para solucionar problemas e a visualização de como esta seqüência de ações atua quando o elemento solucionador é o computador (UFAL, 2007).

A figura 15 mostra o código inserido pelo usuário em uma linguagem própria do AMBAP.



```
Projeto AMBAP
Arquivo  Editar  Procurar  Interpretador  Indutor  Opções  Ajuda
E:\Documents and Settings\Administrador\hanoi1.ila
variaveis
    numerico numeroDeDiscos
funcao movedisco(o,d)
inicio
    escrever o,"--->",d
fim
funcao hanoi(n,o,a,d)
inicio
    se n> 0 entao
        hanoi(n-1,o,d,a)
        movedisco(o,d)
        hanoi(n-1,a,o,d)
    fim_se
fim
inicio
    escrever "Digite o numero de discos: "
    ler numeroDeDiscos
    hanoi(numeroDeDiscos,"O","A","D")
fim
[Editor Ambap] Linha: 4/22  Coluna: 1/22  "hanoi1.ila"
```

Figura 15: Código da Torre de Hanói no AMBAP

Fonte: Banco de imagens do autor

Na Figura 16 o código é executado sem erros e o resultado é apresentado.

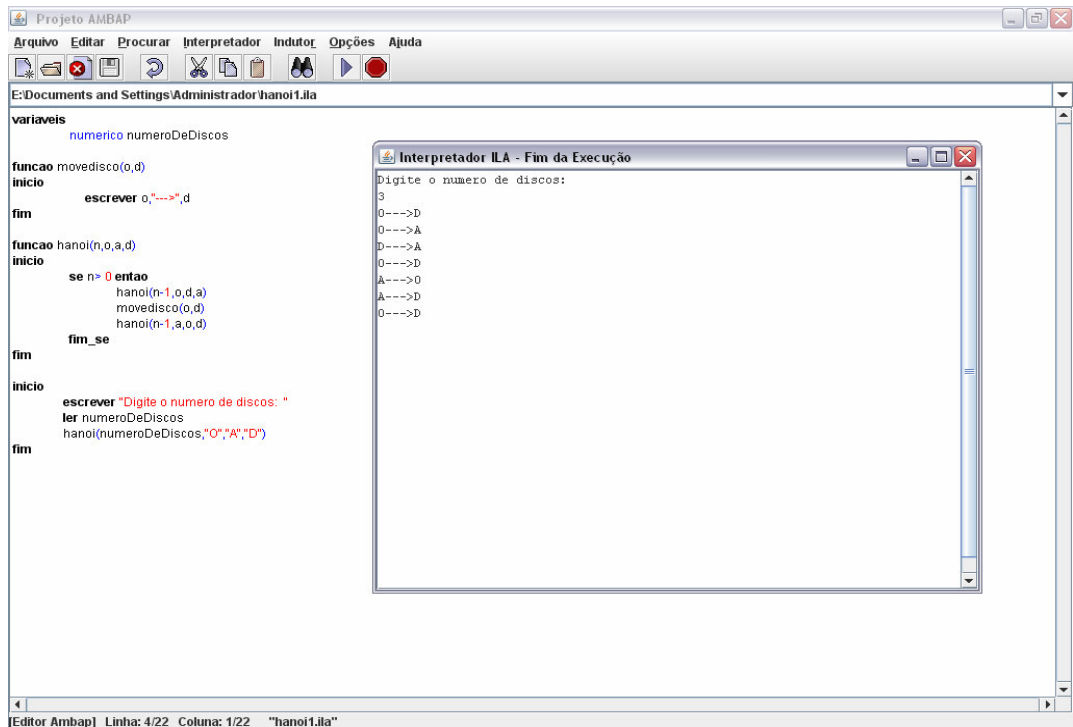


Figura 16: Execução do código da Torre de Hanói

Fonte: Banco de imagens do autor

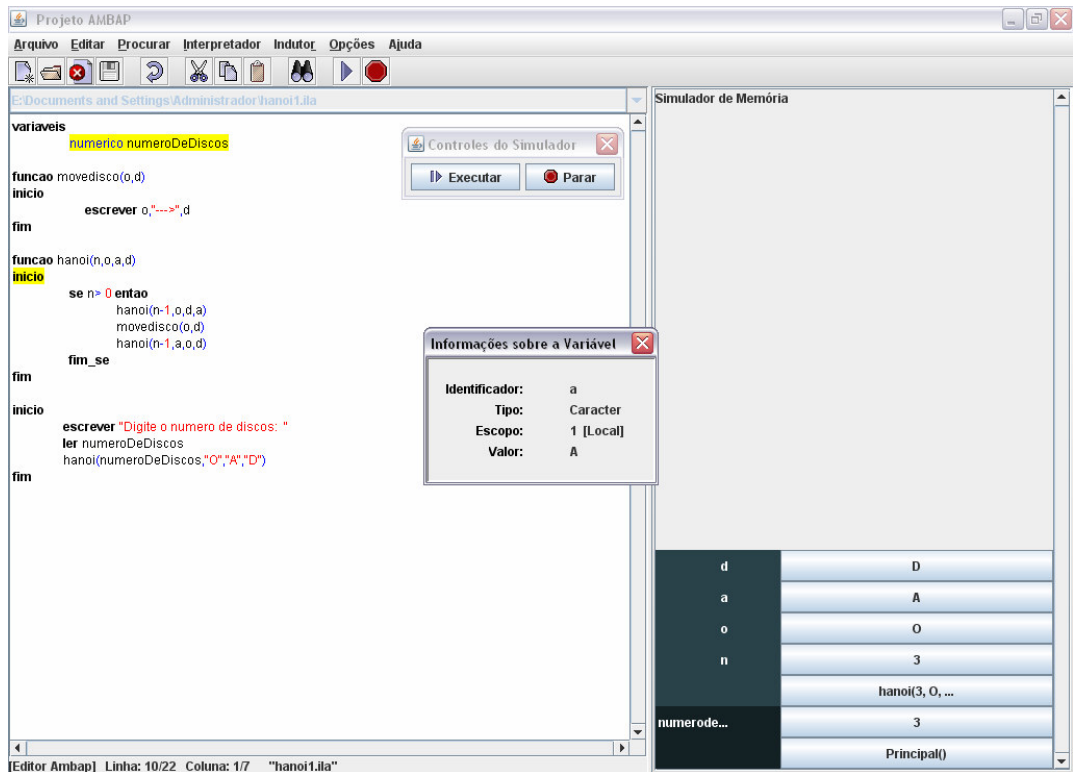


Figura 17: Execução do código da Torre de Hanói passo a passo

Fonte: Banco de imagens do autor

Na figura 17 o código é executado passo a passo e uma simulação da memória é apresentada além do resultado do programa.

3. Ambiente Virtual de Linguagem de Programação (AVLP)

O AVLP é um ambiente computacional que simula o funcionamento interno do computador nos processos de compilação e execução do código digitado pelo usuário.

A linguagem escolhida para simulação foi o C++ por ser uma linguagem de programação de alto nível com facilidades para o uso em baixo nível, multiparadigma, de uso geral, muito popular e muito usada em faculdades e universidades no ensino de linguagem de programação.

O programa aborda os conceitos básicos da linguagem C++, como os operadores, os laços, os comandos de decisão, funções e os arrays. Por ser uma linguagem com muitos detalhes não foi possível, no momento, abordar todas as características do C++, por isso, ficarão para trabalhos futuros as estruturas mais complexas como a orientação a objeto, ponteiros, entre outras.

3.1 A Análise Léxica do AVLP

Inicialmente, foi feito o levantamento da gramática restringida do C++ e logo depois foram definidos os tokens válidos para a análise léxica, como as palavras reservadas, os identificadores, as constantes e os operadores. Durante o processo de análise léxica, são desprezados caracteres não significativos como espaços em branco. Para isso foi criada, no AVLP, a classe **AnaliseLexica** que recebe todo o texto digitado e verifica se cada token pertence ou não à linguagem. Para construção dessa classe foi necessária a criação de um autômato finito apresentado na Figura 18.

Além de reconhecer os símbolos léxicos, a classe **AnaliseLexica** também realiza outras funções, como a de armazenar alguns desses símbolos em tabelas internas e indicar a ocorrência de erros léxicos. Para fazer o papel das tabelas internas, foram criadas duas classes: A classe **Registro** que guarda o nome, o tipo, a linha e a coluna do símbolo reconhecido. A outra classe é a **TabelaDeSimbolos** que foi criada para armazenar os registros; ela possui um vetor que armazena todos os registros que serão usados durante as outras fases da compilação. Já os símbolos não reconhecidos são armazenados em um **ArrayList** para ser enviado à classe **Interface** que os imprimem na tela. A seqüência de tokens reconhecida pela classe **AnaliseLexica** é utilizada como entrada pelo módulo seguinte do simulador

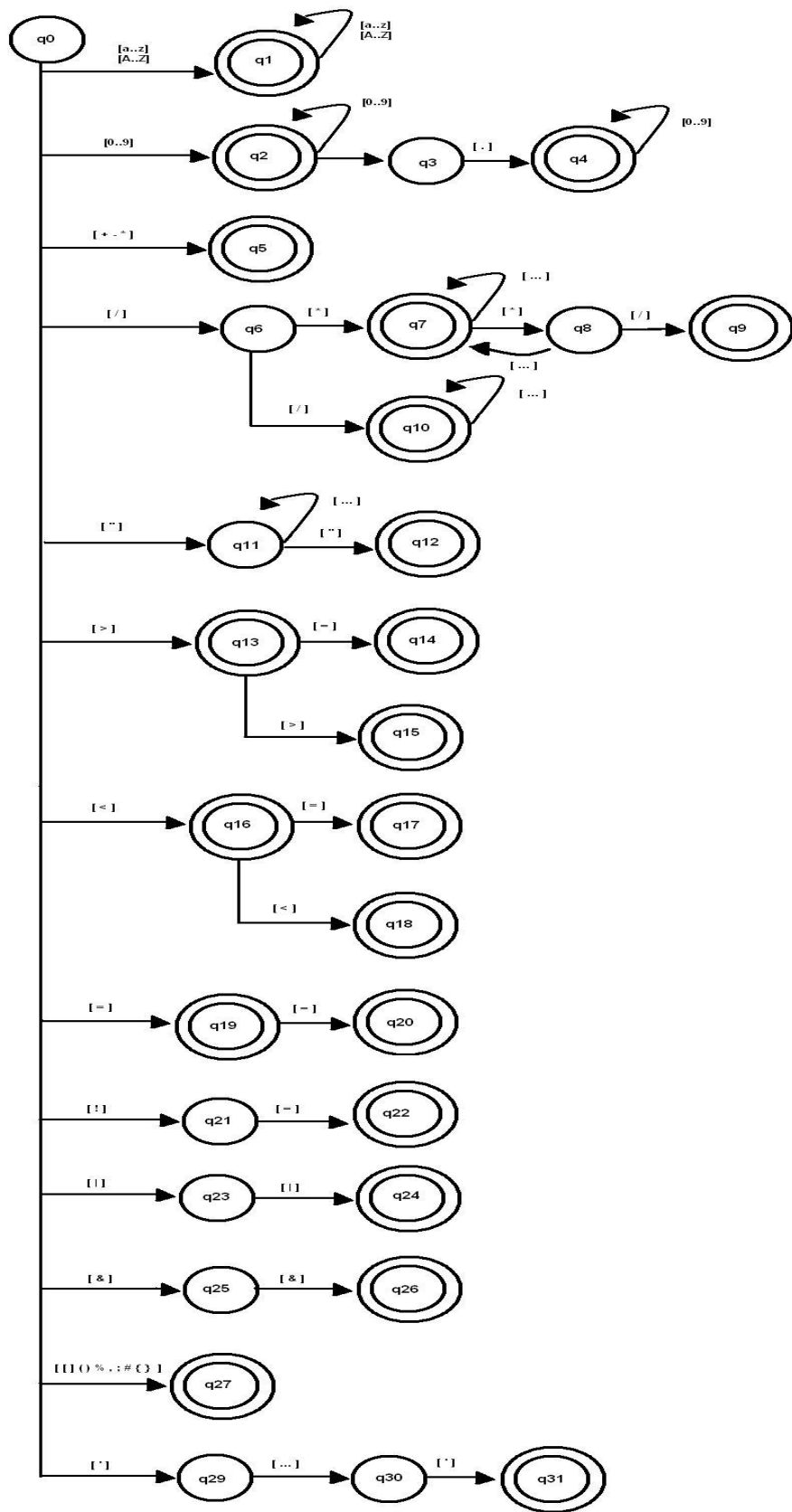


Figura 18: Autômato finito do analisador léxico

Fonte: Banco de imagens do autor

que é o analisador sintático, que no AVL P é interpretada pela classe **AnaliseSintatica**.

A figura 19 é um trecho do código da classe AnaliseLexica. As variáveis nunLinha e linhaInicio armazenam o número da linha atual analisada e a linha inicial do código, já o array reservada armazena todas as palavras reservadas do compilador e a variável registros armazena o token, o tipo do token, a linha e a coluna e a variável tabelaDeSimbolos armazena todos os registros.

```
import java.util.*;
import javax.swing.JOptionPane;
public class AnaliseLexica
{
    // Variáveis privadas
    private int numLinha, linhaInicio;
    private int coluna, colunaDeErro, colunaInicio;
    private String texto;
    private String token, token2;
    private int estado;
    private String concatena;
    private TabelaDeSimbolos registros;
    // Vector<String> v;
    private ArrayList<String[]>relatorioErros;
    String campos[] = new String[3];
    //Palavras Reservadas
    private String reservada[]={ "include", "iostream", "using", "namespace", "std",
        "int", "float", "double", "bool", "string", "char", "void", "return", "cin", "cout",
        "if", "else", "while", "break", "const", "static", "true", "false"};

    //Variáveis públicas
    public char caracter;
    public String concatenaAnalise;

    //Construtor da Classe AnaliseLexica()
    public AnaliseLexica()
    {
        this.linhaInicio=1;
        this.numLinha=1;
        this.coluna=1;
        this.colunaInicio=1;
        this.estado=0;
        this.texto="";
        this.token="";
        this.concatena="";
        registros = new TabelaDeSimbolos();
        relatorioErros = new ArrayList<String[]>();
    }
}
```

Figura 19: Trecho do código da classe AnaliseLexica

Fonte: Banco de imagens do autor

3.2 A Análise Sintática do AVL P

A classe **AnaliseSintatica** verifica se as construções usadas no programa estão gramaticalmente corretas. Ela aplica, no texto fonte, as regras gramaticais da linguagem. Além disso, ela identifica os erros sintáticos e armazena em um *ArrayList*, um array de strings contendo a linha, a coluna e o tipo de erro identificado. Para construção da classe **AnaliseSintatica**, foi usada a análise preditiva tabular com a recuperação de erro local. Isso gerou a tabela de análise preditiva da gramática restringida do C++, esse tipo de análise, bem como o tipo de recuperação

de erros e a tabela gerada foram explicados no capítulo 2. Nessa classe foram embutidas esquemas de tradução, que são ações semânticas associadas às regras de produção da gramática de modo que, quando uma dada produção é processada, essas ações são executadas.

A figura 20 mostra um trecho da classe AnaliseSintatica, a variável tabelaDeSimbolos, que serve como uma base de dados, tem armazenado todos os tokens identificados na classe AnaliseLexica. Token por token da tabelaDeSimbolos é analisada sintaticamente.

```
import java.util.Stack;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.table.*;
import java.util.*;
import java.io.*;
import javax.swing.JOptionPane;
public class AnaliseSintatica
{
    private Stack<String> pilha;

    private TabelaDeSimbolos tabelaDeSimbolos;

    private int posicaoLeitura;

    private String acao, relatorioFinal;

    private boolean pula;

    private ArrayList<String[]> relatorioErros;

    String campos[] = new String[3];

    Registro registro;

    // Construtor da classe;
public AnaliseSintatica()
{
    pilha = new Stack<String>();
    relatorioFinal="";
    this.pula=false;
    relatorioErros = new ArrayList<String[]>();
}
}
```

Figura 20: Trecho do código da classe AnaliseSintatica

Fonte: Banco de imagens do autor

3.2.1 A Análise Semântica do AVLPL

O próximo passo do compilador é a análise semântica. No AVLPL ela é feita pela classe **AnalizadorSemantico**, que verifica se as estruturas sintáticas analisadas fazem sentido. Associada a essa classe está a classe **Funcao**, que verifica cada função em separado, armazenando o nome da função e os seus parâmetros. Ela também armazena os identificadores e verifica se as atribuições feitas aos identificadores são compatíveis, se o identificador foi declarado mais de

uma vez e etc. Também associadas à classe **AnalisadorSemantico**, estão as classe **RegistroFuncao**, que armazena o nome, a localização, o final e os parâmetros da função e a classe **TabelaFuncoes** que armazena vários objetos da classe **RegistroFuncao** em um **ArrayList**. Essas classes são usadas na execução do programa, para localizar uma função e a partir daí executá-la.

A figura 21 é um trecho de código da classe **AnalisadorSemantico**, como ocorre na classe **AnaliseSintatica**, token por token que está armazenado em **tabelaDeSimbolos** é analisado léxicamente.

```
import java.util.ArrayList;
import javax.swing.*;
import java.util.*;
public class AnalisadorSemantico {
    private ArrayList <Identificador> parametros;
    private int posicaoLeitura;
    private boolean using;
    protected int principal;
    public ArrayList <Funcao> funcoes;
    public ArrayList <Identificador> identificador;
    protected TabelaDeSimbolos tabelaSimbolos;
    private String erro;
    private TabelaFuncoes TabFuncoes;
    private ArrayList<String[]> relatorioErros;
    String campos[] = new String[3];
    // Construtor da classe
    public AnalisadorSemantico()
    {
        this.using=false;
        this.principal=0;
        this.posicaoLeitura=0;
        funcoes = new ArrayList<Funcao>();
        identificador = new ArrayList<Identificador>();
        this.erro = "";
        TabFuncoes = new TabelaFuncoes();
        relatorioErros = new ArrayList<String[]>();
    }
}
```

Figura 21: Trecho do código da classe **AnalisadorSemantico**

Fonte: Banco de imagens do autor

3.2.2 Tratamento de erros no AVL

Quando um erro é identificado, o AVL descarta o erro e volta a analisar o código. Cada erro encontrado, é armazenado em um Array que existe em cada classe de análise e logo depois que todo o código é conferido, se houver(em)

erro(s), os erros são enviados para a classe Interface que os exibe na tela informando o usuário o tipo de erro que ele encontrou e onde encontrou. Uma exibição de erros pode ser vista na Figura 22.

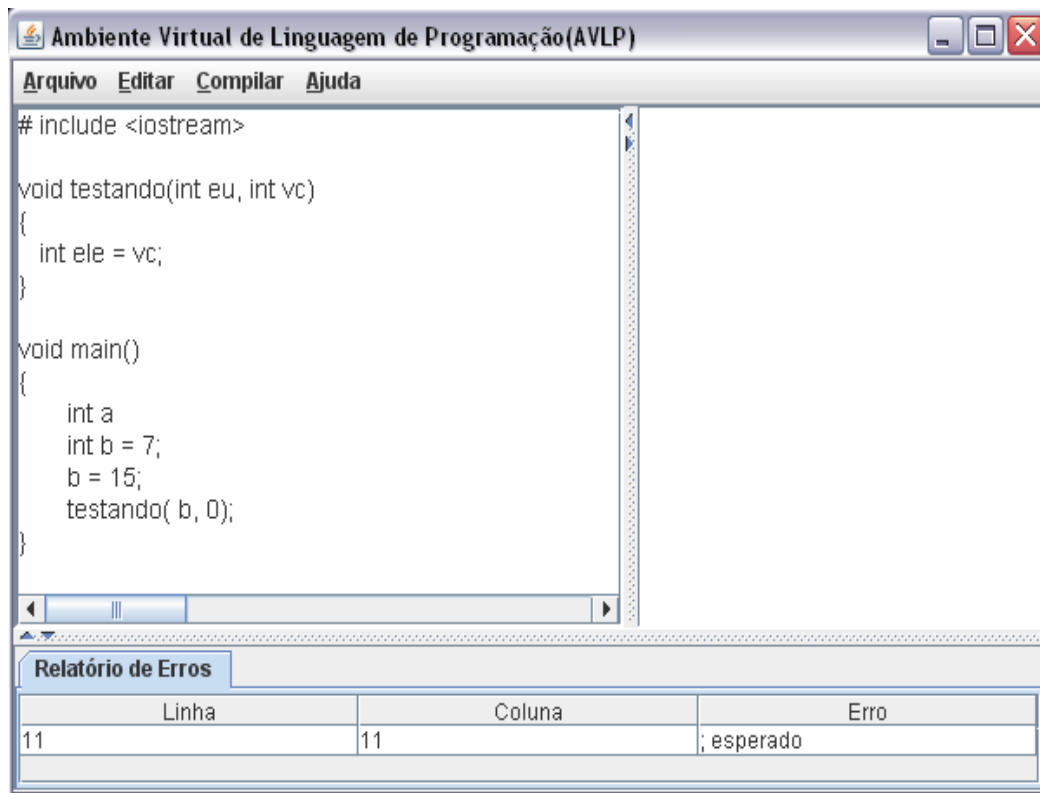


Figura 22: Exibição de erros no AVLP

Fonte: Banco de imagens do autor

3.2.3 A Execução do AVLP

Se o texto digitado passar pela análise léxica, sintática e semântica sem erros, é porque ele está apto a ser executado. Essa "execução" (simulação) é feita pela classe **Interface**, que localiza (se houver) a função "main" nos registros da classe **TabelaFuncao** e da início à execução. A classe ainda conta com o auxílio das classes **RegistroValoresId** que armazena os identificadores, seu tipo e seus respectivos valores; e a classe **TabelaDeValores** que armazena em um ArrayList todos registro de valores que vão sendo consultados durante toda a execução.

A execução é feita passo a passo através de um delay com intervalo de 2 segundos entre uma instrução e outra. E durante a execução, para cada função declarada, é criado na interface um *frame* com o nome da função, esse *frame* é criado na classe **EstruturaDesenho**. Dentro desse *frame* é feita uma animação que contém um bloco maior representando a função, um bloco menor que representa os

laços ou comandos de decisão e um simulador de memória para representar os parâmetros e identificadores. Isso pode ser visto na Figura 23.

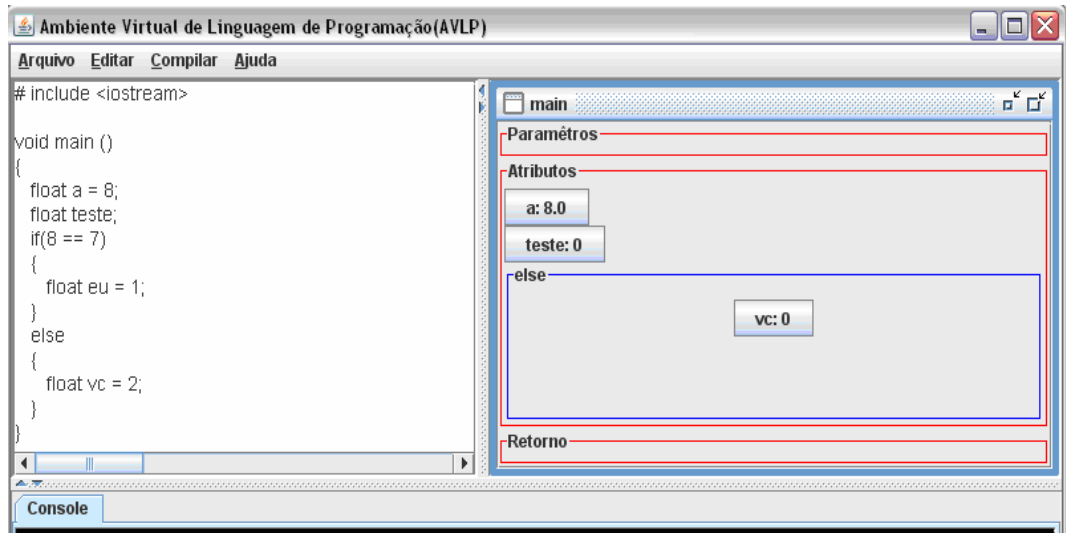


Figura 23: AVLP executando um código em C++

Fonte: Banco de imagens do autor

Para cada função que é chamada, um *frame* é criado. A princípio é criado apenas com o bloco maior seguido dos parâmetros. À medida que os identificadores vão sendo declarados, vão sendo adicionados a esse bloco os botões para representá-los. O mesmo acontece com os laços e comandos de decisão, lembrando que, como ocorre nos compiladores, quando um laço é finalizado, o bloco menor que o representa é removido do *frame*; o mesmo acontece com as variáveis declaradas dentro dele.

Quando o valor de um identificador é modificado, o valor no botão que o representa também é modificado, sendo assim o AVLP simula o que acontece na memória de um computador quando um programa é executado. Isso pode ser visto nas Figuras 24 e 25.

A Figura 24 mostra o exato momento em que a variável **b** é criada e recebe o valor **7**. Já a Figura 25 mostra o momento em que a instrução posterior à executada na Figura 24 é executada, no caso **b = 15**;. Essa instrução ordena que no espaço de memória reservado para a variável **b**, seja colocado o valor **15** e o valor de **7** que está lá será perdido.

A visualização do Array é bem parecida com a visualização de uma variável, só que difere em alguns pontos. A Figura 26 mostra muito bem isso. Uma linha envolve o array para separá-lo das variáveis para não confundir o usuário. Além disso, em cada botão, além do nome e do valor que são apresentados em uma

variável comum, o índice do Array também é apresentado, tornando fácil a leitura por parte do usuário.

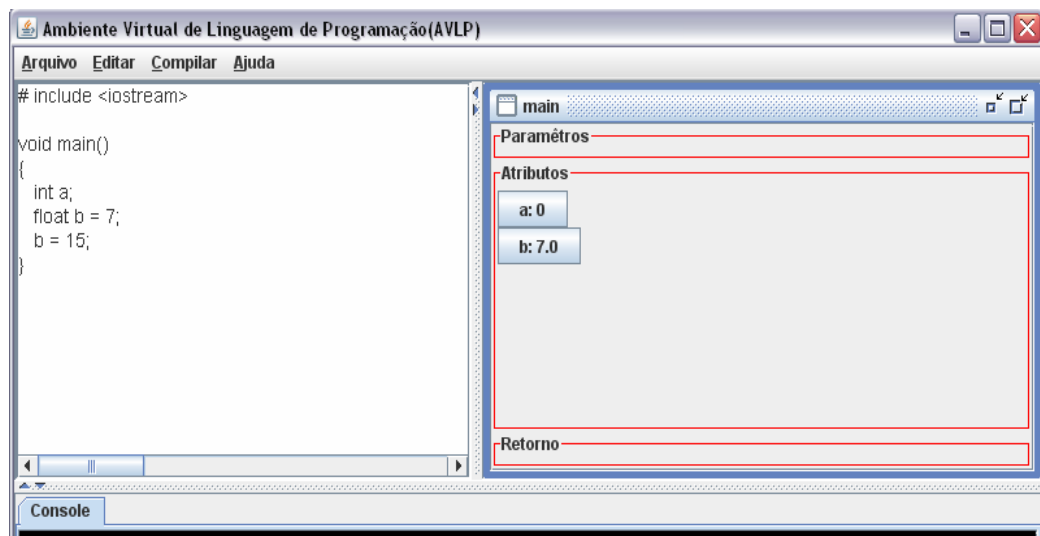


Figura 24: Criação de variável

Fonte: Banco de imagens do autor

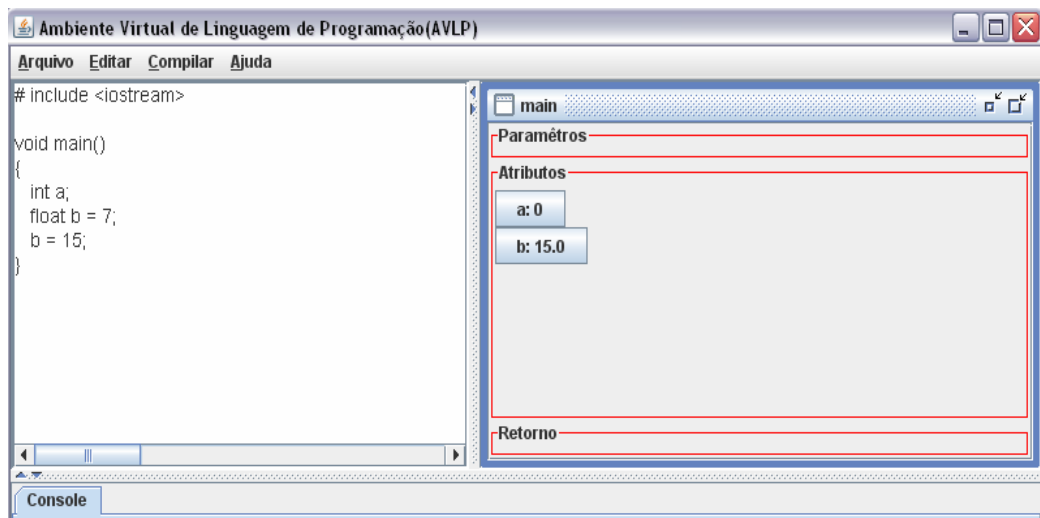


Figura 25: Valor da variável modificada

Fonte: Banco de imagens do autor

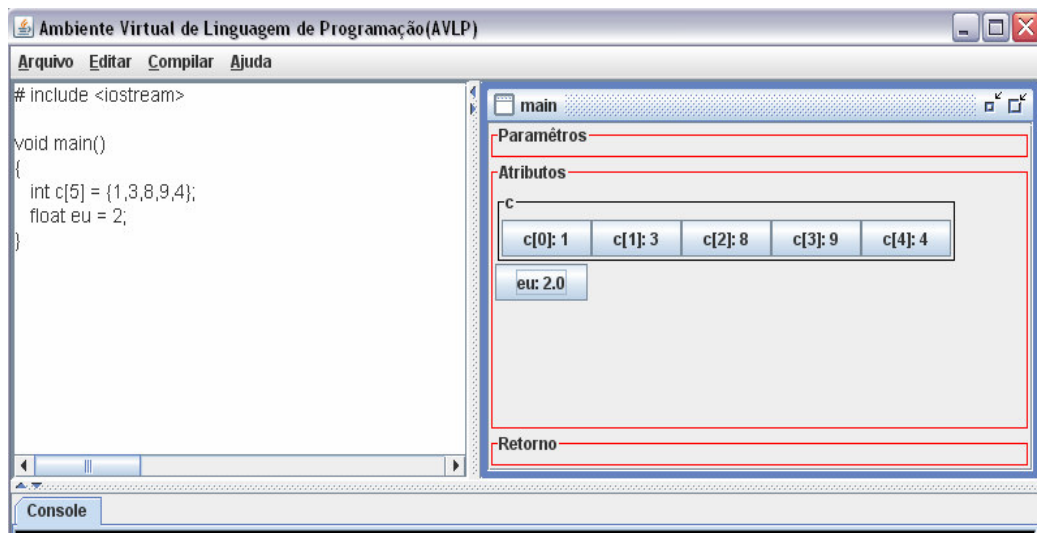


Figura 26: Array sendo criado no AVLP

Fonte: Banco de imagens do autor

Quando uma função é chamada no AVLP, é criado um outro *frame* com o nome da função, essa criação de *frames* é ilimitada. Se essa função contiver parâmetros, eles serão criados em um local reservado especialmente para eles. Isso foi feito para que o usuário não confunda um parâmetro com uma variável comum. Isso pode ser visto na Figura 27.

O mesmo acontece quando a função tem retorno, existe um espaço reservado para tal.

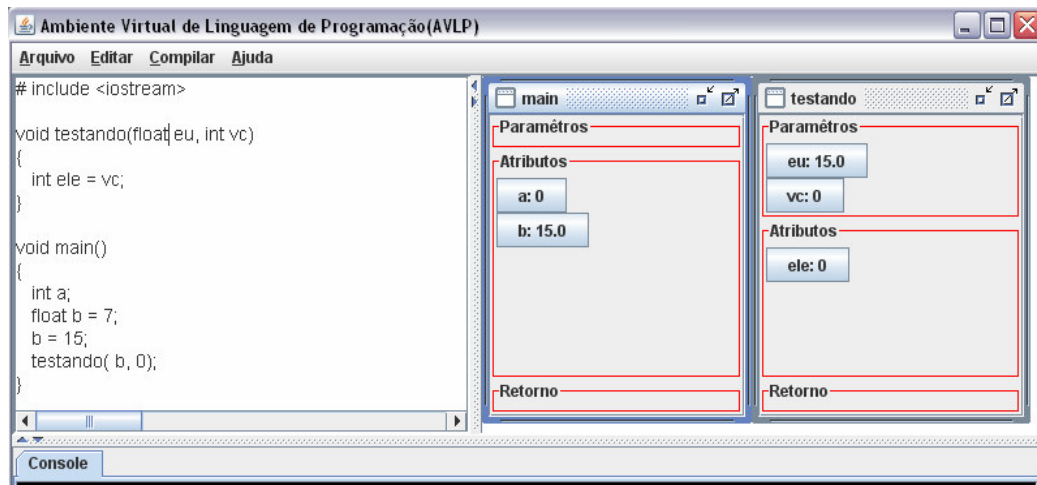


Figura 27: Chamada de função com parâmetros

Fonte: Banco de imagens do autor

Na parte inferior do AVLP, existe o console, que é o local onde o usuário imprime o resultado do programa ou algo mais que ele queira, através dos comandos **cin** e **cout**, como pode ser visto na figura 28.

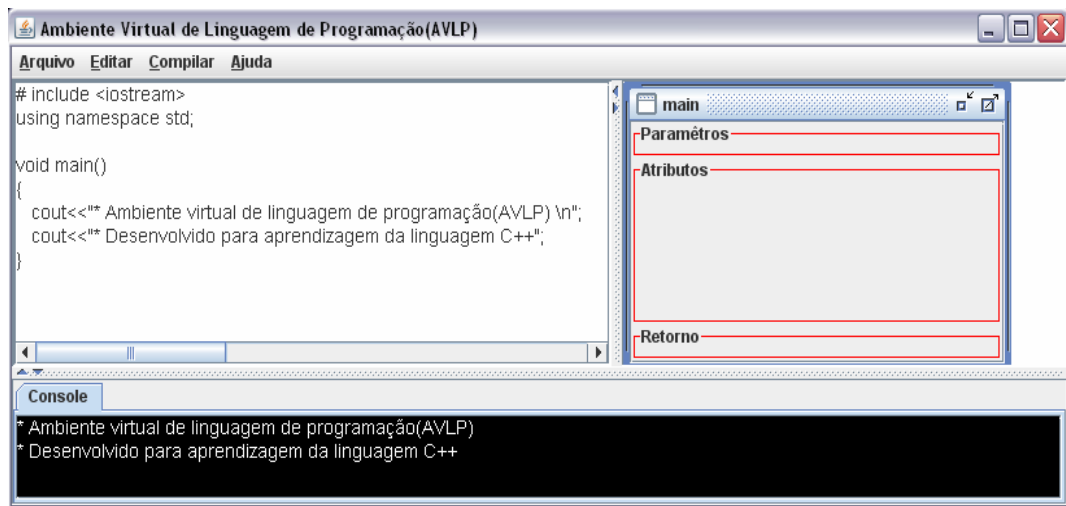


Figura 28: Console do AVLPL

Fonte: Banco de imagens do autor

O usuário pode atribuir valores as variáveis durante o tempo de execução através do comando **cin >>**, quando o AVLPL identifica esse comando, uma caixa de texto aparece pedindo para que o usuário atribua o valor. Isso pode ser visto na figura 29.

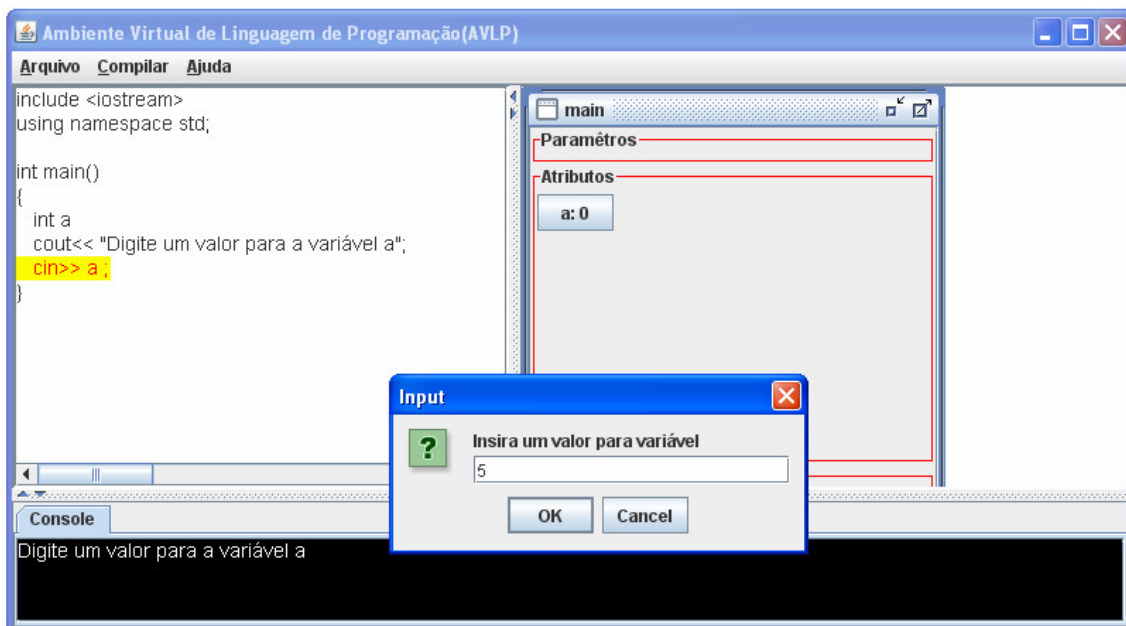


Figura 29: Entrada de dados através do teclado

Fonte: Banco de imagens do autor

4. Conclusões

Aprender a programar é essencial para um aluno de computação, mas para muitos, isso não é tão simples. Pensando nisso várias instituições de ensino e professores vêm desenvolvendo ferramentas para facilitar o aprendizado por parte do aluno. O problema é que essas ferramentas nem sempre estão disponíveis para uma grande quantidade de alunos ou são muito genéricas, com o AMBAP que utiliza uma linguagem própria.

O AVLPL vem com intuito de facilitar o aprendizado de linguagem de programação e de atingir uma grande quantidade de alunos de computação. Com a aparência simples, de fácil usabilidade e mostrando graficamente em tempo real o que está acontecendo internamente no computador, o AVLPL apresenta-se como uma ferramenta agradável e estimulante para a aprendizagem da linguagem C++ e consequentemente da lógica de programação.

Para dar continuidade a esse projeto, seria interessante que fossem sendo adicionados, aos poucos, os vários conceitos de C++ que ficaram de fora dessa versão do AVLPL. O que pode ser feito também é a integração de ambientes virtuais que analisam outras linguagens ao AVLPL, isso tornaria a ferramenta cada vez mais completa.

Referências

COSTA, Evandro De Barros; DONATO Jr, Edmundo Tojal. **ASIMAV : Ambiente de Simulação de Máquinas Virtuais.** Disponível em: <http://www.niee.ufrgs.br/ribie98/CONG_1994/DEM1_4.HTML> acessado em 20 de Dezembro de 2006

PRICE, Ana Maria de Alencar ; TOSCANI, Simão Sirineo. **Implementação de Linguagens de Programação : Compiladores.** 3ª Edição. Rio Grande do Sul -RS : Sagra Luzzatto, 2005.

MENDES, Antonio José Nunes. **Software educativo para apoio à aprendizagem de programação.** Disponível em: <http://www.tise.cl/archivos/tise01/pags/charlas/charla_mendes.htm> acessado em 14 de Janeiro de 2007

UFAL. **AMBAP - Ambiente de Aprendizado de Programação.** Disponível em: <<http://www.ufal.br/tci/ambap/>> Acessado em em 20 de Junho de 2007

ARAÚJO, Eduardo Belo de. **Analizador ANSI-C 2005.** 108 folhas. Tese(Pós-Graduação *Lato Sensu* Administração de Redes Linux), Universidade Federal de Lavras, Lavras-MG, 2005.

STROUSTRUP, Bjarne. **A Linguagem de Programação C++.** 3ª edição. Rio grande do Sul - RS: Bookman: 1999

DEITEL, H.M.; Deitel, P.J. . **C++ Como Programar.** 3º Edição. São Paulo – SP: Bookman: 2001.

MIZRAHI, Victorine Viviane. **Treinamento em linguagem C++.** Módulo 1. São Paulo – SP: Makron Books (Grupo Pearson): 1994

SEBESTA, Robert W. **Conceitos de Linguagens de Programação.** 4ª Edição. Porto Alegre - RS Bookman, 1999.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação.** 5ª Edição. Porto Alegre - RS Bookman, 2003.