UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

BRUNO GABRIEL ARAUJO LEBTAG

Cafezinho - Um ambiente de apoio ao ensino de Programação

Goiânia
2014

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE TRABALHO DE CONCLUSÃO DE CURSO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Cafezinho - Um ambiente de apoio ao ensino de Programação

Autor(a): Bruno Gabriel Araujo Lebtag

Goiânia, 17 de Julho de 2014.

Bruno Gabriel Araujo Lebtag – Autor

Thierson Couto Rosa – Orientador

BRUNO GABRIEL ARAUJO LEBTAG

Cafezinho - Um ambiente de apoio ao ensino de Programação

Trabalho de Conclusão apresentado à Coordenação do Curso de Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Área de concentração: Linguagens de Programação, Tecnologia Educacional, Ensino Profissionalizante.

Orientador: Prof. Thierson Couto Rosa

Goiânia
2014

BRUNO GABRIEL ARAUJO LEBTAG

Cafezinho - Um ambiente de apoio ao ensino de Programação

Trabalho de Conclusão apresentado à Coordenação do Curso de Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, aprovada em 17 de Julho de 2014, pela Banca Examinadora constituída pelos professores:

Prof. Thierson Couto Rosa
Instituto de Informática – UFG
Presidente da Banca

Profa. Deller James Ferreira
Instituto de Informática – UFG

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador(a).

Bruno Gabriel Araujo Lebtag

Graduando em Ciência da Computação pela UFG - Universidade Federal de Goiás. Durante sua graduação, foi monitor das disciplinas de Estruturas de Dados I e II, Programação Orientada a Objeto e Compiladores.

Dedico este trabalho aos dois pais que tive em minha vida: ao meu pai que me gerou e com ao qual convivi muito pouco, mas que mesmo neste pouco tempo me ensinou que o importante é observar e aprender; e a Deus que eu acredito que cuidou e vem cuidado de mim e da minha família.

Agradecimentos

Agradeço a Deus que durantes todos esses anos em meio a muitas dificuldades nunca me abandonou e me possibilitou estar onde eu estou agora. Agradeço expecialmente também a Nossa Senhora de Lourdes que tanto rezei para entrar na UFG e depois com muito esforço e oração consegui finalmente entrar e cursar o curso que tanto queria.

Agradeço a Universidade Federal de Goiás que proporcionou vários momentos felizes na minha vida, com boas amizades, excelentes professores e um ensino de qualidade e que agora me confere uma profissão que eu não saberia e não quereria ser outra coisa.

Agradeço meu orientador Prof^{o} Dr^{o} Thierson C. Rosa pelo apoio, dicas, orientação e suporte durante toda está caminha curta porém intensa, sendo sempre atencioso e sábio.

Agradeço a minha família que sempre me apoiou em tudo. A minha mãe que mesmo sem entender muito o que eu faço sempre me apoiou e ouviu. Ao meu irmão fiel escudeiro e colaborador incansável. Ao meu pai que mesmo não estando mais presente continua torcendo e me apoiando sempre.

Agradeço especialmente a minha amiga Beatriz Proto por ajudar na revisão deste trabalho.

A todos que de forma direta ou indiretamente contribuíram para minha formação, o meu muito obrigado.

Quando você estiver na floresta perdido e não saber o que comer, observe os pássarinhos pois eles conhecem o que é bom ou não.

Reine L. Lebtag,
Numa conversa com seu filho.

Resumo

Lebtag, Bruno G. A.. **Cafezinho - Um ambiente de apoio ao ensino de Programação**. Goiânia, 2014. 92p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

Atividade de aprender a programar computadores é geralmente difícil para a maioria das pessoas. Existem vários obstáculos nesta aprendizagem, e o principal deles é a dificuldade de expressar a solução de um problema em uma linguagem formal. Esta linguagem formal corresponde a uma linguagem de programação. Escrever um programa é traduzir uma solução em uma sequência formada pela combinação de: comandos simples, sequências de comandos, comando de repetição, comando de seleção. Outro obstáculo geralmente encontrado pelos iniciantes é que as linguagens de programação estabelecem determinadas regras de sintaxe e semântica extremamente rígidas para a escrita de programas. Há, ainda, um fator de dificuldade para aprendizes que têm como língua nativa o português que é a compreensão das mensagens de erros emitidas pelos compiladores que são em geral mensagens em inglês e utilizam jargões.
Este trabalho de conclusão de curso tem como objetivo criar um ambiente de programação que visa facilitar o aprendizado de programação focando em diminuir a complexidade da linguagem de programação, criando um ambiente agradável ao aprendizado e minimizar o problema das mensagens de erros emitidas pelo compilador expressando-as em português de forma a facilitar a compreensão do aprendiz.

Palavras–chave

CafezinhoIDE, cafezinho, linguagem, IDE, interpretador, compilador, editor, aprendizado, iniciantes.

Abstract

Lebtag, Bruno G. A.. **Cafezinho - A supportive environment for teaching programming**. Goiânia, 2014. 92p. Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

The Activity to learn computer programming is generally difficult for most people. There are several obstacles in the process, and the biggest among them is the difficulty of expressing the solution of a problem in a formal language. This formal language corresponds to a programming language. Writing a program is to translate a solution into a sequence formed by the combination of: simple commands, command sequences, repeat command, select command. Another obstacle often encountered by beginners is that the programming languages establish certain rules of syntax and semantics extremely rigid for writing programs. There is also a difficulty factor for learners whose native language is Portuguese which is the understanding of the error messages issued by the compilers that are usually messages in English and use jargon.
This project aims to create a programming environment designed to facilitate the learning of programming focusing on reducing the complexity of the programming language, creating a pleasant learning environment and to minimize the problem of error messages issued by the compiler expressing it in Portuguese, in order to facilitate understanding of the learner.

Keywords

CafezinhoIDE, cafezinho, language, IDE, interpreter, compiler, editor, learning, beginner.

Sumário

Lista de Figuras **11**

Lista de Códigos de Programas **12**

1 Introdução **13**

2 Visão Geral do Projeto **15**
- 2.1 Como surgiu a linguagem Cafezinho 15
- 2.2 Ambiente de desenvolvimento para o CafezinhoIDE 16
- 2.3 Arquitetura do Projeto 16
- 2.4 Projetos Semelhantes 17
  - 2.4.1 Logo Programming Language 18
  - 2.4.2 Ch 18
  - 2.4.3 Portugol 18
  - 2.4.4 Scratch 19

3 Linguagem Algoritmica **20**
- 3.1 Visão Geral da Linguagem 21
- 3.2 Tipagem na Linguagem Cafezinho 21
- 3.3 Precedência de Operadores 22
- 3.4 GLC da Linguagem Cafezinho 23

4 Máquina Virtual **28**
- 4.1 Herança e Polimorfismo 28
- 4.2 Arquitetura da Máquina Virtual 28
  - 4.2.1 Celulas de Memória 30
  - 4.2.2 Instruções da Máquina Virtual 31
- 4.3 Execução da Máquina Virtual 32

5 Interface Gráfica do CafezinhoIDE **33**
- 5.1 Interface Gráfica 33
- 5.2 Menus 35
  - 5.2.1 Menu Arquivo 36
  - 5.2.2 Menu Editar 36
  - 5.2.3 Menu Pesquisar 36
  - 5.2.4 Menu Ver 37
  - 5.2.5 Menu Execução 38
  - 5.2.6 Menu Editor 38
  - 5.2.7 Menu Documentos 38

5.2.8 Menu Ajuda 39
5.3 Execução 39
5.4 Depurador 40
5.5 Implementação do Depurador na Interface 40
5.6 Comunicação Máquina Virtual e Interface Gráfica 41

6 Compilador 42
6.1 Análise Léxica 42
6.2 Análise Sintática 42
6.3 Árvore de Sintaxe Abstrato 43
6.4 Análise Semantica 45
6.5 Geração de Código 47
6.6 Geração de Código para Depurador 51

7 Expansões e Melhorias 54
7.1 Melhorias na Interface Gráfica 54
7.2 Melhorias no Editor CafezinhoIDE 54
7.3 Melhorias no Compilador e na Linguagem Cafezinho 55

8 Conclusão 56

Referências Bibliográficas 57

A Linguagem de Programação Cafezinho 58
A.1 Variáveis 59
A.1.1 Cast 60
A.2 Vetores e Matrizes 61
A.2.1 Vetores 62
A.2.2 Matrizes 64
A.3 Operações Aritméticas e Lógicas 66
A.3.1 Soma, Substração e Multiplicação 66
A.3.2 Divisão e Potência 67
A.3.3 Resto da divisão, Shift Esquerdo e Shift Direito 67
A.3.4 Operadores Lógicos 68
A.3.5 Operadores de Atribuição 69
A.3.6 Operador Incremento e Decremento 71
A.3.7 Precedência de operadores 71
A.4 Estrutura Condicional 71
A.5 Estrutura de Repetição 74
A.6 Entrada e Saída 76
A.7 Função e Procedimento 79
A.7.1 Passagem de Parâmetro por Valor e por Referência 82
A.7.2 Retorno de valor 82
A.8 Outras Instruções 83

B Instruções da máquina virtual 84
B.1 Instruções 84

Lista de Figuras

2.1 Tela Inicial do Qt 16
2.2 Representação gráfica dos Componentes do Projeto 17

5.1 Área Editor de Texto 34
5.2 Área Execução do Programa 34
5.3 Área Variáveis 35
5.4 Funcionalidades do Editor de Texto 35
5.5 Localizar 37
5.6 Substituir 37
5.7 CafezinhoIDE solicitando o salvamento do documento 39
5.8 CafezinhoIDE destacando a linha após um breakpoint 41

6.1 Hierarquia de offsets na declaração de variável 48
6.2 Máquina de Pilha 48
6.3 Estado da pilha após uma chamada de função 49

A.1 Saída do Programa Hello World 58
A.2 Representação gráfica da variavel var1 59
A.3 Saída do Problema: pedro e maria - parte 1 62
A.4 Representação gráfica do vetor a[] 62
A.5 Representação gráfica da matriz a[][] 64
A.6 Saída do Problema: pedro e maria - parte 3 73
A.7 Saída do Problema: pedro e maria - parte 4 74
A.8 Saída do problema Leia e Escreva 77
A.9 Saída do Problema: pedro e maria - parte 6 79
A.10 Saída do Problema: pedro e maria - parte 7 82

Lista de Códigos de Programas

4.1 classe MaquinaVirtual 29
4.2 classe CelulaMemoria 30
4.3 classe Instrucao 31
4.4 MaquinaVirtual::executar() 32
6.1 classe Simbolo 46
6.2 Protótipo Análise semantico 46
6.3 classe Referencia 47
6.4 Protótipo geração de código 49
6.5 classe IDebugPasso 51
6.6 classe IDebugVariavelEmpilha 52
6.7 classe IDebugVariavelDesempilha 52
6.8 classe IDebugEmpilhaExec 53
6.9 classe IDebugDesempilhaExec 53

CAPÍTULO 1

Introdução

Atividade de aprender a programar computadores é geralmente difícil para a maioria das pessoas. Existem vários obstáculos nesta aprendizagem. O principal deles é a dificuldade de expressar a solução de um problema em uma linguagem formal que possa ser traduzida em instruções para o computador. Esta linguagem formal corresponde a uma linguagem de programação tal como C, C++, Java, entre outras. Uma linguagem de programação é formada por componentes básicos que são:

a) comandos simples;
b) sequências de comandos;
c) comando de repetição ;
d) comando de seleção.

Escrever um programa é traduzir uma solução em uma sequência formada por esses componentes. Esta forma de expressão não é natural para as pessoas. Exige um condicionamento do raciocínio e, portanto requer treinamento.

Outro obstáculo importante é que as linguagens de programação estabelecem determinadas regras de sintaxe e semântica extremamente rígidas para a escrita de programas. Tais regras são necessárias para que os programas possam ser traduzidos em instruções que o computador executa. Linguagens de programação profissionais têm vários recursos que embora úteis aos profissionais, são complicadores para aprendizes, pois aumentam a complexidade das regras da linguagem.

Há, ainda, um fator dificuldade para aprendizes que têm como língua nativa o português que é a compreensão das mensagens de erros emitidas pelos compiladores. Os compiladores são programas que traduzem programas escritos em linguagens de alto nível para a *linguagem de máquina* que é o conjunto de instruções executadas pelo computador. Quando um programa não está escrito de acordo com as regras da linguagem de alto nível, o compilador emite mensagens de erro e não conclui a tradução. Ocorre que a maioria dos compiladores gera mensagens em inglês e utilizam jargões. Essas mensagens são um complicador para muitos aprendizes que acabam por não conseguir corrigir os erros detectados por não entenderem as mensagens.

Este trabalho de conclusão de curso tem como objetivo criar um ambiente de programação que visa facilitar o aprendizado de programação. Esse ambiente deve permitir ao aprendiz concentrar-se na tarefa de expressar a sua solução em termos dos quatro componentes comentados anteriormente. Com isso, as dificuldades devido aos outros obstáculos, como a complexidade da linguagem de programação e das mensagens de erro devem ser minimizadas neste ambiente.

O ambiente proposto é denominado CafezinhoIDE. Os programas neste ambiente são escritos em uma linguagem de alto nível denominada Cafezinho, que visa ser uma linguagem com recursos básicos de uma linguagem de programação, porém evitando a complexidade das linguagens de programação tradicionais. Além disso, o ambiente tem uma série de recursos para auxiliar a geração de programas pelos aprendizes, tais como:

- facilidades de edição:
  - realce de palavras-chaves;
  - realce da correspondência de pares de delimitadores - (“”,“”) , (“(” , “)”) e (“[”, “]”).
  - numeração das linhas escritas
- facilidade para depuração de código:
  - visualizar a linha atual que irá ser executada
  - visualizar o valor das variáveis declaradas

Este trabalho está organizado em capítulos. No Capítulo 2 abordaremos a ideia inicial do projeto, como ele se dividiu, em qual ambiente o projeto foi criado e fazemos uma breve comparação com outros projetos semelhantes. No Capítulo 3 explicamos os aspectos pertinentes à gramática da linguagem Cafezinho. No Capítulo 4 entramos em detalhes de implementação da máquina virtual que executa o código gerado pelo compilador da linguagem. No Capítulo 5 apresentamos a interface gráfica e suas funcionalidades. No Capítulo 6 abordamos todos os aspectos e características do compilador cafezinho. No Capítulo 7 oferecemos sugestões de continuidade e melhoria. No Capítulo 8 apresentamos as conclusões do projeto. Para finalizar, no apêndice introduzimos um pequeno tutorial sobre a linguagem Cafezinho e discrevemos todas as instruções da máquina virtual implementadas para o projeto.

CAPÍTULO 2

Visão Geral do Projeto

Ao contrário de outros projetos que visam ensinar a lógica de programação[12] sem o uso de uma linguagem de programação, este projeto propõem uma linguagem simplificada. A linguagem utilizada é denominada Cafezinho e não tem todas as funcionalidades das linguagens de programação C[10], C++[13] ou Java[7]. Porém a linguagem possui construções básicas importantes para o programador principiante.

Outro objetivo do projeto foi o de criar um ambiente de desenvolvimento de programas em português e que permitisse a edição de programas e o acompanhamento de suas execuções através de uma interface gráfica.

Com estes pontos em mente foi desenvolvido o projeto CafezinhoIDE, um ambiente de programação com: editor de texto, compilador e interpretador da linguagem Cafezinho e depurador.

2.1 Como surgiu a linguagem Cafezinho

A linguagem cafezinho foi projetada para ser utilizada como linguagem fonte na disciplina de compiladores ministrado pelo Universidade Federal de Goiás pelo Professor Dr^{o} Thierson Couto Rosa e foi criada no primeiro semestre de 2013.

A Linguagem é uma evolução da linguagem Cezinho utilizada pela primeira vez no segundo semestre de 2008 na mesma disciplina. A linguagem Cezinho é baseado na linguagem de programação C e contém um subconjunto de instruções da linguagem, mas com algumas modificações, por exemplo, em Cezinho a entrada e saída são implementadas por duas instruções simples: read e write, respectivamente enquanto que na linguagem C, a entrada e saída são implementadas por funções com muito mais recursos. A linguagem Cafezinho corresponde a uma tradução das palavras chaves da linguagem Cezinho para o português. A linguagem será abordada em detalhes no Capítulo 3.

2.2 Ambiente de desenvolvimento para o CafezinhoIDE

Para desenvolver este projeto foi utilizado a linguagem de programação C++ usando Qt [2] que é uma framework multi-plataforma para desenvolvimento de software para interface gráfica criado pela empresa Trolltech.

O Qt (pronuncia-se como a palavra inglesa cute) está disponível sob a licença comercial, GPL v3 e LGPL v2. Possui suporta para plataformas como Windows, OS X, X11, entre outros. Possui uma ampla documentação, tutorial, vídeos, etc., além de uma comunidade muito ativa e sempre pronta para ajudar. O Qt também pode ser utilizado em plataformas de dispositivos moveis como Nokia, Maemo. Além disso, ele é utilizado por empresas grandes como Disney, DreamWorks SKG, LucasFilms, NASA, etc. A figura 2.1 apresenta a tela inicial do editor do Qt.

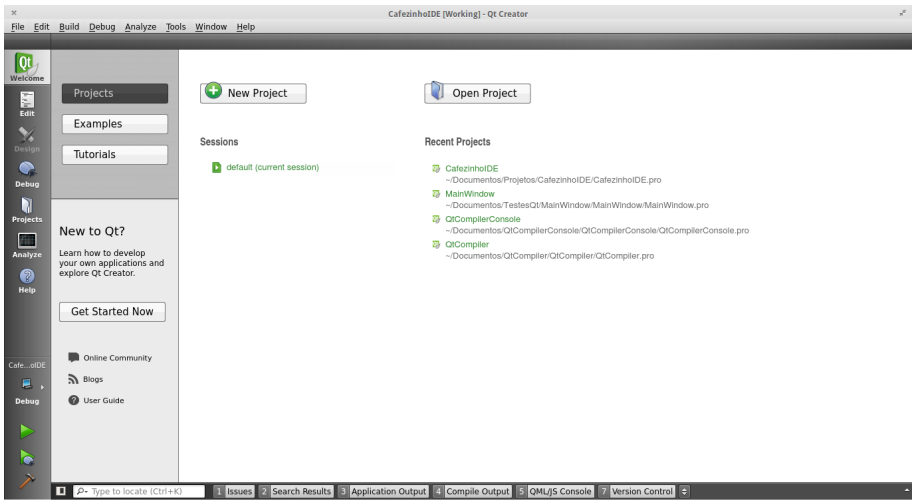


Figura 2.1: *Tela Inicial do Qt*

2.3 Arquitetura do Projeto

O desenvolvimento de uma linguagem de programação é algo complexo e trabalhoso, porém permite a separação do desenvolvimento em etapas bem distintas, ex.: análise léxica, análise sintática, análise semântica, backend do compilador, frontend do compilador, etc.

O projeto também tem outras partes como: desenvolvimento da interface gráfica, execução do código intermediário gerado, depurador do código intermediário gerado. Por isso podemos dividir o projeto em 4 grandes partes ou componentes: Compilador, Maquina Virtual, Interface Gráfica, Depurador. A Figura 2.2 mostra os quatro componentes do projeto. Os quais são comentados suscintamente a seguir. Estes componente serão detalhados nos próximos capítulos.

O componente Compilador pode dividido funcionalmente em 4 módulos:

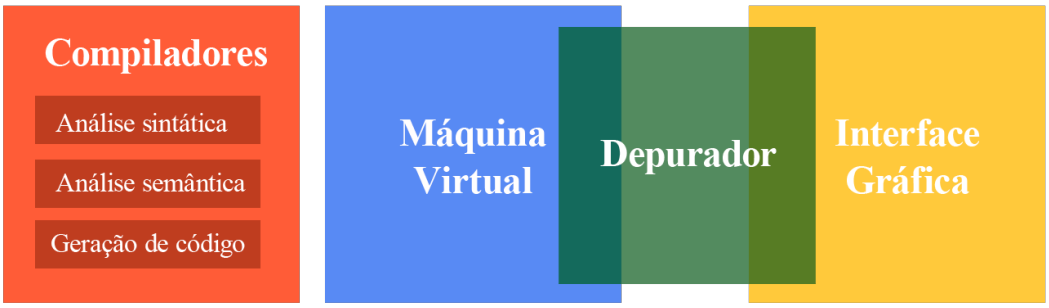


Figura 2.2: *Representação gráfica dos Componentes do Projeto*

Dentro do componente Compilador podemos dividir em 4 etapas:

- Analisador léxico: Este módulo lê a sequencia de caracteres que forma o programa em cafezinho, a procura de palavras-chaves, operadores e identificadores que são denominado átomos da linguagem. Para a construção deste módulo foi utilizado o programa Flex (Fast Lexical Analyser Generator) da GNU [11]
- Analisador sintático: Este módulo agrupa os átomos retornados pelo analisador léxico para verificar se a sequência de átomos do programa em Cafezinho obedece as regras sintáticas da linguagem, as quais são descritas pela gramática da linguagem, apresentada na Seção 3.4. O analisador sintático foi implementado utilizando-se o programa Bison [11].
- Analisador semântico: É o módulo que verifica se o programa escrito na linguagem Cafezinho está correto semanticamente. Foi codificado utilizando-se a linguagem C++.
- Geração de Código: Nesta etapa percorre-se a representação intermediaria do programa Cafezinho gerada pela etapa de analise sintática e, posteriormente, gerando instruções para a máquina virtual. Este modulo foi desenvolvido na linguagem C++.

No componente *máquina virtual* foi desenvolvido uma máquina virtual responsável por simular a execução do programa traduzido pelo compilador. No componente *interface gráfica* foi desenvolvida toda a interface gráfica, sua prototipação, codificação e interação com os componentes Compilador e Máquina Virtual. O Componente *Depurador* foi parte implementado na interface gráfica e parte implementado na máquina virtual.

2.4 Projetos Semelhantes

Existem vários ambientes de programação e cada uma possuindo um foco principal, dentre eles existem alguns que possuem como objetivo principal auxiliar o

aprendizado de programação. Nesta seção apresentaremos algumas desses ambientes e o que diferenciam do projeto CafezinhoIDE.

2.4.1 Logo Programming Language

Logo é um dialeto da linguagem de programação Lisp que trabalha com o paradigma de programação funcional, possui várias implementações entre elas podemos destacar: MicroWorlds Logo e Imagine Logo. Logo foi criada em 1967 por Daniel G. Bobrow, Wally Feurzeig, Seymour Papert e Cynthia Solomon [8]. Tem como principal objetivo ensinar os aspectos da linguagem Lisp para crianças.

O projeto CafezinhoIDE tem como objetivo ensinar adolescentes e crianças porém trabalha com o paradigma da programação imperativa que é diferente da funcional além de ser uma linguagem voltada para alunos da linguagem portuguesa, facilitando assim o aprendizado da linguagem de programação e por consequências a aprendizagem dos aspectos de programação

2.4.2 Ch

Ch[1] é um ambiente de desenvolvimento em linguagem C/C++ que pode ser utilizado por estudantes, professore e engenheiros, tudo de forma interpretada. Foi Originalmente desenhada para ensinar os iniciantes em matemática, computação e programação em C/C++. Possui um ambiente integrado chamado ChIDE com interpretador, editor e depurador.

Assim como ChIDE o projeto CafezinhoIDE é uma IDE que integra interpretador da linguagem Cafezinho, editor e Depurador, além de ser baseado na linguagem C. Porém é mais simplificado do que a linguagem C, com instruções na língua portuguesa, bem como as mensagens de erro emitidas pelo compilador.

2.4.3 Portugol

Portugol IDE[5] é um ambiente de desenvolvimento criado no Instituto Politécnico de Tomar em Portugal para o ensino no próprio instituto de linguagem de programação. A linguagem Portugol é baseada na linguagem Pascal. Ela é muito utilizada no Brasil para ensino de programação em cursos de programação e em algumas faculdades por ser em Português.

A Linguagem Cafezinho é baseada na linguagem C que é muito mais utilizada do que a linguagem Pascal, além disso, a linguagem C é base de várias outras linguagens de programação, como C++, Java, C#. Assim, aprender a programar com uma linguagem próxima do C é mais vantajoso que uma em Pascal.

2.4.4 Scratch

Scratch é um ambiente e linguagem de programação gratuito criado no Media Lab do MIT[12]. Scratch trabalha com programação orientada a eventos e utiliza-se de múltiplos objetos ativos chamados de *sprites*. A linguagem procura abstrair todos os aspectos da programação para que qualquer pessoa aprenda a programar sem o conhecimento prévio de qualquer linguagem de programação.

O projeto CafezinhoIDE objetiva o ensino de programação, porém sem abstrair todos os seus aspectos, uma vez que permite ao aluno ter um domínio sobre a lógica de programação e sobre sua implementação. O projeto visa também criar conteúdo informativo a cerca da linguagem Cafezinho e sobre o ambiente de desenvolvimento.

CAPÍTULO 3

Linguagem Algoritmica

A Linguagem Cafezinho foi originalmente criada para a disciplina de Compiladores desenvolvida pelo Profº Thierson, seu objetivo original era ensinar aos alunos da graduação a desenvolver um compilador. A gramática original é simples e tenta ser fácil para os alunos conseguirem desenvolver um compilador em um semestre.

O Projeto CafezinhoIDE visa a aprendizagem de programação de uma forma simplificada mais sem abstrair o que é programar. Portanto a gramática do projeto é uma expansão da gramática original para tornar a linguagem mais próxima das linguagens de programação utilizadas no mercado.

O projeto da linguagem Cafezinho considera que o aluno que está aprendendo a programar pode ter dificuldades com várias estruturas de repetição ou várias estruturas condicionais ou várias estruturas de salto, etc. Assim, na linguagem Cafezinho existe um conjunto reduzido dessas instruções. Porém algumas novas funcionalidades foram adicionadas à gramática original como:

- inicialização de uma variável (escalar ou vetor);
- dois novos tipos de dados: Nulo e Real;
- implementação de cast para os valores do tipo: real, int, car;
- declaração de vetor de várias dimensões;
- operador de potência: **;
- operadores de atribuição : +=, -=, *=, /=, %=, &=, |=, **=, ^=, /=, »=, «=;
- operações de shift e binárias: », «, |, &, ^;
- passagem de vetor de várias dimensões como parâmetro de função;
- expansão das funcionalidades da instrução escreva para passar vários parâmetros a serem impressos;
- expansão das funcionalidades da instrução leia para ler vários parâmetros de maneira atômica.

3.1 Visão Geral da Linguagem

A gramática da linguagem tem como inspiração a linguagem C, a qual é amplamente utilizada, além de que várias outras linguagens de programação possuem uma escrita similar a ela. Dessa forma, o aluno que aprender Cafezinho pode facilmente se adaptar à outras linguagens, uma vez já familiarizado com os aspectos da programação. A seguir apresentamos todos os comandos da linguagem Cafezinho:

- uma estrutura de repetição (*enquanto . . . execute*);
- duas estruturas condicionais (*se . . . então* e *se . . . então . . . senão*);
- uma instrução de leitura (*leia*);
- uma instrução de escrita (*escreva*);
- uma instrução para indicar uma nova linha (*novalinha*), podendo ser utilizada em conjunto com a instrução escreva;
- uma instrução para limpar o terminal (*limpar*);
- uma instrução para encerrar imediatamente a execução do programa (*terminar*);

A linguagem trabalha com 4 tipos de dados:

- caractere (*car*) ex.: a, b, +;
- inteiro (*int*) ex.: 12342, 10, 1, 245. Aceita valores em hexadecimal (ex.: 0x123, 0x56, 0x34) e valores em octeto (ex.: 034, 056, 077);
- real (real) ex.: 123.123, 546e+10, 12323e-123;
- nulo (nulo).

Porém, o tipo nulo não pode ser declarado como variável ou parâmetro, mas pode ser utilizado como tipo de retorno de uma função ou no protótipo da função.

3.2 Tipagem na Linguagem Cafezinho

A Linguagem Cafezinho possui uma tipagem estática (o tipo tem que ser previamente declarado) e fraca (variável tipo *int* pode receber *car* e variável tipo *real* pode receber *int* e *car*) semelhante à linguagem C. Algumas instruções são estritamente tipadas sobre operandos do tipo *int*:

- shift esquerdo e direito;
- operações binárias &, |, ^;
- operação de módulo %;
- operações de comparação <, >, >=, <=, ==, !=;

As operações potência e divisão retornam, estritamente, valores do tipo real. Operações de: +, -, * obedecem a regra do tipo que é o "maior recipiente", ex.: variável tipo real somado a variável tipo car retorna um valor do tipo real.

3.3 Precedência de Operadores

A linguagem Cafezinho possui a seguinte hierarquia de precedência começando com a maior precedência para menor e da esquerda para direita:

1. ()
2. ++, –, !, ~, (tipo)
3. **, *, /, %
4. +, -
5. », «
6. <, <=, >=, >
7. ==, !=
8. &
9. ^
10. |
11. &&
12. ||
13. ? :
14. =, operador=
15. ,

3.4 GLC da Linguagem Cafezinho

programa → *unidade_traducao*

unidade_traducao → *declaracao_externa*
| *unidade_traducao declaracao_externa*

declaracao_externa → *tipo_especificador declarador_funcao*
instrucao_composta
| *tipo_especificador inicio_lista_declaracao* ;

tipo_especificador → **NULO** | **CAR** | **INT** | **REAL**

declarador_funcao → **IDENTIFICADOR** (*lista_parametro*)
IDENTIFICADOR ( )

lista_parametro → *declaracao_parametro*
| *lista_parametro* , *declaracao_parametro*

declaracao_parametro → *tipo_especificador declarador*

declarador → **IDENTIFICADOR** *lista_ponteiro_vetor*
| **IDENTIFICADOR**

lista_ponteiro_vetor → [ ]
| [**INT_CONST**]
| *lista_ponteiro_vetor* [**INT_CONST**]

instrucao_composta → { }
| { *lista_instrucao* }
| { *lista_declaracao* }
| { *lista_declaracao lista_instrucao* }

lista_instrucao → *instrucao*
lista_instrucao instrucao

$$
\begin{aligned}
\textit{instrucao} &\rightarrow \textit{instrucao_composta} \\
&\mid \textit{instrucao_expressao} \\
&\mid \textit{instrucao_decisao} \\
&\mid \textit{instrucao_iteracao} \\
&\mid \textit{instrucao_entrada}_{s}\textit{aida} \\
&\mid \textit{instrucao_salto} \\
&\mid \textit{instrucao_sistema}
\end{aligned}
$$

$$
\textit{instrucao_expressao} \rightarrow ; \mid \textit{expressao};
$$

$$
\begin{aligned}
\textit{instrucao_decisao} &\rightarrow \textbf{SE}\,(\,\textit{expressao}\,)\,\textbf{ENTAO}\,\textit{instrucao} \\
&\mid \textbf{SE}\,(\,\textit{expressao}\,)\,\textbf{ENTAO}\,\textit{instrucao}\,\textbf{SENAO}\,\textit{instrucao}
\end{aligned}
$$

$$
\textit{instrucao_iteracao} \rightarrow \textbf{ENQUANTO}\,(\,\textit{expressao}\,)\,\textbf{EXECUTE}\,\textit{instrucao}
$$

$$
\begin{aligned}
\textit{instrucao_entrada_saida} &\rightarrow \textbf{LEIA}\,\textit{expressao}; \\
&\mid \textbf{ESCREVA}\,\textit{expressao}; \\
&\mid \textbf{NOVA_LINHA};
\end{aligned}
$$

$$
\begin{aligned}
\textit{instrucao_salto} &\rightarrow \textbf{RETORNE}\,; \\
&\mid \textbf{RETORNE}\,\textit{expressao};
\end{aligned}
$$

$$
\begin{aligned}
\textit{instrucao_sistema} &\rightarrow \textbf{TERMINAR}\,; \\
&\mid \textbf{LIMPAR};
\end{aligned}
$$

$$
\begin{aligned}
\textit{lista_declaracao} &\rightarrow \textit{declaracao} \\
&\mid \textit{lista_declaracao}\,\textit{declaracao}
\end{aligned}
$$

$$
\textit{declaracao} \rightarrow \textit{tipo_especificador}\,\textit{inicio_lista_declaracao}
$$

$$
\begin{aligned}
\textit{inicio_lista_declaracao} &\rightarrow \textit{inicio_declarador} \\
&\mid \textit{inicio_lista_declaracao}\,,\,\textit{inicio_declarador}
\end{aligned}
$$

$$
\begin{aligned}
\textit{inicio_declarador} &\rightarrow \textit{declarador} \\
&\mid \textit{declarador} = \textit{inicializador}
\end{aligned}
$$

inicializador → *expressao_atribuicao*
| { *lista_inicializador* }

lista_inicializador → *inicializador*
| *lista_inicializador* , *inicializador*

expressao_atribuicao → *expressao_condicional*
| *expressao_unaria operador_atribuicao*
expressao_atribuicao

operador_atribuicao → =
| **MULT_ATRIBUICAO**
| **DIV_ATRIBUICAO**
| **MOD_ATRIBUICAO**
| **ADICAO_ATRIBUICAO**
| **SUBTRACAO_ATRIBUICAO**
| **ESQ_ATRIBUICAO**
| **DIR_ATRIBUICAO**
| **E_ATRIBUICAO**
| **XOR_ATRIBUICAO**
| **OU_ATRIBUICAO**
| **POT_ATRIBUICAO**

expressao_condicional → *expressao_logico_ou*
| *expressao_logico_ou* ? *expressao* :
expressao_condicional

expressao_logico_ou → *expressao_logico_e*
| *expressao_logico_ou* **OU_OP** *expressao_logico_e*

expressao_logico_e → *expressao_inclusivo_ou*
| *expressao_logico_e* **E_OP** *expressao_inclusivo_ou*

expressao_inclusivo_ou → *expressao_exclusivo_ou*
| *expressao_inclusivo_ou* | *expressao_exclusivo_ou*

expressao_exclusivo_ou → *expressao_e*
| *expressao_exclusivo_ou* ^ *expressao_e*

$$
\begin{aligned}
\textit{expressao_e} &\rightarrow \textit{expressao_igualdade} \\
&\mid \textit{expressao_e}\ \&\ \textit{expressao_igualdade}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_igualdade} &\rightarrow \textit{expressao_relacional} \\
&\mid \textit{expressao_igualdade}\ \textbf{EQ_OP}\ \textit{expressao_relacional} \\
&\mid \textit{expressao_igualdade}\ \textbf{NE_OP}\ \textit{expressao_relacional}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_relacional} &\rightarrow \textit{expressao_shift} \\
&\mid \textit{expressao_relacional}\ \textbf{LT_OP}\ \textit{expressao_shift} \\
&\mid \textit{expressao_relacional}\ \textbf{BT_OP}\ \textit{expressao_shift} \\
&\mid \textit{expressao_relacional}\ \textbf{LE_OP}\ \textit{expressao_shift} \\
&\mid \textit{expressao_relacional}\ \textbf{GE_OP}\ \textit{expressao_shift}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_shift} &\rightarrow \textit{expressao_aditiva} \\
&\mid \textit{expressao_shift}\ \textbf{ESQ_OP}\ \textit{expressao_aditiva} \\
&\mid \textit{expressao_shift}\ \textbf{DIR_OP}\ \textit{expressao_aditiva}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_aditiva} &\rightarrow \textit{expressao_multiplicativa} \\
&\mid \textit{expressao_aditiva} + \textit{expressao_multiplicativa} \\
&\mid \textit{expressao_aditiva} - \textit{expressao_multiplicativa}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_multiplicativa} &\rightarrow \textit{expressao_cast} \\
&\mid \textit{expressao_multiplicativa} * \textit{expressao_cast} \\
&\mid \textit{expressao_multiplicativa} / \textit{expressao_cast} \\
&\mid \textit{expressao_multiplicativa}\ \%\ \textit{expressao_cast} \\
&\mid \textit{expressao_multiplicativa}\ \textbf{POT_OP}\ \textit{expressao_cast}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_cast} &\rightarrow \textit{expressao_unaria} \\
&\mid (\ \textit{tipo_especificador}\ )\ \textit{expressao_cast}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_unaria} &\rightarrow \textit{expressao_posfix} \\
&\mid \textbf{INC_OP}\ \textit{expressao_unaria} \\
&\mid \textbf{DEC_OP}\ \textit{expressao_unaria} \\
&\mid \textit{operador_unario}\ \textit{expressao_cast}
\end{aligned}
$$

$$
\begin{aligned}
\textit{operador_unario} &\rightarrow - \\
&\mid\ ! \\
&\mid\ \tilde{}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_posfix} \rightarrow\ & \textit{expressao_primaria} \\
& |\textit{expressao_posfix}\ \textbf{INC_OP} \\
& |\textit{expressao_posfix}\ \textbf{DEC_OP}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expressao_primaria} \rightarrow\ & \textbf{IDENTIFICADOR} \\
& |\textbf{IDENTIFICADOR} \\
& |\textbf{IDENTIFICADOR}(\ \textit{lista_expressao}\ ) \\
& |\textbf{IDENTIFICADOR}(\ ) \\
& |\textbf{INT_CONST} \\
& |\textbf{CAR_CONST} \\
& |\textbf{PALAVRA_LITERAL} \\
& |\textbf{REAL_CONST} \\
& |\textbf{NOVA_LINHA} \\
& |(\ \textit{expressao}\ )
\end{aligned}
$$

$$
\begin{aligned}
\textit{lista_expr_vetor} \rightarrow\ & [\textit{expressao}] \\
& |\textit{lista_expr_vetor}\ [\ \textit{expressao}\ ]
\end{aligned}
$$

$$
\textit{expressao} \rightarrow \textit{lista_expressao}
$$

$$
\begin{aligned}
\textit{lista_expressao} \rightarrow\ & \textit{expressao_atribuicao} \\
& |\textit{lista_expressao}\ ,\ \textit{expressao_atribuicao}
\end{aligned}
$$

CAPÍTULO 4

Máquina Virtual

O CafezinhoIDE não gera código de máquina para ser executado no processador da máquina onde o ambiente é executado . Ao término das etapas de análise léxica e sintática, se o programa de entrada escrito em Cafezinho estiver de acordo com as regras sintáticas e semânticas, é gerado um código intermediário. Esse código é, então, executado numa "máquina virtual" implementada pelo CafezinhoIDE.

4.1 Herança e Polimorfismo

Para criação da máquina virtual foram utilizados alguns princípios importantes da programação orientada a objetos. O primeiro é o conceito de herança onde, na nomenclatura da linguagem C++, a classe que é herdada é chamada classe base e a classe herdeira é chamada classe derivada. Na herança, a classe derivada herda (possui também) as características (propriedades e métodos) da classe base.

Outro princípio muito importante é o Polimorfismo, onde um método pode ter várias implementações mudando apenas a assinatura do método, além disso, um método herdado da classe base pode ser reimplementado se este for declarado como virtual. Na linguagem C++ não existe exatamente o conceito de interface, como na linguagem Java, porém pode-se conseguir o mesmo resultado utilizando-se de método virtual puro que é uma função virtual que deve ser implementada por uma classe derivada. O polimorfismo de método em C++ só pode ser alcançado através da utilização de ponteiros.

4.2 Arquitetura da Máquina Virtual

A máquina virtual foi implementada simulando uma arquitetura com pilha do programa, registradores de propósito geral, registradores de propósito específico, área do texto (código a ser executado) e registradores flag para operações booleanas. A pilha do programa e os registradores são variáveis da classe CelulaMemoria. Todos estes componentes da máquina virtual foram agrupados em uma classe chamada MaquinaVirtual,

como mostrado no código 4.1, assim permitindo a instanciação e execução de varias máquina virtuais simultâneas caso seja necessário.

A Máquina Virtual possui:

- quatro registradores de propósito geral, com o objetivo de armazenar informações temporariamente. São eles: eax, ebx, ecx, edx (uma homenagem a arquitetura x86);
- registrador de Ponteiro de Pilha, *pp*;
- registrador Endereço de Retorno, *er*;
- registrador de Ponteiro Global, para guardar o inicio da pilha, *pg*;
- registrador Contator de Instrucao, *pc*
- flags para indicar operação de maior (*bg*), menor (*sf*) e igual (*ef*);
- flag para indicar erro, *erf*;
- vetor de CelulaMemoria, funciona como a pilha do programa;
- vetor de ponteiros da classe *Instrucao*, funciona como a área de texto (código do programa);
- vetor de ponteiros de inteiros (int) para serem utilizados para guardar a informação de qual é o índice da instrução para a qual um desvio deve mover. Funcionam como rótulos em um assembly convencional.

Código 4.1 `classe MaquinaVirtual`

```
1  class MaquinaVirtual
2  {
3  public:
4      int pc;
5      CelulaMemoria pp;
6      CelulaMemoria eax;
7      CelulaMemoria ebx;
8      CelulaMemoria ecx;
9      CelulaMemoria edx;
10     CelulaMemoria bp;
11     CelulaMemoria er;
12     CelulaMemoria pg;
13     QVector<Instrucao> codigo;
14     QVector<int*>rotulo;
15     QVector<CelulaMemoria> memoria;
16     bool bf;
17     bool sf;
18     bool ef;
19     bool erf;
20 }
```

É importante notar que a pilha que é formada a partir do vetor de *CelulaMemoria* cresce em direção aos endereços menores da memoria, ao contrário do que acontece nas arquiteturas convencionais. Além disso, a unidade básica da pilha é uma *CelulaMemoria*, o que não acontece nas outras em que a unidade é um byte.

4.2.1 Celulas de Memória

Na linguagem Cafezinho trabalhamos com 3 tipos de dados: inteiro, caractere e real. Porém, internamente o computador trabalha com uma representação numérica inteira para representar caracteres, portanto lidamos apenas com dois tipos distintos de dados: inteiro e ponto flutuante. A classe *CelulaMemoria*, apresentada no código 4.2, possui um union que pode armazenar int ou double. A classe também armazena qual tipo de dado (inteiro ou double) aquela célula atualmente armazena.

Código 4.2 `classe CelulaMemoria`

```
1  class CelulaMemoria
2  {
3  public:
4      CelulaMemoria();
5      // Overload dos operadores
6      // =, +, -, *, /, &, |, ^,
7      // >>, <<, %, ~, ++, --
8      // ==, !=, >, <, >=, <=
9      // Criado o metodo pot();
10     union
11     {
12         int inteiro;
13         double real;
14     }celula;
15     enum
16     {
17         INTEIRO, REAL
18     }tipo;
19 };
```

A linguagem de programação C++ oferece a funcionalidade ao programador de sobrescrever o comportamento padrão do compilador quando este encontra um operador, por exemplo, +, -, /, etc. Esta capacidade é denominada sobrecarga de operadores e é usada pela classe *CelulaMemoria* para permitir uma maior legibilidade do código. Estes

novos comportamentos implementados obedecem a regra do “maior recipiente” assim, por exemplo, uma operação sobre inteiro e double, retorna uma célula do tipo double.

Na classe *MaquinaVirtual* os registradores de propósito geral, registradores de propósito específico e pilha do programa são representados como um vetor do tipo *CelulaMemoria*.

4.2.2 Instruções da Máquina Virtual

Para implementar as instruções que a Máquina Virtual pode executar foi definida a classe Instrucao, cuja declaração é mostrada no Código 4.3, que funcionaria como uma interface (do Java) para as demais classes.

Para cada instrução que a maquina virtual pode executar é criada uma classe específica, a qual implementa o método *void execute(MaquinaVirtual &vm)* definido na classe Instrução.

Código 4.3 `classe Instrucao`

```
1 class Instrucao
2 {
3 public:
4     Instrucao();
5     virtual ~Instrucao();
6     virtual void execute(MaquinaVirtual &vm) = 0;
7     virtual TipoInstrucao::TipoInstrucao tipoInstucao()=0;
8 };
```

A implementação do método *execute(MaquinaVirtual &vm)* utiliza os registradores de propósito geral para, por exemplo, executar alguma operação matemática, fazer uma operação lógica, ligar ou desligar uma flag, salvar ou copiar um dado da pilha de execução ou, alterar o registrador pc (contador de programa) em uma instrução de desvio.

É importante notar que no processo de geração de código estas instruções serão instanciadas e inseridas em um vetor de objetos da classe *Instrucao*. Esse vetor corresponde à área de texto do código. Porém, estas instruções só serão executadas depois de terem todas sido instanciadas.

Para poder utilizar estas instruções de forma maleável, os registradores que são operandos de uma instrução, são passados como parâmetros para o método construtor da instrução. Desta maneira podemos usar a mesma instrução para aplicar a operação sobre registradores diferentes. É responsabilidade das classes que implementam o método *execute(MaquinaVirtual &vm)* de incrementar o registrador *pc*, exceto, as instruções de salto.

No apêndice B apresentamos uma descrição detalhada de todas as instruções criadas para o projeto CafezinhoIDE.

4.3 Execução da Máquina Virtual

Para executar a máquina virtual basta chamar a função void executar(), apresentada no código 4.4, essa por sua vez executa a instrução que está no índice *pc*. Cada Instrução incrementa o valor de *pc*, a menos que ocorra um erro na execução da instrução, ou a instrução execute um desvio para outra instrução. A execução das instruções termina quando uma instrução *IParar* for encontrada.

Código 4.4 `MaquinaVirtual::executar()`

```
1  void executar()
2  {
3      while(execute&&(!erf))
4      {
5          codigo[pc]->execute(*this);
6
7          if(pc<0||pc>=codigo.size()||erf)
8          {
9              if(!tp)
10             {
11                 //mensagem de erro
12                 erf = true;
13             }
14         }
15     }
16 }
```

O princípio da execução do código ocorre devido ao polimorfismo sobre o métodos *executar()*. Para a classe *MaquinaVirtual*, existe apenas um vetor de instâncias da classe *Instrucao*, que possuem o método *execute(MaquinaVirtual &vm)* . Porém cada instância é das classes herdadas de *Instrucao* e possuem implementações diferentes. Caso não fosse utilizado essa técnicas seria necessário projetar uma forma de fazer essa separação de implementações.

CAPÍTULO 5

Interface Gráfica do CafezinhoIDE

Um dos objetivos do projeto é a criação de uma interface interativa e rica para facilitar o uso de várias funcionalidades do CafezinhoIDE. É função da interface fazer a interfaciação com a edição de programas, compilação e execução e depuração de um programa do usuário.

Para o desenvolvimento da interface gráfica, foi utilizado como fonte de inspiração, o programa Visual Studio[4], da Microsoft, devido ao fato de possuir um ambiente e um depurador muito poderoso. Outros softwares que influenciaram bastante foram o ChIDE, apresentado na subseção 2.4.2, e o editor de texto Sublime Text Editor[3], pesquisados na etapa de análise de interface gráfica similares para o desenvolvimento da interface gráfica do projeto CafezinhoIDE.

Antes do desenvolvimento da interface foi desenvolvido um protótipo da interface gráfica usando a ferramenta moqups.com no desenvolvimento da etapa de Wireframe e verificar a viabilidade da interface, além de dar uma visão de como a interface gráfica seria.

5.1 Interface Gráfica

A interface gráfica do ambiente CafezinhoIDE possui três regiões principais:

- Editor de Texto: área onde o usuário pode escrever o código que posteriormente poderá ser gravado no disco e executado. A figura 5.1 mostra a área do editor.



Figura 5.1: *Área Editor de Texto*

- Execução do programa: área onde o usuário pode ver as saídas do programa, conforme mostra a figura 5.2.



Figura 5.2: *Área Execução do Programa*

- Variáveis: área onde o usuário pode inspecionar o valor das variáveis quando o programa estiver sendo executado no modo depurador. Essa área é apresentada na figura 5.3.

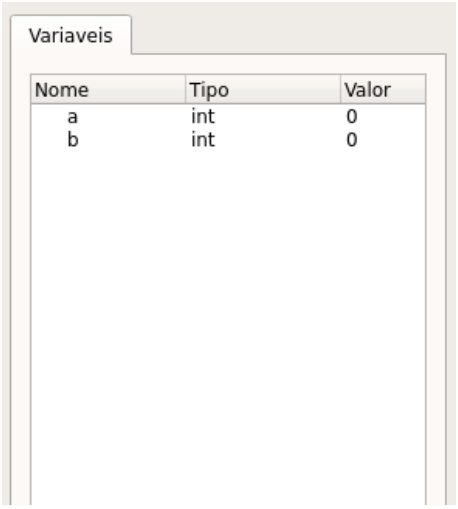


Figura 5.3: *Área Variáveis*

Na área do editor de texto o usuário pode escrever o código do programa, editá-lo e salvá-lo. Para facilitar a visualização do código o editor realça o nome das palavras chaves da linguagem Cafezinho. O Editor também possui a funcionalidade de enumeração das linhas escritas. Além disso, o editor também implementa a funcionalidade de realçar os pares de chaves , parênteses () e colchetes [] conforme o cursor caminha no texto. A figura 5.4 mostra alguns desses recursos.



Figura 5.4: *Funcionalidades do Editor de Texto*

5.2 Menus

Todas as funcionalidades do CafezinhoIDE podem ser acessadas pelos menus superiores. Entre eles podemos destacar: salvar o código, localizar palavras no texto,

executar ou depurar o código, entre outras.

5.2.1 Menu Arquivo

No menu Arquivo podemos encontrar todas as funcionalidades relacionadas ao documento. Temos:

- Novo: cria um novo documento em uma aba, para permitir ao usuário escrever o código;
- Abrir: abre uma tela apresentando os arquivos que o usuário possui no disco, e permite que o mesmo selecione e abre no editor, códigos escritos em Cafezinho (*.cafe);
- Reabrir: apresenta uma lista, contendo os últimos cinco arquivos que foram abertos;
- Fechar: fecha o documento atual se este se encontrar salvo, caso contrário, solicita ao usuário o salvamento do documento;
- Salvar: salva o documento atual no disco, caso esse ainda não foi previamente salvo, apresenta a tela solicitando onde o usuário deseja salva-lo;
- Salvar Como: abre a tela, solicitando ao usuário, o local no disco que ele deseja salvar o documento, atualmente selecionado;
- Sair: fecha o ambiente CafezinhoIDE, caso exista algum documento não salvo, abre a tela, solicitando ao usuário o salvamento do mesmo.

5.2.2 Menu Editar

No menu editar encontramos todas as funcionalidade de edição do documento. Temos:

- Copiar: copia o texto selecionado no documento para a área de transferência;
- Colar: cola no documento o texto, se esse, existir na área de transferência;
- Recortar: recorte o texto selecionado no documento para a área de transferência;
- Desfazer: desfaz a última ação feita pelo usuário;
- Refazer: refaz a última ação feita pelo usuário;
- Duplicar: Duplica o texto da linha que o cursor está atualmente, e o insere na linha logo abaixo.

5.2.3 Menu Pesquisar

No menu Pesquisar encontramos as funcionalidades de busca e substituição de palavras. Temos:

- Localizar: localiza e realça a ocorrências da palavra ou frase informada pelo usuário, como pode ser visto na figura 5.5. Esta busca pode ser feita:
  - Ignorando a diferença entre maiúsculo e minúsculo;
  - procurando por palavra ou frase que combine no documento de forma exclusiva.



Figura 5.5: *Localizar*

- Localizar Próximo: realça a próxima ocorrência da palavra ou frase informada;
- Localizar Anterior: realça a ocorrência anterior da palavra ou frase informada;
- Substituir: busca no documento, por ocorrência da palavra ou frase informada e substitui-a por outra, também informada pelo usuário. Como apresentado a figura 5.6;
- Ir para linha: move o cursor, no documento atualmente selecionado, para a linha que o usuário informar.



Figura 5.6: *Substituir*

5.2.4 Menu Ver

No menu ver podemos mudar algumas características visuais do editor de texto, são elas:

- Barra de ferramentas: habilita ou desabilita a barra de ferramentas;
- Barra de Status: habilita ou desabilita a barra de status;
- Realce a sintaxe: habilita ou desabilita a funcionalidade de realce das palavras chaves no texto;
- Número de linhas: habilita ou desabilita a funcionalidade de enumerar as linhas do documento;

- Depurador: habilita ou desabilita a área de variáveis;
- Execução do Programa: habilita ou desabilita a área de execução do programa;
- Reiniciar: reiniciar a interface gráfica, para a forma padrão, isto é, a de todas as opções acima citadas habilitadas.

5.2.5 Menu Execução

No menu execução encontramos as funções para executar o programa ou depurar o código, assim temos:

- Executar: compila o código do documento atual selecionado e o executa;
- Parar: para a execução do programa;
- Próximo: compila o código do documento atual selecionado e o executa, caso esse ainda não foi compilado e executado, senão, executa a próxima instrução no código. Se a instrução atual for uma chamada de função esse a executa em plano de fundo, ou seja, sem a interação com o usuário;
- Entrar: trabalha da mesma forma que o menu próximo. Porém se a instrução atual for uma chamada de função esse não a executa em plano de fundo, permitindo assim, o usuário acompanhar a execução da mesma passo a passo;
- Continuar: continua a execução do código sem a interação com o usuário a cada instrução até encontrar algum breakpoint, caso exista;
- Ligar/Desligar Breakpoint: inseri ou remove um ponto de parada (*breakpoint*) na linha onde o cursor atualmente se encontra.

5.2.6 Menu Editor

No menu editor encontramos as funcionalidades referentes ao editor de texto, são elas:

- Fonte: apresenta uma tela com todas as fontes contidas no sistema operacional do usuário;
- Maior: aumenta a fonte no editor;
- Menor: diminui a fonte no editor;
- Reiniciar: Reinicia a fonte e o tamanho no editor para alguma fonte do tipo sans-serif contida no sistema e tamanho de 9px;

5.2.7 Menu Documentos

No menu documento encontra-se as funcionalidades referentes aos documentos e abas no editor, temos:

- Próximo: move para o próximo documento aberto no editor;
- Anterior: retorna para o documento aberto anteriormente no editor;
- Salvar Todos: salva todos os documentos no disco, caso exista algum documento não salvo, solicita ao usuário para salvá-lo;
- Fechar Todos: fecha todos os documentos abertos no editor. Caso exista algum documento não salvo, solicita ao usuário para salvá-lo.

5.2.8 Menu Ajuda

No menu ajuda encontramos informações sobre o programa e sobre a linguagem cafezinho, temos:

- Ajuda CafezinhoIDE: contém explicações sobre o ambiente CafezinhoIDE;
- Tutorial Linguagem Cafezinho: contém informações sobre a linguagem Cafezinho;
- Sobre CafezinhoIDE : apresenta informações sobre o projeto CafezinhoIDE e sobre os autores do projeto.

5.3 Execução

No editor de texto do ambiente CafezinhoIDE, o usuário pode entrar com o código na linguagem Cafezinho. Para poder executar o código o usuário precisa, primeiramente, salvá-lo, caso contrário o compilador não irá executar o código e irá solicitar ao usuário para que o salve, como pode ser visto na figura 5.7.

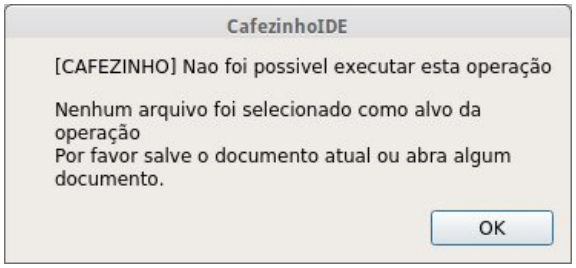


Figura 5.7: *CafezinhoIDE solicitando o salvamento do documento*

O ambiente CafezinhoIDE permite a execução de apenas um arquivo por vez, porém permite trabalhar com vários códigos simultâneos em diferentes abas. Uma vez que um código entra em execução, o ambiente não permite que o usuário edite o código, pois assim o ambiente possui um maior controle sobre o texto, devido a necessidade de indicar a linha em que pode ter ocorrido eventuais erros ou em que esteja sendo analisada em modo de depuração.

O usuário pode, a qualquer momento, interromper a execução do código, assim se houver alguma repetição sem fim de algum bloco do código, o usuário pode pará-lo imediatamente.

5.4 Depurador

O ambiente CafezinhoIDE oferece a funcionalidade de depuração do código, para auxiliar o usuário, que está aprendendo a programar, a entender melhor a execução do programa passo a passo.

Existem três maneiras de habilitar o compilador para executar no modo depurador, são elas:

- Pressionando o botão “Próximo” localizado no menu “Execução”;
- Pressionando o botão “Entra” localizado no menu “Execução”;
- Inserindo um ponto de parada(*breakpoint*) no código e pressionando o botão “Executar” localizado no menu “Execução”.

Ao pressionar o botão “Próximo” o compilador irá compilar o programa, e executar até a primeira instrução do programa, caso não exista erros. Após isso, o ambiente ficará esperando o usuário entrar com a próxima ação. Ao pressionar novamente o botão este irá executar a próxima instrução do código, caso essa seja uma chamada de função, o ambiente irá executá-la em plano de fundo.

O botão “Entrar” apresenta o mesmo comportamento do botão “Próximo”, porém se a instrução a ser executada for uma chamada de função, a máquina virtual não a executa sem a interação do usuário, permitindo-o, assim, que ele a acompanhe.

Os pontos de paradas (*breakpoints*) são marcações feitas no código para indicar ao depurador até qual linha ele deve executar em sequência sem solicitar a intervenção do usuário. Quando o depurador encontra um ponto de parada, ele interrompe a execução do programa e fica em estado de espera por alguma ação do usuário. Quando os pontos de paradas são inseridos no código, o compilador gera código no modo depurador.

O usuário poderá acompanhar a execução passo a passo do código através do realce da linha que está sendo executada no editor, como apresentado na figura 5.8. O usuário também poderá acompanhar os valores das variáveis instanciadas através da área de variáveis, mostrada na figura 5.3.

5.5 Implementação do Depurador na Interface

Parte da implementação do depurador é feita na interface gráfica. Para implementar a função de realçar a linha que será executada, a maior parte do trabalho fica a

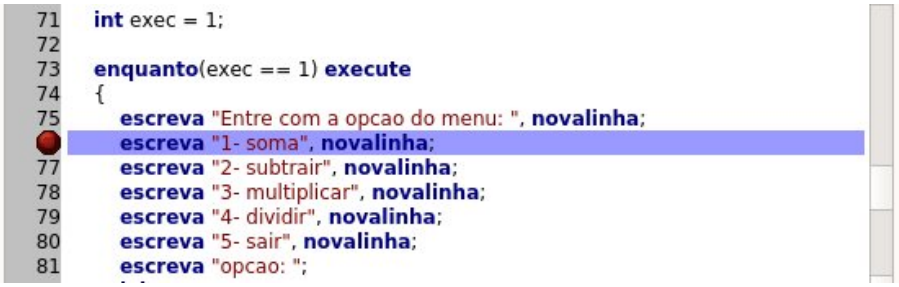


Figura 5.8: *CafezinhoIDE destacando a linha após um breakpoint*

cargo do compilador e da máquina virtual, assim a interface gráfica é responsável apenas por realçar a linha que a máquina virtual a informa.

Assim que uma variável é declarada no código, a interface gráfica a apresenta na "Área de Variáveis" durante a depuração. Caso exista uma chamada de função, a variável da chamada anterior perde o foco e é escondida na "Área de Variáveis", porém não é deletada. Quando seu escopo chega ao fim a variável é removida da área.

Para gerenciar as chamadas de funções foi desenvolvido a classe *GerenciadorVariaveis* que gerencia essas chamadas através de uma pilha de instâncias da classe *GenQuadro*. A classe *GenQuadro*, por sua vez, gerencia todas as variáveis dentro de uma chamada de função através da estrutura de dados hash. Cada variável é representada por uma instância da classe *GenVarEscalar* para indicar uma variável escalar, *GenVarVetorial* para indicar uma variável vetorial e *GenVarVetPonteiro* para indicar uma variável ponteiro.

5.6 Comunicação Máquina Virtual e Interface Gráfica

Durante a execução do código escrito pelo usuário, existem vários momentos, onde, faz-se necessário a sincronização da máquina virtual e interface gráfica, já que elas trabalham em threads separadas e precisam ser sincronizadas.

Para sincronizar as threads, foi utilizada a técnica de programação paralela Monitor [9] que é composto por um *Mutex* (Exclusão Mútua) [6] e *Condition Variables* (Variáveis de Condição). Esta técnica serve para que a máquina virtual pare sua execução e fique esperado uma resposta da interface gráfica. Essa sincronização é necessária nas etapas de:

- inserção e remoção de variável;
- atualização da linha atualmente sendo executada;
- início e fim do escopo de uma função;
- operação de leitura de dados informado pelo usuário.

CAPÍTULO 6

Compilador

Neste capitulo trataremos de todos os aspectos de programação do compilador, além de explicar pontos importantes sobre a geração de códigos para a máquina virtual e para auxiliar o depurador. Para isso, veremos explicações sobre:

- análise Léxica;
- análise Sintática;
- análise Semântica;
- geração de código para a máquina virtual;
- geração de código para o depurador.

6.1 Análise Léxica

A função da análise léxica é identificar no texto informado pelo usuário, sequências de caracteres que formam os símbolos da linguagem, denominados símbolos léxicos ou tokens. A análise léxica é implementada pelo Analisador Léxico, o qual é chamado pelo Analisador Sintático.

O Analisador Léxico é gerado pelo programa Flex no qual são informadas as regras que descrevem os tokens da gramática Cafezinho. O analisador sintático consegue invocar o analisador léxico que foi gerado, através da função *yylex()*.

O programa Flex utiliza variáveis globais para gerenciar seus estados internos, impedindo assim, a sua utilização de modo concorrente ou reentrante. Apesar de ser possível habilitar o Flex para operar no modo reentrante, o CafezenhoIDE não utiliza esta funcionalidade, isto implica na impossibilidade de executar código simultaneamente.

6.2 Análise Sintática

Os tokens separados na análise léxica servem de entrada para o Analisador Sintático. A função da análise sintática é a de determinar a estrutura gramatical dos tokens e verificar se estão de acordo com uma determinada gramática formal.

Para gerar o Analisador Sintático, foi utilizado o programa GNU Bison que recebe as regras gramaticais da linguagem num formato próprio e gera o código que as implementa. O analisador sintático pode ser invocado através da função *yyparser()*. A gramática da linguagem Cafezinho foi apresentada na seção 3.4.

Na etapa de análise sintática é construída a *árvore de sintaxe abstrata*, que é uma abstração do código informado pelo usuário, contendo apenas as informações necessárias para análise semântica e geração de código, esta será melhor explanada logo a seguir.

6.3 Árvore de Sintaxe Abstrato

Para o desenvolvimento da árvore de sintaxe abstrata, foram necessárias modelar em forma de classes da linguagem de programação C++, todas as partes necessárias do código, para as etapas de analise semântica e geração de código, são elas:

- constante inteira, representada pela classe NInteiro;constante inteira, representada pela classe NInteiro;
- constante caractere, representada pela classe NCaracter;
- constante real, representada pela classe NInteiro;
- declaração de variável escalar, representada pela classe NDeclVarEscalar;
- declaração de variável vetorial, representada pela classe NDeclVarVetorial;
- declaração de função, representado pela classe NDeclaracaoFuncao;
- bloco de instruções, representada pela classe NBloco;
- identificador escalar, representado pela classe NIdentificadorEscalar;
- identificador vetorial, representado pela classe NIdentificadorVetorial;
- chamada de função, representada pela classe NChamadaFuncao;
- expressão que é uma instrução, representada pela classe NInstrucaoExpressao;
- operação de atribuição, representada pela classe NAtribuicao;
- operação binária, representada pela classe NOperacaoBinaria;
- operação unária, representada pela classe NOperacaoUnaria;
- operação terciária, representada pela classe NOperacaoTerciaria;
- operação de cast, representada pela classe NCast;
- lista de expressões, representada pela classe NListaExpressao;
- retorne de função, representada pela classe NRetorne;
- operação de leitura, representada pela classe NLeia;
- operação de escreva, representada pela classe NEscreva;
- operação de nova linha, representada pela classe NNovaLinha;
- instrução se, representada pela classe NSe;
- instrução se...senão, representa pela classe NSeSenao;

- instrução enquanto, representada pela classe NEnquanto;
- conjunto de todas as listas de expressões que iniciam um vetor, representada pela classe NInicializadorVetor;
- lista de expressões que iniciam um vetor, representado pela classe NListaInicializador;
- instrução de termino de execução, representado pela classe NTerminar;
- instrução que permite ao programador limpar o terminal de entrada, representado pela classe NLimpar;

O código 6.1 apresenta a seguinte árvore de sintaxe abstrata, nessa árvore a ordem de precedência pai-filho é determinada pela quantidade de hifens, assim o nó que possuir mais hifens é descendente do que possuir menos hifens.

```
1 nulo programa()
2 {
3     int a[4] = {1,2,3,4};
4     a[4] = 10;
5     escreva a[4], novalinha;
6 }
```

Codigo 6.1: *Exemplo Código para gerar a AST*

$|BLOCO, linha: 0$
$|-DECLARACAO\ FUNCAO: "programa", linha: 1$
$|--BLOCO, linha: 2$
$|---DECLARACAO\ VARIAVEL\ VETORIAL: "a", linha: 3$
$|----INTEIRO: 4, linha: 3$
$|---INICIALIZADOR\ VETOR, linha: 3$
$|----LISTA\ INICIALIZADOR, linha: 3$
$|-----INTEIRO: 1, linha: 3$
$|-----INTEIRO: 2, linha: 3$
$|-----INTEIRO: 3, linha: 3$
$|-----INTEIRO: 4, linha: 3$
$|---INSTRUCAO\ EXPRESSAO, linha: 4$
$|----LISTA\ EXPRESSOES, linha: 4$
$|-----ATRIBUICAO, linha: 4$
$|------IDENTIFICADOR\ VETORIAL: "a", linha: 4$
$|--------LISTA\ EXPRESSOES, linha: 4$
$|---------INTEIRO: 4, linha: 4$
$|------INTEIRO: 10, linha: 4$
$|---ESCREVA, linha: 5$
$|----LISTA\ EXPRESSOES, linha: 5$
$|-----IDENTIFICADOR\ VETORIAL: "a", linha: 5$
$|------LISTA\ EXPRESSOES, linha: 5$
$|--------INTEIRO: 4, linha: 5$
$|-----NOVA\ LINHA, linha: 5$

6.4 Análise Semantica

A árvore de sintaxe abstrata serve de entrada para a análise semântica, essa irá percorrer todos os nós da árvore, fazendo as devidas verificações e certificando de que estão de acordo com as regras semânticas da linguagem Cafezinho. Nesta etapa, constrói-se a *tabela de símbolos*. Essa tabela armazena informações sobre variáveis (escalar ou vetorial) e funções declaradas. Quando o Analisador Semântico encontra alguma dessas declarações, ele consulta a tabela para ver se o elemento já foi previamente declarado no mesmo escopo, nesse caso é retornado uma mensagem de erro.

A tabela de símbolos é implementada através da estrutura de dados hash. Cada entrada na tabela irá armazenar uma pilha de instancias da classe *Simbolo*, apresentado no código 6.1. Esta classe armazena o nó da árvore de sintaxe abstrata e a profundidade

(escopo) atual.

Código 6.1 classe Simbolo

```
1  class Simbolo
2  {
3  public:
4      Simbolo(No* no = 0, int profundidade = 0);
5      ~Simbolo();
6      int profundidade;
7      No* no;
8  };
```

A etapa de análise semântica executa ainda outras verificações como:

- avalia uma expressão verificando se o tipo de cada elemento da expressão é compatível com os demais;
- avalia se cada expressão de uma lista de expressões que inicializa um vetor são compatíveis com o tipo do vetor;
- avalia se todos os parâmetros de uma chamada de função são compatíveis com os argumentos contidos na declaração da função chamada;

O algoritmo desenvolvido para fazer a análise semântica utiliza a técnica de programação denominada *recursão* para tornar o código mais simples. Além disso, o algoritmo foi projetado para poder trabalhar no modo reentrante, ou seja, para permitir seu uso de forma concorrente. O código 6.2 apresenta o protótipo da função de análise semântica bem como a tabela de símbolos.

Código 6.2 Protótipo Análise semantico

```
1  typedef QStack<Simbolo*> PilhaSimbolo;
2  typedef QHash<QString, PilhaSimbolo*> TabelaSimbolo;
3  typedef TabelaSimbolo::iterator IteradorTabelaSimbolo;
4  typedef QList<IteradorTabelaSimbolo> Remover;
5  typedef QList<IteradorTabelaSimbolo>::iterator IteratorRemover;
6
7  IteradorTabelaSimbolo analise_semantica(TabelaSimbolo &tabela,
8                                          No* no,
9                                          int profundidade,
10                                         No* funcao=0);
```

6.5 Geração de Código

Passadas as etapas de análise léxica, sintática e semântica e estando o código de acordo com as regras da linguagem Cafezinho, é iniciada a etapa de Geração de Código que gera código para a máquina virtual implementada no ambiente CafezinhoIDE.

Nessa etapa, novamente, é construída uma tabela, denominada *tabela de referência*, porém agora, para armazenar informações importantes para geração de código como: o endereço atribuído a uma variável ou função, se a variável é do tipo escalar ou vetorial ou se a variável é ou não um parâmetro. Essas informações se referem, somente, às declarações de funções e variáveis (escalares ou vetoriais).

A tabela de referência é implementada através da estrutura de dados hash. Cada entrada na tabela irá armazenar uma pilha de instancias da classe *Referencia*, apresentado no código 6.3.

Código 6.3 `classe Referencia`

```
1  class Referencia
2  {
3  public:
4      Referencia( No* origem,
5                  int profundidade = 0,
6                  int offset = 0,
7                  bool parametro = false,
8                  bool vetor = false,
9                  bool variavel = false
10     );
11     Referencia();
12     ~Referencia();
13     int profundidade;
14     int offset;
15     bool parametro;
16     bool vetor;
17     No* origem;
18     //Informacao para o depurador...
19     bool variavel;
20 };
```

Uma parte fundamental da geração de código é a atribuição de endereço as variáveis alocadas. Esta atribuição é feita, incrementando sequencialmente, para cada declaração de variável contida em um bloco, o endereço (*offset*) recebido como parâmetro pela função que faz a geração de código, como pode ser visto na figura 6.1.

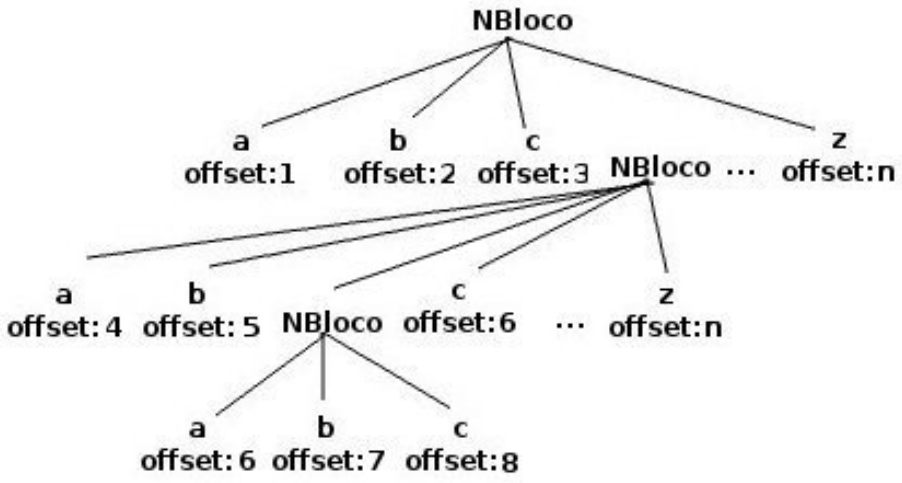


Figura 6.1: *Hierarquia de offsets na declaração de variável*

A função que faz a geração de código utiliza-se da técnica hibrida de máquina de pilha e registradores, para poder avaliar uma expressão. Nessa técnica, todos os elementos da expressão são empilhados de trás para frente exceto o último que é colocado no acumulador (registrador *eax*), então desempilha todos os elementos e executa as devidas operações mantendo o resultado no acumulador, como apresentando na figura 6.2.

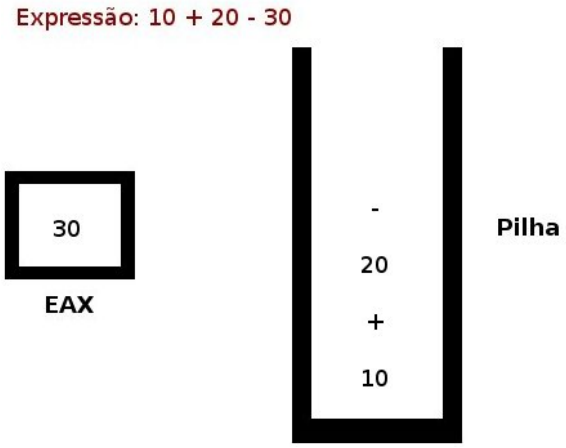


Figura 6.2: *Máquina de Pilha*

Outra parte importante na geração de código é a geração de código de uma declaração de função, chamada de função e retorno de função, conhecida como convenção de chamada (*Calling Convention*).

Na geração de código de uma chamada de função, a função chamadora empilha todos os argumentos em ordem inversa, antes porém, empilha o valor do registrador base da pilha (*bp*) atual. Por último, invoca a função chamada. O estado da pilha de execução após uma chamada é apresentado na figura 6.3.

Na geração de código de declaração de função, empilha o valor do registrador endereço de retorno (*er*) que foi inserido pela função chamadora ao invocá-lo. Atualiza



Figura 6.3: *Estado da pilha após uma chamada de função*

então, o registrador de base de pilha para o endereço do ponteiro de pilha (*pp*) atual.

Na geração de retorno de função, o valor retornado é movido para o acumulador (*eax*), copia-se o valor do registrador base de pilha (*bp*) para o registrador ponteiro de pilha (*pp*), desempilha o registrador de retorno de função (*er*) e o restaura, desempilha todos os argumentos que foram inseridos pela função chamadora e desempilha e restaura o registrador base de pilha (*bp*).

O algoritmo desenvolvido para fazer a geração de código, assim como na análise semântica, utilizou-se da técnica de programação denominada *recursão*, e foi projeto, para poder trabalhar no modo reentrante. O protótipo do algoritmo de geração de código, bem como a tabela de referências é apresentado no código 6.4.

Código 6.4 Protótipo geração de código

```
1  typedef QStack<Referencia> PilhaRef;
2  typedef QHash<QString, PilhaRef> TabelaRef;
3  typedef QHash<QString, PilhaRef>::iterator IteradorTabelaRef;
4  typedef QList<IteradorTabelaRef> RemoverRef;
5  typedef QList<IteradorTabelaRef>::iterator IteradorRemoverRef;
6
7  void gerar_codigo(MaquinaVirtual &vm,
8                    TabelaRef &tabela,
9                    No *no,
10                   int profundidade,
11                   int offset,
12                   No *funcao = 0);
```

O trecho de código apresentando no código 6.1 é convertido para o seguinte assembly da máquina virtual, seguindo a nomenclatura definida no apêndice B.

1 : *sp bp*, 0[*pp*]
2 : *adc pp*, *pp*, 1
3 : *inv* 5
4 : *parar*
5 : *mov bp*, *pp*
6 : *sp er*, 0[*pp*]
7 : *adc pp*, *pp*, 1
8 : *adc pp*, *pp*, 4
9 : *mov eax*, 1
10 : *sp eax*, 1[*bp*]
11 : *mov eax*, 2
12 : *sp eax*, 2[*bp*]
13 : *mov eax*, 3
14 : *sp eax*, 3[*bp*]
15 : *mov eax*, 4
16 : *sp eax*, 4[*bp*]
17 : *mov eax*, 10”
18 : *mov ebx*, *eax*
19 : *sp ebx*, 0[*pp*]
20 : *adc pp*, *pp*, 1
21 : *mov eax*, 4
22 : *adc ecx*, *bp*, 1
23 : *adc eax*, *ecx*, *eax*
24 : *sub pp*, *pp*, 1
25 : *cp ebx*, 0[*pp*]
26 : *sp ebx*, 0[*eax*]
27 : *mov eax*, 4
28 : *mov ebx*, *eax*
29 : *adc eax*, *bp*, 1
30 : *adc eax*, *eax*, *ebx*
31 : *cp eax*, 0[*eax*]
32 : *eco eax*
33 : *sis novalinha*
34 : *sub pp*, *pp*, 4
35 : *mov pp*, *bp*
36 : *cp er*, 0[*pp*]
37 : *sub pp*, *pp*, 1
38 : *cp bp*, 0[*pp*]
39 : *ret er*

6.6 Geração de Código para Depurador

Quando o usuário deseja, ele pode colocar o compilador no modo depurador, este por sua vez, gera código para permitir acompanhar a execução do programa escrito. O compilador gera instruções complementares que servem como "marcadores" no código, e permitem comunicar a máquina virtual quais ações, ela deve tomar.

A instrução *IDebugPasso*, apresentado o protótipo no código 6.5, marca o inicio de cada instrução, e indicam a máquina virtual que essa deve, por sua vez, comunicar a interface gráfica para atualizar a marcação da linha que está sendo executado, como foi mostrado na figura 5.8.

Código 6.5 `classe IDebugPasso`

```
1 class IDebugPasso : public Instrucao
2 {
3 public:
4     IDebugPasso(int linha);
5     void execute(MaquinaVirtual &vm);
6     TipoInstrucao::TipoInstrucao tipoInstucao();
7     int linha;
8 };
```

As classes *IDebugVariavelEmpilha*, apresentado o protótipo no código 6.6, e *IDebugVariavelDesempilha*, apresentado o protótipo no código 6.7, servem para indiciar, respectivamente, o inicio e o fim do escopo de uma variável. Quando a máquina virtual os encontra, esse comunica a interface gráfica, para inseri-los ou remove-los da área de variáveis, como mostrado na figura 5.3.

Código 6.6 `classe IDebugVariavelEmpilha`

```
1  class IDebugVariavelEmpilha : public Instrucao
2  {
3  public:
4      IDebugVariavelEmpilha(No* no,
5                            int offset,
6                            int profundidade,
7                            No* pno = NULL);
8      void execute(MaquinaVirtual &vm);
9      TipoInstrucao::TipoInstrucao tipoInstucao();
10     No* no;
11     No* pno;
12     int offset;
13     int profundidade;
14 };
```

Código 6.7 `classe IDebugVariavelDesempilha`

```
1  class IDebugVariavelDesempilha : public Instrucao
2  {
3  public:
4      IDebugVariavelDesempilha(No* no);
5      void execute(MaquinaVirtual &vm);
6      TipoInstrucao::TipoInstrucao tipoInstucao();
7      No* no;
8  };
```

Para controlar a chamada de função, as classes *IDebugEmpilhaExec*, apresentado o protótipo no código 6.8, e *IDebugDesempilhaExec*, apresentado o protótipo no código 6.9, servem para indicar, respectivamente, o inicio e o fim de uma chamada de função e permitem que a máquina virtual e interface gráfica ,trabalhem em sincronismo para executar, em plano de fundo, quando o usuário assim requisitar.

Código 6.8 `classe IDebugEmpilhaExec`

```
1 class IDebugEmpilhaExec : public Instrucao
2 {
3 public:
4     IDebugEmpilhaExec();
5     void execute(MaquinaVirtual &vm);
6     TipoInstrucao::TipoInstrucao tipoInstucao();
7 };
```

Código 6.9 `classe IDebugDesempilhaExec`

```
1 class IDebugDesempilhaExec : public Instrucao
2 {
3 public:
4     IDebugDesempilhaExec();
5     void execute(MaquinaVirtual &vm);
6     TipoInstrucao::TipoInstrucao tipoInstucao();
7 };
```

CAPÍTULO 7

Expansões e Melhorias

Neste capítulo, apresentaremos algumas possíveis melhorias ao projeto CafezinhoIDE.

7.1 Melhorias na Interface Gráfica

A interface gráfica do projeto CafezinhoIDE não foi criado a partir de uma identidade virtual própria, o qual precisaria de um profissional da área de design para poder criá-la. O projeto precisa de uma remodelagem na interface, seguindo um padrão visual definido por um profissional.

É importante notar que essa interface gráfica deve ter um visual moderno, pois chama a atenção do aluno que está iniciando sua aprendizagem em programação. Outro ponto importante é a inserção de elementos gráficos como: ícones e botões com desenhos, para tornar o ambiente com uma aparência profissional.

Para facilitar o controle do código conforme este se expande, criar uma área apresentando todas as funções declaradas e suas respectivas variáveis, ao passo que, quando o usuário selecionar algum desses elementos, o cursor move-se para a linha onde ele foi definido.

7.2 Melhorias no Editor CafezinhoIDE

O editor cafezinho poderia ser melhorado para ter novas funcionalidades como:

- fonte personalizada para o editor, tornando a legibilidade melhor;
- permitir alterar a cor do realce das palavras reservadas da linguagem Cafezinho para tornar o sistema mais personalizável;
- implementar a verificação, em quando o usuário ainda está digitando, da análise léxica, sintática e semântica do código, indicando com uma marcação em vermelho, de possíveis erros encontrado no código. Está técnica é muito utilizada pelos principais editores do mercado;

- implementar a funcionalidade de mostrar ou esconder blocos de código, apresentando um símbolo de [+] no canto do editor para mostrar o código escondido e [-] para esconder o bloco;
- trabalhar com outras codificações de texto.

7.3 Melhorias no Compilador e na Linguagem Cafezinho

O compilador Cafezinho pode ser expandido para trabalhar com múltiplos arquivos, permitindo assim, que o usuário possa quebrar o programa, facilitando e organizando o desenvolvimento. Essa capacidade é interessante, pois ensina o aluno a segmentar o código, o que é muito importante em projetos grandes.

Outro ponto seria a implementação do compilador, para poder trabalhar com múltiplos arquivos de forma simultânea, configurando a ferramenta Flex para trabalhar de forma reentrante. Os algoritmos desenvolvidos para o analisador semântico e geração de código, já trabalham de forma reentrante, assim, seriam necessários apenas algumas modificações no projeto original.

A gramática da linguagem Cafezinho poderia ser expandida para trabalhar com mais estruturas como:

- de repetição: faça...enquanto, para...faça, execute...enquanto
- de seleção: selecione...caso...
- de salto: pare, continue;

Outros elementos poderiam ser inseridos como ponteiros e estruturas.

Todos esses visando alunos que possuem um maior conhecimento em programação.

Uma funcionalidade muito importante que não foi inserida na geração de código é a capacidade que avaliar uma expressão booleana e determinar a menor expressão que determina se a expressão como um todo é verdadeira ou falsa, está técnica é conhecida como avaliação de curto circuito (*Short-circuit evaluation*).

CAPÍTULO 8

Conclusão

Neste trabalho foi abordado e explanado, todos os aspectos do projeto CafezinhoIDE, da criação da interface gráfica e suas funcionalidades, dos aspectos de implementação do compilador Cafezinho, de como é feito a geração e interpretação do código, de como o depurador funciona e de como a interface gráfica e a máquina virtual se comunicam.

O projeto CafezinhoIDE oferece um ambiente de programação agradável, criando um editor com funcionalidades modernas, presente nos principais editores comerciais. O projeto minimiza os aspectos que podem dificultar a aprendizagem de programação, com uma linguagem de programação simples, porém poderosa e soluciona o problema da interpretação das mensagens de erros emitidas pelo compilador que geralmente são em ingles e com jargão, com mensagens simples e em português.

Porém o projeto não esconde do aprendiz a dificuldade de expressar a solução de um problema em uma linguagem formal, por que esta é a parte fundamental na aprendizagem de programação e não deve ser escondida.

Este trabalho, também foca em sua expansão, sugerindo formas de melhoria do projeto. Além de servir como manual a outro alunos que possam continuar a trabalhar no projeto.

Ao termino, oferece ao aprendiz de programação, uma introdução a linguagem Cafezinho e a programação, explicando todos os detalhes com códigos e figuras para facilitar a aprendizagem do aluno.

Este trabalho se concentrou no projeto e na implementação da IDE Cafezinho. Não foi objetivo desse experimentar a IDE no ensino de programação. Entretanto, reconhecemos que essa experimentação é de fundamental importância para avaliar o quanto a IDE Cafezinho se compatibiliza as práticas didáticas e para levantar requisitos de alteração e melhoria na IDE. Portanto, sugerimos como trabalho futuro a investigação do uso da IDE Cafezinho em disciplinas que ensinam o desenvolvimento de algoritmos e de programação de computadores.

Referências Bibliográficas

[1] **Site official do chide**, July 2014.

[2] **Site official do qt**, July 2014.

[3] **Site official do sublime text editor**, July 2014.

[4] **Site official do visual studio**, July 2014.

[5] ANTÓNIO MANSO, LUÍS OLIVEIRA, C. G. M. **Ambiente de Aprendizagem de Algoritmos Portugol IDE**. 2009.

[6] DIJKSTRA, E. W. **Solution of a problem in concurrent programming control.** *Commun. ACM*, 8(9):569, 1965.

[7] GOSLING, J.; MCGILTON, H. **The Java Language Environment**. Sun Microsystems Computer Company, May 1995.

[8] HARVEY, B. **Computer Science Logo Style**. MIT Press, 1997.

[9] HOARE, C. A. R. **Monitors: An operating system structuring concept.** *Commun. ACM*, 17(10):549–557, 1974. Corrigendum: CACM 18(2): 95 (1975).

[10] KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language Second Edition**. Prentice-Hall, Inc., 1988.

[11] LEVINE, J. R. **flex and bison - Unix text processing tools.** O'Reilly, 2009.

[12] MIT. **Scratch**, 2007.

[13] STROUSTRUP, B. **The C++ programming language**. Addison-Wesley, 3rd edition, 1997.

APÊNDICE A

Linguagem de Programação Cafezinho

A linguagem de programação cafezinho é simples e fácil. Começamos pelo famoso *Hello World*, como pode ser visto no Código A.1:

```
1 nulo programa()
2 {
3     escreva "Hello World!"; /*Isto é um comentário*/
4 }
```

Codigo A.1: *Hello World em Cafezinho*

Na linha 1 declaramos a função principal da linguagem chamada de *programa()*. É fundamental sua declaração e é a partir dela que se inicia a execução de todo o programa escrito em Cafezinho, obrigatoriamente essa função não possui nenhum parâmetro. Na linha 3 encontramos a função escreva, essa função é um comando pertencente a linguagem, responsável por imprimir na tela o parâmetro passado. Posteriormente falaremos mais sobre os comandos da linguagem. Cada instrução na linguagem Cafezinho é separada por ";", o qual indica o final da instrução. A saída do Código A.1 é apresentando na figura **??**.

Observe também que, o texto entre "/*" e "*/" representa um comentário, ou seja, o compilador irá ignorar todo o seu conteúdo, sendo ele visível apenas para o programador. É uma boa prática de programação comentar as instruções do código.

Aprofundemos agora nos aspectos de programação e na linguagem Cafezinho.



Figura A.1: *Saída do Programa Hello World*

Aprofundemos agora mais nos aspectos de programação e na linguagem cafezinho.

A.1 Variáveis

Na programação uma variável contém uma referência à um endereço localizado na memória que pode armazenar algum valor dentro de um domínio estabelecido. O programador não precisa saber qual é o endereço da variável, mas pode acessá-la através de um nome definido durante sua declaração. Uma representação gráfica de uma variável é apresentada na figura A.2.

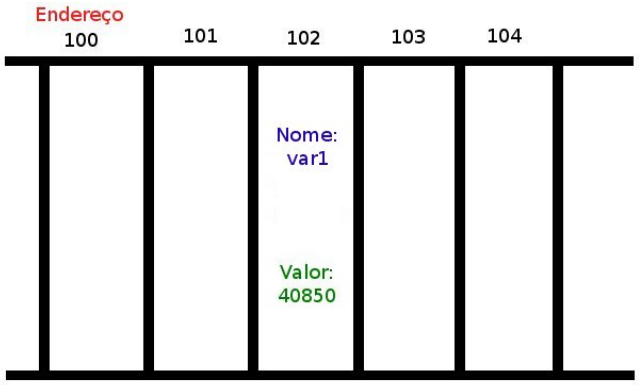


Figura A.2: *Representação gráfica da variavel var1*

A linguagem Cafezinho admite três tipos de dados distintos:

- **inteiro**: Representa um número inteiro com domínio de -2,147,483,648 à 2,147,483,647.
- **car**: Representa um caractere, mas pode também ser expressado através do número correspondente da tabela ASCII, seu domínio vai de -128 à 127 ou de 0 à 255.
- **real**: Representa um número real com domínio de 2.3E-308 a 1.7E+308. Observe que em Cafezinho usa-se a notação norte-americana, assim ao invés de vírgula usa-se ponto para representar a parte fracionária do número real.

Na linguagem também existe um quarto tipo chamado nulo porém ele não pode ser alocado e serve apenas para indicar que uma função não retorna tipo. Falaremos mais sobre isso posteriormente.

A declaração de variáveis na linguagem é exemplificada no Código A.2. Observe que as variáveis “nome4” e “nome5” possuem o mesmo tipo da variável “nome3” e as três formam uma única instrução, porém “nome1” e “nome2” não possuem o mesmo tipo e cada uma forma uma instrução diferente.

```
1 nulo programa()
2 {
3     /* Declaração de variável */
4     car nome1;
```

```
5       real nome2;
6       /* Ou */
7       int nome3, nome4, nome5
8       /*
9         Declare mais variáveis
10        colocando , nome_variavel
11      */ ;
12
13  }
```

Codigo A.2: *Declaração de variável*

Durante a declaração de uma variável, é possível inicializar seu valor, um exemplo disso pode ser visualizado no Código A.3.

```
1   nulo programa()
2   {
3       /* Declaração de variável */
4       car nome1 = 'a';
5       real nome2 = 4.345;
6       int nome3 = 10, nome4 = 20, nome5 = 30;
7   }
```

Codigo A.3: *Inicializando variáveis na Declaração*

A.1.1 Cast

Na linguagem Cafezinho uma variável com um determinado tipo pode receber valores de variáveis de tipos com tamanhos menores, porém o contrário não é possível, assim real pode receber real, inteiro e car; inteiro pode receber inteiro e car; porém car só pode receber car porque não existe outro tipo com tamanho menor.

Entretanto, na linguagem Cafezinho existe a operação de *cast*, exemplificada no Código A.4. Essa operação permite que uma variável receba valores de variáveis com tamanho maior, porém a parte que ultrapassa o contêiner menor é descartado, por exemplo, se um inteiro receber um real, a parte flutuante é descartada. Para efetuar um cast basta fazer: (**tipo_cast**) nome_variável.

```
1   nulo programa()
2   {
3       car c; int i; real r;
4
5       c = 'a'; /* Ok */
6       i = 10; /* Ok */
7       r = 10.123; /* Ok */
8
9       r = i; /* Ok */
```

```
10      r = c; /* Ok */
11
12      i = c; /* Ok */
13
14      c = r; /* Erro! */
15      c = (car) r;
16      /* Ok! será efetuado um cast */
17      i = r ; /* Erro! */
18      i = (int) r;
19      /* Ok! será efetuado um cast */
20  }
```

Codigo A.4: *Operação de Cast*

A.2 Vetores e Matrizes

Imagine o seguinte problema: deseja-se salvar as três notas de Pedro e Maria e imprimir o cálculo das médias. Esse problema é resolvido no Código A.5 e a saída é mostrada na Figura A.3.

```
1   nulo programa()
2   {
3       int pedro_nota1, pedro_nota2, pedro_nota3;
4       int maria_nota1, maria_nota2, maria_nota3;
5
6       pedro_nota1 = 10;
7       pedro_nota2 = 3;
8       pedro_nota3 = 6;
9
10      maria_nota1 = 10;
11      maria_nota2 = 7;
12      maria_nota3 = 4;
13
14      escreva "media de pedro: ";
15      escreva (pedro_nota1 + pedro_nota2 + pedro_nota3)/3;
16      novalinha;
17
18      escreva "media de maria: ";
19      escreva (maria_nota1 + maria_nota2 + maria_nota3)/3;
20      novalinha;
21  }
```

Codigo A.5: *Problema: pedro e maria - parte 1*

Agora imagine o problema de ter que se armazenar 10 notas, seria, então, necessário criar mais 20 variáveis. Imagine agora que ao invés de serem dois alunos,

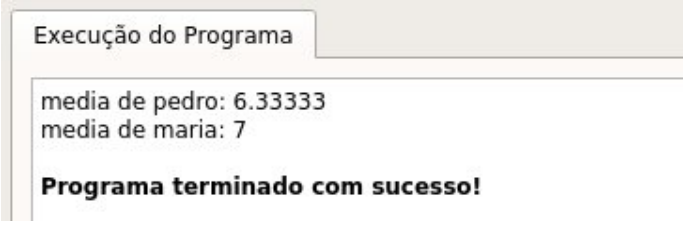


Figura A.3: *Saída do Problema: pedro e maria - parte 1*

fossem 10 alunos, para isso seria necessário criar 100 variáveis e muita criatividade para nomear tantas variáveis.

A.2.1 Vetores

Para tentar solucionar o problema mencionado anteriormente, a linguagem oferece a capacidade de se criar vetores. Um vetor é idêntico a uma variável a respeito de sua funcionalidade de armazenar valores, mas ao invés de se alocar um espaço de memória, são alocados quantos espaços contínuos forem necessários. Os valores podem ser acessados através de índices, onde os índices são enumerados a partir de zero. Todos os elementos de um vetor possuem o mesmo tipo. Sua representação é feita na Figura A.4.



Figura A.4: *Representação gráfica do vetor a[]*

Exemplos de declarações de vetores são mostrados no Código A.6. Assim como para variáveis, vetores podem ser declarados em uma mesma linha utilizando-se uma vírgula para separar diferentes alocações de vetores e, além disso, também é possível intercalar vetores e variáveis durante a declaração.

```
1  nulo programa()
2  {
3      /* vetor tipo int*/
4      int a[10];
5
6      /* vetor tipo real */
7      real b[20];
8
9      /* vetor tipo car */
10     car c[30], d[100], f[200];
```

```
11
12      /* intercalação variável e vetor */
13      int k, v[100];
14
15      /* intercalação vetor e variável*/
16      car nome[100], letra;
17  }
```

Codigo A.6: *Declaração de vetor*

É possível inicializar um vetor da mesma forma que variáveis, porém circundando os valores entre chaves, como apresentado no Código A.7.

```
1  nulo programa()
2  {
3      /* vetor tipo int*/
4      int a[10] = {0,1,2,3,4,5,6,7,9};
5      car texto[5] = {'t','e','x','t','0'};
6  }
```

Codigo A.7: *Inicialização de vetor*

O problema descrito anteriormente pode ser reescrito como no Código A.8. Observe que o vetor "pedro" é declarado com dimensão 3, mas só é possível acessar as posições 0, 1 e 2, isto é, o índice do vetor é enumerado a partir de zero, tentar acessar a posição "pedro[3]" geraria um erro. Este é um erro muito comum para iniciantes e, às vezes, até para os profissionais esquecidos.

```
1  nulo programa()
2  {
3      int pedro[3];
4      int maria[3];
5
6      /*
7        Podemos acessar o elemento no vetor
8        fazendo nome_variavel[indice] = valor
9      */
10     pedro[0] = 10;
11     pedro[1] = 3;
12     pedro[2] = 6;
13
14     maria[0] = 10;
15     maria[1] = 7;
16     maria[2] = 4;
17
18     escreva "media de pedro: ";
19     escreva (pedro[0] + pedro[1] + pedro[2])/3;
20     novalinha;
```

```
21
22      escreva "media de maria: ";
23      escreva (maria[0] + maria[1] + maria[2])/3;
24      novalinha;
25  }
```

Codigo A.8: *Problema: pedro e maria - parte 2*

A.2.2 Matrizes

A linguagem cafezinho oferece ainda um outro tipo de variável conhecido como matriz. Uma matriz é semelhante a usada na matemática básica, isto é, um conjunto de elementos que podem ser acessados através da linha e coluna, como na Figura A.5. Entretanto, na linguagem Cafezinho pode-se ter mais dimensões do que linhas e colunas.

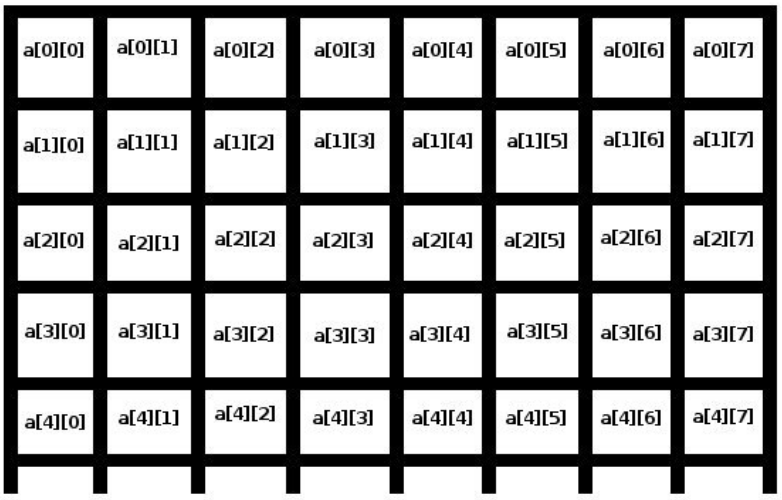


Figura A.5: *Representação gráfica da matriz a[][]*

Assim como para vetores, índices de matrizes são enumerados a partir de zero. Declarações de matrizes na linguagem Cafezinho são exibidas no Código A.9.

```
1  nulo programa()
2  {
3      /* vetor tipo int*/
4      int a[10][10];
5
6      /* vetor tipo real */
7      real b[20][2];
8
9      /* intercalação variável, vetor, matriz */
10     int k, v[100], d[20][30];
11
```

```
12      /* matriz de várias dimensões*/
13      int nome[100][30][3][1];
14
15      /*
16        Podemos acessar o elemento no vetor
17        fazendo nome_variavel[indice][indice]... = valor
18      */
19
20      nome[40][10][2][0] = 100;
21
22      d[19][29] = 45;
23  }
```

Codigo A.9: *Declaração de matriz*

As matrizes também podem ser inicializadas em sua declaração para cada dimensão. Seus elementos devem ser circundados entre chaves e todos os elementos daquela dimensão também devem ser circundados entre chaves, isso é exemplificado no Código A.10.

```
1   nulo programa()
2   {
3       int valores1 [2][2] = {
4                                 {1,2},
5                                 {3,4}
6                               };
7       int valores2[3][2][2] = {
8                                     {
9                                         {1,2},{3,4}
10                                    },
11                                    {
12                                        {5,6},{7,8}
13                                    },
14                                    {
15                                        {9,10},{11,12}
16                                    }
17                                };
18  }
```

Codigo A.10: *Inicialização de matriz*

Um vetor é, na verdade, uma matriz linha e não existe diferenciação explícita na linguagem Cafezinho entre ambos, portanto tudo o que se aplica a uma matriz aplica-se também a um vetor e vice-versa. Usaremos apenas o termo matriz.

A.3 Operações Aritméticas e Lógicas

A principal função de um programa de computador é transformar dados de entrada em dados de saída. Para tal faz-se necessário a utilização de operações matemáticas e lógicas sobre os dados. Na linguagem Cafezinho operadores trabalham sobre tipos de dados específicos e retornam tipos específicos de dados. Apresentaremos as operações que a linguagem Cafezinho suporta e os tipos de dados que trabalham.

A.3.1 Soma, Substração e Multiplicação

A linguagem Cafezinho oferece os operadores de soma(+), substração(-) e multiplicação(*). Esses operadores trabalham sobre qualquer tipo de dado (int, car, real) e retornam o tipo do maior contêiner, por exemplo, se somarmos um int a um real a operação retorna um valor do tipo real. Exemplos dessas operações são mostradas no Código A.11.

```
1  nulo programa()
2  {
3      int a=10, resultado1;
4      real b = 20.23, resultado2
5      car c = 'c', resultado3;
6      /*
7        variavel c contém
8        o valor letra c
9      */
10
11     resultado2 = b + a;
12     /* b + a retorna real */
13
14     resultado1 = a * c;
15     /* a * c retorna real */
16
17     resultado3 = (car)(a - c);
18     /* precisa dar um cast! */
19
20     resultado2 = (a+b)*c - c;
21     /* pode fazer expressões complexas */
22
23 }
```

Codigo A.11: *Soma, Substração e Multiplicação*

A.3.2 Divisão e Potência

A linguagem oferece também o operador de divisão (/), além de oferecer um operador que poucas linguagens oferecem, o operador de potência (**). Ambos operadores trabalham com qualquer tipo de dado, mas retornam apenas valores de tipo real. Divisão e potência são exibidas no Código A.12.

```
1  nulo programa()
2  {
3      int a=10, resultado1;
4      real b = 20.23, resultado2
5      car c = 'c', resultado3;
6
7      resultado2 = a**c;
8      /* a(10) elevado a c('c' = 99 em ascii )*/
9
10     resultado1 = (int)(a/2);
11
12     resultado2 = b/c;
13
14     resultado2 = (b**10 + 2)/2 - c;
15     /* pode fazer expressões complexas */
16
17 }
```

Codigo A.12: *Divisão e Potência*

A.3.3 Resto da divisão, Shift Esquerdo e Shift Direito

Para completar, a linguagem Cafezinho oferece, também, o operador de resto de divisão (%), shift esquerdo («) e shift direito (»), eles trabalham exclusivamente sobre operadores do tipo int e retornam exclusivamente valores de tipo inteiro. O operador de shift esquerdo desloca os bits do número para a esquerda dependendo da quantidade de casas desejada. O operador de shift a direita faz o mesmo, mas deslocando para a direita. Essas operações são mostradas no Código A.13.

```
1  nulo programa()
2  {
3      int a=10, resultado1;
4      real b = 20.23, resultado2
5      car c = 'c', resultado3;
6
7      resultado2 = a%((int)(b));
8
9      resultado1 = a%2;
10
```

```
11      resultado2 = b/c;
12
13      resultado1 = 2<<1;
14
15      resultado2 = 32>>5;
16
17      resultado2 = ((b**10 + 2)/2 - c)%2;
18      /* pode fazer expressões complexas */
19
20  }
```

Codigo A.13: *Resto, Shift Esquerdo e Direito*

A.3.4 Operadores Lógicos

A linguagem Cafezinho também oferece operadores lógicos seguindo as regras de Boole, mas não oferece o tipo lógico de dados, assim zero representa valor falso e qualquer número diferente de zero representa valor verdadeiro. A linguagem oferece:

- E lógico (**e** ou &&);
- Ou Lógico (**ou** ou ||);
- Não Lógico (~ ou !);
- Maior (>);
- Menor (<);
- Maior ou igual (>=);
- Menor ou igual (<=);
- igual (==);
- diferente (!=);
- E binário (&): Este operador executa operação lógica E bit a bit;
- Ou binário (|): Este operador executa operação lógica OU bit a bit;
- Ou Exclusivo (^): Este operador executa operação lógica XOR bit a bit;

A linguagem cafezinho não oferece a funcionalidade de Curto Circuito, portanto a expressão sempre será avaliada até o final. As operações lógicas são mostradas no Código A.14.

```
1  nulo programa()
2  {
3      int result;
4
5      result = 10 > 12;
6      /* result é igual a 0 (falso) */
7
```

```
8       result = 10 < 12;
9       /* result é igual a 1 (verdadeiro) */
10
11      result = 10 <= 12;
12      /* result é igual a 1 (verdadeiro) */
13
14      result = 10 >= 12;
15      /* result é igual a 0 (falso) */
16
17      result = 0 ou 1;
18      /* result é igual a 1 (verdadeiro) */
19
20      result = 0 e 1;
21      /* result é igual a 0 (falso) */
22
23      result = 0 == 1;
24      /* result é igual a 0 (falso) */
25
26      result = 0 != 1;
27      /* result é igual a 1 (verdadeiro) */
28
29      result = 32 | 2;
30      /* result é igual a 34 */
31
32      result = 32 & 2;
33      /* result é igual a 0 */
34
35      result = 32 ^ 2;
36      /* result é igual a 8 */
37
38  }
```

Codigo A.14: *Operações Lógicas*

A.3.5 Operadores de Atribuição

A linguagem cafezinho também oferece outros operadores de atribuição além de "=", como apresentado no Código A.15. Este operadores são compostos da seguinte forma: "variável operação = expressão" e é equivalentes à "variável = variável operação expressão". Temos:

- var += exp: equivalente à var = var + exp;
- var -= exp: equivalente à var = var - exp;
- var *= exp: equivalente à var = var * exp;
- var /= exp: equivalente à var = var / exp;

- var %= exp: equivalente à var = var % exp;
- var **= exp: equivalente à var = var ** exp;
- var »= exp: equivalente à var = var » exp;
- var «= exp: equivalente à var = var « exp;
- var |= exp: equivalente à var = var | exp;
- var &= exp: equivalente à var = var & exp;
- var ^= exp: equivalente à var = var ^ exp;

```
1  nulo programa()
2  {
3      real result1 = 2;
4      int result2 = 2;
5
6      result1 += 10;
7      /* result1 = result1 + 10 */
8
9      result1 -= 10;
10     /* result1 = result1 - 10 */
11
12     result1 *= 10;
13     /* result1 = result1 * 10 */
14
15     result2 /= 10;
16     /* result2 = result2 / 10 */
17
18     result1 %= 10;
19     /* result1 = result1 % 10 */
20
21     result2 **= 10;
22     /* result2 = result2 ** 10 */
23
24     result1 >>= 10;
25     /* result1 = result1 >> 10 */
26
27     result1 <<= 10;
28     /* result1 = result << 10 */
29
30     result1 |= 10;
31     /* result1 = result | 10 */
32
33     result1 &= 10;
34     /* result1 = result & 10 */
35 }
```

Codigo A.15: *Operadores de Atribuição*

A.3.6 Operador Incremento e Decremento

A linguagem cafezinho também oferece o operador de incremento(++) e decremento(–) que podem ser usado de forma prefixa (++var ou –var) ou pós-fixa (var++ ou var–) que são equivalentes à var = var + 1 e var = var - 1 respectivamente. A diferença do operador pós-fixo para o prefixo é que a operação de atribuição daquela variável só será realizado depois desta operação ao contrário do prefixo, como no Código A.16.

```
1  nulo programa()
2  {
3      int a = 2, result;
4
5      result = ++a + 3;
6      /*
7        result é igual a 6 por que
8        primeiro incrementa a (a=3)
9        então faz a(3) + 3 = 6
10     */
11
12     a = 2;
13
14     result = a++ + 3;
15
16     /*
17       result é igual a 5 por que
18       primeiro faz a(2) + 3 = 5
19       depois incrementa a(a=3)
20     */
21 }
```

Codigo A.16: *Operadores de Atribuição*

A.3.7 Precedência de operadores

Na linguagem Cafezinho existe precedência entre operadores, essa regra é explicada na Seção 3.3.

A.4 Estrutura Condicional

Diariamente tomamos determinadas ações diante da ocorrência de certos eventos, assim também acontece na programação. Para proporcionar ao mundo da programação essa necessidade de agir diante de um evento, existem as *Estruturas Condicionais*, que realizam o papel de regimentar estas situações onde é necessário tomar alguma atitude.

A linguagem cafezinho, por sua vez, oferece dois tipos destas *Estruturas Condicionais*: **se** (*condição*) **entao** *ação* ; e **se** (*condição*) **entao** *ação* **senao** *ação*. Para a demonstração destas instruções, iremos reescrever o Código A.5, o qual pode ser visto no código A.17, contudo, desta vez, usando o que nós já sabemos sobre o vetores e também, imprimindo a mensagem aprovado, no caso de a nota for maior ou igual ao valor 6. A saída pode ser visto na figura A.6.

```
1  nulo programa()
2  {
3      /*
4        Declarando vetores para guardar
5        as notas
6      */
7      int pedro[3];
8      int maria[3];
9
10     /*
11       Declarando variáveis para
12       guardar as médias
13     */
14     int media_pedro, media_maria;
15
16     /* inicializando as notas */
17     pedro[0] = 10;
18     pedro[1] = 3;
19     pedro[2] = 6;
20
21     maria[0] = 10;
22     maria[1] = 7;
23     maria[2] = 4;
24
25
26     /* computando as médias*/
27     media_pedro = (int)((pedro[0] + pedro[1] + pedro[2])/3);
28     media_maria = (int)((maria[0] + maria[1] + maria[2])/3);
29
30     /* Testar se Pedro está aprovado*/
31     se( media_pedro >= 6 ) entao
32       escreva "Pedro Aprovado";
33     novalinha;
34     /* Testar se Maria está aprovado*/
35     se( media_maria >= 6 ) entao
36       escreva "Maria Aprovado";
37 }
```

Codigo A.17: *Problema: pedro e maria - parte 3*

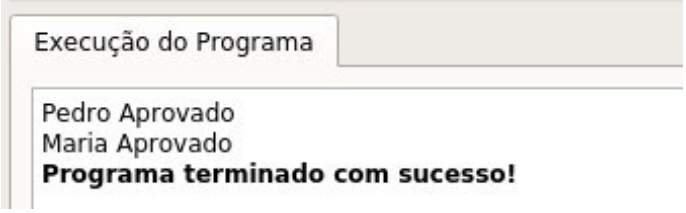


Figura A.6: *Saída do Problema: pedro e maria - parte 3*

Em Cafezinho, se for apenas uma ação por cada instrução condicional, então não há a necessidade de agrupa-las entre chaves{ }. Porém, se houver mais de uma ação, faz-se necessário. Agora alteremos novamente o código para imprimir a mensagem reprovado, caso a nota for menor que o valor 6, e alteremos a nota de Pedro para podermos testar esta nova funcionalidade, como pode ser visto no código A.18, e com a saída mostrada na figura A.7.

```
1  nulo programa()
2  {
3      /*
4        Declarando vetores para guardar
5        as notas
6      */
7      int pedro[3];
8      int maria[3];
9
10     /*
11       Declarando variáveis para
12       guardar as médias
13     */
14     int media_pedro, media_maria;
15
16     /* inicializando as notas */
17     pedro[0] = 5;
18     pedro[1] = 3;
19     pedro[2] = 6;
20
21     maria[0] = 10;
22     maria[1] = 7;
23     maria[2] = 4;
24
25
26     /* computando as médias*/
27     media_pedro = (int)((pedro[0] + pedro[1] + pedro[2])/3);
28     media_maria = (int)((maria[0] + maria[1] + maria[2])/3);
29
30     /* Testar se Pedro está aprovado*/
31     se( media_pedro >= 6 ) entao
```

```
32      {
33        escreva "Pedro Aprovado";
34        novalinha;
35      }
36      senao
37      {
38        escreva "Pedro Reprovado";
39        novalinha;
40      }
41      /* Testar se Maria está aprovado*/
42      se( media_maria >= 6 ) entao
43      {
44        escreva "Maria Aprovado";
45        novalinha;
46      }
47      senao
48      {
49        escreva "Maria Reprovado";
50        novalinha;
51      }
52  }
```

Codigo A.18: *Problema: pedro e maria - parte 4*

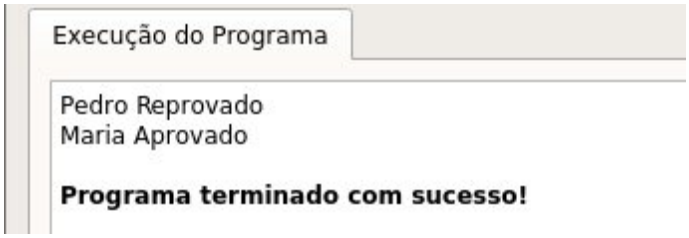


Figura A.7: *Saída do Problema: pedro e maria - parte 4*

A.5 Estrutura de Repetição

Algo que traz muita confusão aos iniciantes no mundo da programação é justamente as várias *Estrutura de Repetição* presentes com frequência nas linguagens de programação, assim como a situação mais apropriada de, uma destas ou qual destas, ser utilizada. Em Cafezinho, por sua vez, sendo uma linguagem que preza pela simplificação e facilitação do trabalho de programar, esta questão foi também sintetizada e cogitada da melhor forma ao aluno que está aprendendo a programar. Sendo assim, em Cafezinho há apenas uma estrutura de repetição: **enquanto** (*condição*) **execute** *ação*.

Portanto, para a compreensão desta instrução, iremos reescrever o problema já visto no código A.5 para que as funcionalidades presentes no programa façam o cálculo de

média, usando-se da estrutura de repetição da linguagem, como apresentando no código A.19.

```
1  nulo programa()
2  {
3      /*
4        Declarando vetores para guardar
5        as notas
6      */
7      int pedro[3];
8      int maria[3];
9      int i; /* contador */
10
11     /*
12       Declarando variáveis para
13       guardar as médias
14     */
15     int media_pedro, media_maria;
16
17     /* inicializando as notas */
18     pedro[0] = 5;
19     pedro[1] = 3;
20     pedro[2] = 6;
21
22     maria[0] = 10;
23     maria[1] = 7;
24     maria[2] = 4;
25
26
27     /* computando as médias*/
28     i=0;
29     enquanto ( i < 3) execute
30     {
31          media_pedro += pedro[i];
32          media_maria += maria[i];
33          ++i;
34     }
35     media_pedro = (int)(media_pedro/3);
36     media_maria = (int)(media_maria/3);
37
38     /* Testar se Pedro está aprovado*/
39     se( media_pedro >= 6 ) entao
40     {
41       escreva "Pedro Aprovado";
42       novalinha;
43     }
```

```
44      senao
45      {
46        escreva "Pedro Reprovado";
47        novalinha;
48      }
49      /* Testar se Maria está aprovado*/
50      se( media_maria >= 6 ) entao
51      {
52        escreva "Maria Aprovado";
53        novalinha;
54      }
55      senao
56      {
57        escreva "Maria Reprovado";
58        novalinha;
59      }
60  }
```

Codigo A.19: *Problema: pedro e maria - parte 5*

Observemos que, se fosse necessário o aumento do número de alunos, o cálculo da média permaneceria igual. Entretanto, o único valor que haveria de mudar seria o da condição de parada (i<3).

A.6 Entrada e Saída

A linguagem Cafezinho possui uma operação de entrada*(leia)* e uma operação de saída *(escreva)*. Estas operações podem receber mais de um parâmetro por vez, bastando apenas separa-los por uma vírgula ",". Além de que, a operação */escreva* pode trabalhar em parceria com a operação *novalinha*, bastando tão somente passa-la como argumento, o código A.20 apresenta essas funcionalidades, sua saída pode ser vista na figura A.8.

```
1  nulo programa()
2  {
3      int a;
4      car b;
5      real c;
6
7      leia a,b,c;
8
9      escreva novalinha, "Escreva: ", a, novalinha, b, novalinha, c;
10 }
```

Codigo A.20: *Leia e Escreva*

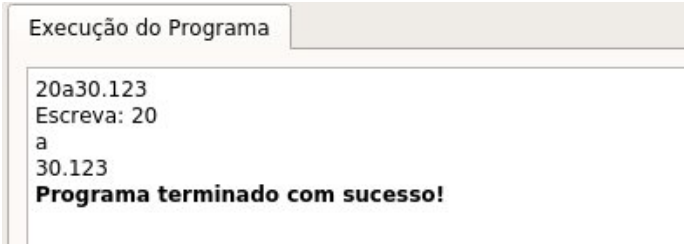


Figura A.8: *Saída do problema Leia e Escreva*

Agora podemos reescrever o problema apresentado no código A.5 para podermos assim ler as notas de "pedro" e "maria", computá-las e imprimir o resultado, como pode ser visto no código A.21, e com a saída do programa na figura A.9.

```
1  nulo programa()
2  {
3      /*
4        Declarando vetores para guardar
5        as notas
6      */
7      int pedro[3];
8      int maria[3];
9      int i; /* contador */
10
11     /*
12       Declarando variáveis para
13       guardar as médias
14     */
15     int media_pedro, media_maria;
16
17     /* inicializando as notas */
18     i=0;
19     enquanto (i < 3) execute
20     {
21          escreva (i+1), " nova de Pedro: ";
22          leia pedro[i];
23          novalinha;
24          ++i;
25     }
26
27     i=0;
28     enquanto (i < 3) execute
29     {
30          escreva (i+1), " nova de Maria: ";
31          leia maria[i];
32          novalinha;
33          ++i;
34     }
```

```
35
36
37      /* computando as médias*/
38      i=0;
39      enquanto ( i < 3) execute
40      {
41           media_pedro += pedro[i];
42           media_maria += maria[i];
43           ++i;
44      }
45      media_pedro = (int)(media_pedro/3);
46      media_maria = (int)(media_maria/3);
47
48      /* Testar se Pedro está aprovado*/
49      se( media_pedro >= 6 ) entao
50      {
51        escreva "Pedro Aprovado";
52        novalinha;
53      }
54      senao
55      {
56        escreva "Pedro Reprovado";
57        novalinha;
58      }
59      /* Testar se Maria está aprovado*/
60      se( media_maria >= 6 ) entao
61      {
62        escreva "Maria Aprovado";
63        novalinha;
64      }
65      senao
66      {
67        escreva "Maria Reprovado";
68        novalinha;
69      }
70  }
```

Codigo A.21: *Problema: pedro e maria - parte 6*

Agora facilmente conseguimos aumentar a capacidade do programa para ler 1000, ou indeterminadas, notas alterando tão somente o valor de 3, presente em (i < 3), por 1000 ou qualquer outro valor. Dessa forma, um programa o qual inicialmente era pequeno e pouco funcional, rapidamente se torna potente e capaz de exercer trabalhos muito mais úteis.

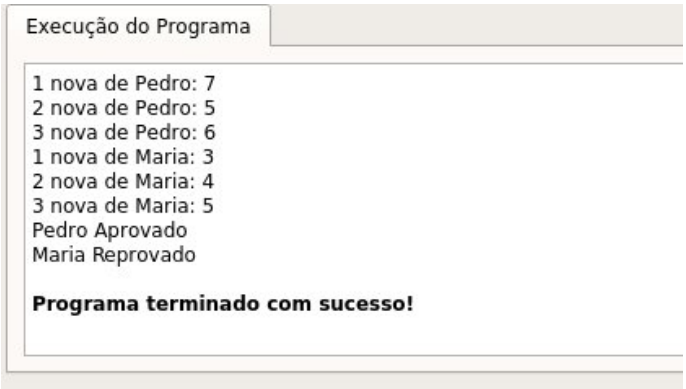


Figura A.9: *Saída do Problema: pedro e maria - parte 6*

A.7 Função e Procedimento

Muitas vezes, quando se programa, precisamos repetir ou copiar uma parte do código repetidas vezes, tornando o código extenso e difícil de fazer sua manutenção. Para solucionar este problema, a linguagem Cafezinho oferece a funcionalidade de **função** e **procedimento**.

Uma **função** é uma sub-rotina do programa que retorna algum valor. Já um **procedimento** não retorna valor algum. Uma **função** pode retornar os três tipos básicos da linguagem *(int, car, real)* e um **procedimento** retorna nada *(nulo)*.

Contudo, nas linguagens de programação não existe está separação entre função e procedimento, explicitamente. Também a linguagem Cafezinho trabalha dessa maneira, não distinguindo estas funcionalidades. Portanto, tendo isto já sido informado e explicado, deste momento em diante, será usado na sequência desse texto, apenas o termo função.

Definimos uma função na linguagem Cafezinho da seguinte forma: *tipo_retorno* **nome_da_função** (*tipo_parâmetros* **nome_parâmetros**) { *instruções* }. Agora podemos chama-la da seguinte forma: var = *nome_da_função*(**argumentos**) ou apenas *nome_da_função*(**argumentos**). É importante notar que para poder chamar uma função devemos tê-la declarado previamente.

Em programação chamamos de *cabeçalho* ou *protótipo* da função a combinação de: tipo de retorno, nome da função e parâmetros. Denominamos *parâmetros* todas as variáveis que estão declaradas no cabeçalho da função e chamamos de *argumentos* todos os valores que são passados na chamada de função.

Podemos observar que a função *programa()* é um tipo especial de função que pertence a linguagem Cafezinho, e é obrigatório sua declaração, pois a partir dela é que se é inicializada a execução do programa, além de que, é importante resaltar que esta não possui parâmetros.

Novamente vamos reescrever o problema A.5, separando em funções a parte de ler as notas, processar as médias e imprimir se está aprovado ou reprovado, mostrado no

código A.22, e com a saída apresentada na figura A.10.

```
1  /*
2   * Imprime um vetor de car até encontrar zero
3   */
4  nulo imprime(car vet[])
5  {
6       int i=0;
7       enquanto(vet[i]!=0) execute
8       {
9            escreva vet[i];
10           ++i;
11      }
12 }
13 /*
14  *  Testa e imprime se está aprovado ou reprovado
15  */
16 nulo esta_aprovado(int media, car nome[])
17 {
18      imprime(nome);
19      se( media >= 6 ) entao
20      {
21           escreva " Aprovado";
22      }
23      senao
24      {
25           escreva " Reprovado";
26      }
27      novalinha;
28 }
29 /*
30  *  Lê os dados das notas
31  */
32 nulo leia_dados(int qtd, int dados[], car nome[])
33 {
34      int i=0;
35      enquanto (i < qtd) execute
36      {
37           escreva (i+1), " nota de ";
38           imprime(nome);
39           escreva ": ";
40           leia dados[i];
41           novalinha;
42           ++i;
43      }
44 }
45 /*
```

```
46  *  Calcula a média
47  */
48 int calcula_media(int qtd, int dados[])
49 {
50      int i=0, media = 0;
51      enquanto (i < qtd) execute
52      {
53           media += dados[i];
54           ++i;
55      }
56      retorne (int)(media/qtd);
57 }
58
59 nulo programa()
60 {
61      /*
62           Declarando vetores para guardar
63           as notas
64      */
65      int pedro[3];
66      int maria[3];
67      car nome_pedro[6] = {'P','e','d','r','o', (car)(0)};
68      car nome_maria[6] = {'M','a','r','i','a', (car)(0)};
69      /* Usando 0 para indicar o final da palavra*/
70
71      /*
72           Declarando variáveis para
73           guardar as médias
74      */
75      int media_pedro, media_maria;
76
77      /* Lê os dados */
78      leia_dados(3, pedro, nome_pedro);
79      leia_dados(3, maria, nome_maria);
80
81      /* Calcula as médias */
82      media_pedro = calcula_media(3, pedro);
83      media_maria = calcula_media(3, maria);
84
85      /* Imprime se está aprovado ou reprovado */
86      esta_aprovado(media_pedro, nome_pedro);
87      esta_aprovado(media_maria, nome_maria);
88 }
```

Codigo A.22: *Problema: pedro e maria - parte 7*

Em Cafezinho quando declaramos um parâmetro do tipo *matriz*, a especificação

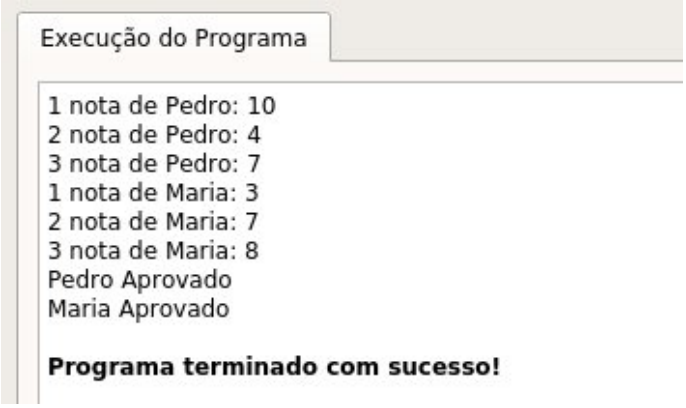


Figura A.10: *Saída do Problema: pedro e maria - parte 7*

do tamanho da primeira dimensão da matriz é opcional, como pode ser visto no código A.22. E quando uma *matriz* é passada a uma função como parâmetro, não se deve passar suas dimensões, mas apenas o seu nome, como se fosse uma variável escalar.

A.7.1 Passagem de Parâmetro por Valor e por Referência

Em computação, na *passagem de parâmetro por valor* o que é passado para função é apenas uma cópia do valor que a variável original contém e não ela propriamente. Assim, se for alterado o parâmetro dentro da função, o valor da variável original não é alterado. Já na *passagem de parâmetro por referência* é passado a própria variável, e qualquer alteração no parâmetro altera também a variável original.

Todo parâmetro na linguagem Cafezinho é passado por valor, exceto matrizes que são passadas por referência. Ou seja, se um parâmetro que é uma matriz for alterado dentro da função, a matriz original também será.

A.7.2 Retorno de valor

Uma função da linguagem Cafezinho pode retornar apenas um valor, e este valor somente pode ser dos tipos básicos da linguagem Cafezinho (*(int, car, real)*), além de não poder retornar uma *matriz*. Para se retornar algum valor, usa-se a instrução: /retorne valor. Ela irá retornar imediatamente a função chamadora com o valor que foi especificado, isto implica que qualquer instrução que exista depois da instrução *retorne* não será executado.

Toda função que retorna algum valor deve ter pelo menos uma instrução *retorne* dentro do seu código, caso contrário o compilador Cafezinho reclamará de um erro. Uma função que não retorna valor pode usar uma instrução, mas sem especificar valor e a partir deste ponto a função irá retornar a função chamadora imediatamente.

A.8 Outras Instruções

A linguagem Cafezinho ainda oferece a instrução *terminar* que irá encerrar imediatamente a execução do programa. Ela é diferente da instrução *retorne* que sai da função chamada e retorna para função chamadora, mas está, por sua vez, aborta a execução do programa.

A linguagem também possui a instrução *limpar* que limpa o terminal, ou seja, a área de entrada dos dados, para permitir uma maior interação programador-ambiente e dar lhe uma visão maior de como a programação funciona e como ocorre a interação entre seus elementos.

APÊNDICE B

Instruções da máquina virtual

Apresentaremos a seguir todas as instruções implementadas para a máquina virtual do projeto CafezinhoIDE. Para isso, definimos como valor imediato, uma constante numérica, ex.: 10, 20.123, 0x34 ou uma constante caractere, ex.: 'a', 'x', 'z', abreviada pelo nome*imm*. Definimos também como registrador todos aqueles descritos na Seção 4.2, e podendo ser abreviado pelo nome *reg*.

B.1 Instruções

mov reg1, *reg2*

reg1 = *reg2*, implementado pela classe IMove.

mov *reg1*, *imm*

reg1 = *imm*, implementado pela classe IMoveIm.

sp *reg1*, *imm* [*reg1*]

memoria[*reg1* + *imm*] = *reg1*, implementado pela classe ISalva.

cp *reg1*, *imm* [*reg1*]

reg1 = memoria[*reg1* + *imm*], implementado pela classe ICarrega.

adc *reg1*, *reg2*, *reg3*

reg1 = *reg2* + *reg3*, implementado pela classe IAdc.

adc *reg1*, *reg2*, *imm*

reg1 = *reg2* + *imm*, implementado pela classe IAdcIm.

sub *reg1*, *reg2*, *reg3*

reg1 = *reg2* - *reg3*, implementado pela classe ISub.

sub *reg1*, *reg2*, *imm*

reg1 = *reg2* - *imm*, implementado pela classe ISubIm.

mult *reg1*, *reg2*, *reg3*

reg1 = *reg2* * *reg3*, implementado pela classe IMult.

mult *reg1*, *reg2*, *imm*

reg1 = *reg2* * *imm*, implementado pela classe IMultIm.

div *reg1*, *reg2*, *reg3*

reg1 = *reg2* / *reg3*, implementado pela classe IDiv.

div *reg1*, *reg2*, *imm*

reg1 = *reg2* / *imm*, implementado pela classe IDivIm.

pot *reg1*, *reg2*, *reg3*

$reg1 = reg2^{reg3}$

implementado pela classe IPotencia.

pot *reg1*, *reg2*, *imm*

$reg1 = reg2^{imm}$

implementado pela classe IPotenciaIm.

rest *reg1*, *reg2*, *reg3*

reg1 = *reg2* % *reg3*, implementado pela classe IResto.

rest *reg1*, *reg2*, *imm*

reg1 = *reg2* % *imm*, implementado pela classe IRestoIm.

e *reg1*, *reg2*, *reg3*

reg1 = *reg2* & *reg3*, implementado pela classe IEBit.

e *reg1*, *reg2*, *imm*

reg1 = *reg2* & *imm*, implementado pela classe IEBitIm.

ou *reg1*, *reg2*, *reg3*

reg1 = *reg2* | *reg3*, implementado pela classe IOuBit.

ou *reg1*, *reg2*, *imm*

reg1 = *reg2* | *imm*, implementado pela classe IOuBitIm.

xor *reg1*, *reg2*, *reg3*

reg1 = *reg2* ^*reg3*, implementado pela classe IOuBit.

xor *reg1*, *reg2*, *imm*

reg1 = *reg2* ^*imm*, implementado pela classe IXorBit.

shd *reg1*, *reg2*, *reg3*

reg1 = *reg2* » *reg3*, implementado pela classe IXorBitIm.

she *reg1*, *reg2*, *reg3*

reg1 = *reg2* « *reg3*, implementado pela classe IShiftEsq.

lmq *reg1*, *reg2*, *reg3*

reg1 = *reg2* > *reg3*, implementado pela classe ILigMaiorQ.

lmq *reg1*, *reg2*, *imm*

reg1 = *reg2* > *imm*, implementado pela classe ILigMaiorQIm.

lmeq *reg1*, *reg2*, *reg3*

reg1 = *reg2* < *reg3*, implementado pela classe ILigMenorQ.

lmeq *reg1*, *reg2*, *imm*

reg1 = *reg2* < *imm*, implementado pela classe ILigMenorQIm.

lme *reg1*, *reg2*, *reg3*

reg1 = *reg2* >= *reg3*, implementado pela classe ILigMaiorEq.

lme *reg1*, *reg2*, *imm*

reg1 = *reg2* >= *imm*, implementado pela classe ILigMaiorEqIm.

lmee *reg1*, *reg2*, *reg3*

reg1 = *reg2* <= *reg3*, implementado pela classe ILigMenorEq.

lmee *reg1*, *reg2*, *imm*

reg1 = *reg2* <= *imm*, implementado pela classe ILigMenorEqIm.

leq *reg1*, *reg2*, *reg3*

reg1 = *reg2* == *reg3*, implementado pela classe ILigEq.

leq *reg1*, *reg2*, *imm*

reg1 = *reg2* == *imm*, implementado pela classe ILigEqIm.

lne *reg1*, *reg2*, *reg3*

reg1 = *reg2* != *reg3*, implementado pela classe ILigNaoEq.

lne *reg1*, *reg2*, *imm*

reg1 = *reg2* != *imm*, implementado pela classe ILigNaoEqIm.

lne *reg1*, *reg2*

reg1 = ! *reg2*, implementado pela classe INaoBit.

lne *reg1*, *imm*

reg1 = ! *imm*, implementado pela classe INaoBitIm.

Neg *reg1*, *reg2*

reg1 = - *reg2*, implementado pela classe INeg.

Neg *reg1*, *imm*

reg1 = - *imm*, implementado pela classe INegIm.

bool *reg1*

se *reg1* != 0 entao *reg1* = 1, implementado pela classe IBoolean.

cast *reg1*, tipo

reg1 = (tipo) *reg1*, implementado pela classe ICast

cmp *reg1*, *reg2*

compara *reg1* com *reg2*.

Se *reg1* > *reg2* então o estado das flags ficam:

bf = **true**;
sf = **false**;
ef = **false**.

Se *reg1* < *reg2* então o estado das flags ficam:

bf = **false**;
sf = **true**;
ef = **false**.

Se *reg1* >= *reg2* então o estado das flags ficam:

bf = **true**;
sf = **false**;
ef = **true**.

Se *reg1* <= *reg2* então o estado das flags ficam:

bf = **false**;
sf = **true**;
ef = **true**.

implementado pela classe ICmp.

cmp *reg1*, *imm*

compara *reg1* com *imm*.

Se *reg1* > *imm* então o estado das flags ficam:

bf = **true**;
sf = **false**;
ef = **false**.

Se *reg1* < *imm* então o estado das flags ficam:

bf = **false**;
sf = **true**;
ef = **false**.

Se *reg1* >= *imm* então o estado das flags ficam:

bf = **true**;
sf = **false**;
ef = **true**.

Se *reg1* <= *imm* então o estado das flags ficam:

bf = **false**;
sf = **true**;
ef = **true**.

implementado pela classe ICmpIm.

ret

er = pc +1, implementado pela classe IRet;

ecos texto

imprime o texto informado, implementado pela classe IEscritaPalavraIm;

ecoc *reg1*

imprime o valor que está em *reg1* como caractere, implementado pela IEscrita-Char;

ecoc *imm*

imprime o valor de *imm* como caractere, implementado pela IEscritaChar;

eco *reg1*

imprime o valor que está em *reg1* como número inteiro, implementado pela IEscritaInt;

eco *imm*

imprime o valor de *imm* como número inteiro, implementado pela IEscritaIntIm;

ecor *reg1*

imprime o valor que está em *reg1* como número de ponto flutuante, implementado pela IEscritaDouble;

ecor *imm*

imprime o valor de *imm* como número de ponto flutuante, implementado pela IEscritaDoubleIm;

nada

apenas incrementa o contador de programa, implementado pela classe INada.

parar

para a máquina virtual, implementado pela classe IParar.

sis *num*

executa uma operação de sistema indicado pelo número, são elas: 1- limpar tela e 2 - inserir uma quebra de linha, implementado pela classe ISistema.

debug_passo linha

marcador que indica a linha atual que será executada, implementando pela classe IDebugPasso

debug_empilha_exec

marcador que indica o início de uma chamada de função, implementando pela classe IDebugEmpilhaExec

debug_desempilha_exec

marcador que indica o fim de uma chamada de função, implementando pela classe IDebugDesempilhaExec

debug_var_empilha *nome_variavel*, offset, nó

marcador que indica o início do escopo de uma variável, implementando pela classe IDebugVariavelEmpilha

debug_var_desempilha *nome_variavel*

marcador que indica o fim do escopo de uma variável, implementando pela classe IDebugVariavelDesempilha