**UNIVERSIDADE FEDERAL DA GRANDE DOURADOS**
**BACHARELADO EM SISTEMAS DE INFORMAÇÃO**

**JACONS DE SOUZA MORAIS**

**RAFAEL FERREIRA AMARAL**

# IMPLEMENTAÇÃO DE UM AMBIENTE DE DESENVOLVIMENTO INTEGRADO PARA COMPILAR, EXECUTAR E DEPURAR PSEUDOCÓDIGO ESTRUTURADO EM BROWSERS

**DOURADOS**
**2016**

JACONS DE SOUZA MORAIS
RAFAEL FERREIRA AMARAL

# IMPLEMENTAÇÃO DE UM AMBIENTE DE DESENVOLVIMENTO INTEGRADO PARA COMPILAR, EXECUTAR E DEPURAR PSEUDOCÓDIGO ESTRUTURADO EM BROWSERS

Trabalho de Conclusão de Curso de graduação apresentado à banca examinadora como exigência parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Dr. Wellington Lima dos Santos

DOURADOS
2016

**JACONS DE SOUZA MORAIS**

**RAFAEL FERREIRA AMARAL**

# IMPLEMENTAÇÃO DE UM AMBIENTE DE DESENVOLVIMENTO INTEGRADO PARA COMPILAR, EXECUTAR E DEPURAR PSEUDOCÓDIGO ESTRUTURADO EM BROWSERS

Trabalho de Conclusão de Curso aprovado como requisito para a obtenção do título de Bacharel em Sistemas de informação, pela banca examinadora:

---

Orientador: Prof. Dr. Wellington Lima dos Santos
FACET - UFGD

---

Prof. Dr. Joinvile Batista Junior
FACET -UFGD

---

Prof. Me. Rodrigo Porfírio da Silva Sacchi
FACET - UFGD

Dourados, 23 de setembro de 2016

“O que sabemos é uma gota, o que ignoramos é um oceano”.

Isaac Newton

# Resumo

Este trabalho teve como objetivo a implementação, por meio da linguagem *javascript* e das ferramentas *Codemirror* e *Boostrap*, de um ambiente de desenvolvimento integrado, composto por editor de código, compilador, interpretador e depurador, para ser utilizado no ensino de algoritmos e lógica de programação para alunos de graduação, por meio de pseudocódigo em português estruturado. O compilador suporta tipos primitivos de dados como inteiro, real, lógico, caractere, string e ponteiro, além dos tipos compostos registro e arranjo. Possui as principais estruturas de controle de fluxo ensinadas atualmente nas disciplinas de programação como “caso”, “se”, “para”, “repita” e “enquanto”. O depurador do ambiente tem recursos como visualização da pilha de chamadas de procedimentos e funções, visualização e alteração dos valores das variáveis, execução passo a passo e ponto de parada.

Palavras-chave: compilador e interpretador de pseudocódigo, algoritmos e lógica, ferramenta de programação, português estruturado.

# Sumário

**Lista de Figuras**

**Lista de Tabelas**

**Lista de Palavras Estrangeiras**

**1 Introdução** **10**

1.1 Motivação e objetivos . . . . . 11

1.2 Trabalhos relacionados . . . . . 12

**2 Fundamentação Teórica** **18**

2.1 Linguagens Livre de Contexto . . . . . 18

2.1.1 Descrevendo uma Linguagem Livre de Contexto . . . . . 19

2.2 Compiladores . . . . . 20

2.3 Etapas de Compilação . . . . . 20

2.3.1 Análise Léxica . . . . . 21

2.3.2 Análise Sintática . . . . . 23

2.3.3 Análise Semântica . . . . . 24

2.3.4 Geração de Código . . . . . 25

2.4 Tabela de Símbolos . . . . . 27

2.5 Compilador Pascal S . . . . . 28

**3 Desenvolvimento do Projeto** **31**

3.1 Especificação do Dialeto Proposto . . . . . 31

3.1.1 Estrutura Léxica da Linguagem . . . . . 31
3.1.2 Declarações . . . . . 33
3.1.3 Tipos de Dados . . . . . 35
3.1.4 Estrutura Sequencial . . . . . 39
3.1.5 Estrutura Condicional . . . . . 40
3.1.6 Estruturas de Repetição . . . . . 41
3.1.7 Instruções de Entrada e Saída . . . . . 42
3.1.8 Subprogramas predefinidos . . . . . 43
3.1.9 Avaliação de Expressões . . . . . 45
3.2 Compilador PortugolJS . . . . . 46
3.2.1 Compilador . . . . . 47
3.2.2 Interpretador . . . . . 50
3.2.3 Depurador . . . . . 56
3.3 Ferramentas Utilizadas . . . . . 58
3.3.1 CodeMirror . . . . . 59
3.3.2 Bootstrap . . . . . 59
3.3.3 JavaScript . . . . . 59

**4 Conclusão** **61**

**5 Referências Bibliográficas** **63**

# Lista de Figuras


1 Média de Aproveitamento por Disciplinas . . . . . 12

2 Etapas de um processo de compilação . . . . . 21

3 Árvore de análise sintática da instrução A := (B * C) + 100; . . . . . 24

4 Diagrama de Execução do Interpretador . . . . . 51

5 Diagrama de uso da memória . . . . . 52

6 Tela de Execução de um Programa . . . . . 55

7 Modo de depuração do compilador. . . . . 58

# Lista de Tabelas


1 Comparação dos Trabalhos Relacionados . . . . . 13

2 Quadro comparativo dos tipos de dados suportados . . . . . 14

3 Funcionalidades relacionadas à subprogramas . . . . . 14

4 Comparação de estruturas de controle suportadas. . . . . 14

5 Símbolos gerados na análise léxica. . . . . 22

6 Especificação de um identificador. . . . . 32

7 Palavras reservadas do sistema. . . . . 32

8 Operadores da Linguagem. . . . . 33

9 Tipos de Dados. . . . . 36

10 Cabeçalhos de funções matemáticas . . . . . 43

11 Cabeçalhos de sub-programas para operações com string . . . . . 44

12 Cabeçalhos de sub-programas para gerenciamento de memória. . . . . 44

13 Funções para geração de números aleatórios e tempo. . . . . 44

14 Ordem de Precedência de Operadores. . . . . 46

15 Tabela Verdade dos Operadores Lógicos . . . . . 46

16 Instruções de Operações Matemáticas do Interpretador . . . . . 53

17 Instruções de Operações com Strings. . . . . 54

18 Instruções de Controle do Interpretador. . . . . 54

18 Instruções de Controle do Interpretador. . . . . 55

19 Erros de Execução . . . . . 56

20 Teclas de Atalho de Depuração. . . . . 58

# Lista de Palavras Estrangeiras

| | |
|---|---|
| **browser** | Software responsável por exibir e executar conteúdo da internet. |
| **web** | Rede de computadores mundial. |
| **bit** | Dígito binário, menor unidade de memória computacional. |
| **byte** | Sequência de 8 bits. |
| **token** | Símbolo reconhecido na análise léxica. |

CAPÍTULO 1

# INTRODUÇÃO

Um problema comum à maioria das instituições de ensino superior, especialmente no Brasil, é o alto índice de reprovação nas disciplinas de algoritmos e programação, em muitos casos, decorrente de uma formação básica deficiente do público alvo, agravada pela abstração intrínseca aos algoritmos que solucionam os exercícios e os trabalhos apresentados nestes componentes curriculares.

Existem algumas formas de representação ou escrita de algoritmos, porém a mais utilizada é o pseudocódigo, que no Brasil é denominado português estruturado, ou também Portugol. Trata-se de uma pseudo-linguagem, com elementos de português e das linguagens de programação Pascal e ALGOL, que contém as principais estruturas de uma linguagem de programação de alto nível, com uma sintaxe mais próxima à linguagem humana, portanto mais familiar e mais facilmente assimilada.

A flexibilidade proporcionada pela pseudolinguagem gera como principal inconveniente a falta de padronização ou regras, por isso as experiências no sentido de se construírem ferramentas para a execução de algoritmos em pseudocódigo são projetos isolados, geralmente de natureza acadêmica, limitados a interpretadores que executam linha por linha, com nenhuma ou limitada capacidade de depuração, enquanto não encontrarem um erro. Desta forma, após escrever seu pseudocódigo, resta ao aluno realizar um cansativo teste de mesa para verificar se sua lógica está correta, executá-lo em um desses interpretadores experimentais, ou traduzi-lo para uma linguagem de programação de alto nível como Pascal, C ou Java.

Muitos professores preferem ensinar algoritmos e lógica de programação diretamente em alguma linguagem de programação, tal como as citadas. Se esta disciplina for pré-requisito para outras, com foco em programação tal como ocorre nos cursos da área de computação, essa abordagem parece ser realmente mais producente. Adotada uma linguagem de programação, surge outra batalha, qual seja, a escolha de um IDE, que idealmente deve ser gratuito, multiplataforma, com interface organizada, leve em termos recursos do sistema, portável, ou de fácil instalação.

Após a adoção de um IDE, é comum o professor constatar que aluno gasta mais tempo fazendo escolhas como nome de projeto, pasta de destino, opções do compilador, entre outras, do que editando, compilando e executando um programa com pouco mais de 10 linhas de código, para resolver um exercício simples. Em IDEs populares como o Code Blocks[1], por exemplo, não é possível depurar o código se o arquivo fonte não estiver associado a um projeto.

## 1.1 Motivação e objetivos

Em um estudo de caso, realizado por TRINDADE (2014) na Universidade Federal da Grande Dourados, foram avaliadas as médias das médias de aproveitamento dos alunos nas disciplinas da área de computação entre 2006 e 2012, nos Cursos de Análise de Sistemas e Sistemas de Informação. Na Figura 1 pode-se observar que as disciplinas de “Algoritmos” e “Algoritmos e Programação”, ambas do primeiro ano destes cursos e representadas pelas duas barras inferiores, possuem as médias mais baixas entre as disciplinas analisadas.

Nos casos em que a disciplina de algoritmos e programação não representa o primeiro elo de uma cadeia de disciplinas, acredita-se que a utilização do Portugol pode propiciar uma curva de aprendizagem mais inclinada, especialmente se forem utilizadas ferramentas de apoio como Portugol Studio[2], Portugol IDE[3], VisualAlg[4], Portugol Online[5] e G-Portugol[6], as quais precisam ser instaladas.

Visando contribuir para a melhoria deste cenário, surgiu da ideia do desenvolvimento deste trabalho, cujo objetivo geral é a implementação de um software, a ser executado em navegadores, diferente das outras ferramentas, com as funcionalidades de edição, compilação, execução e depuração de algoritmos em português estruturado e com a finalidade de servir como uma ferramenta de apoio para o ensino de algoritmos e lógica de programação para alunos de graduação de instituições de ensino superior.

[1] Disponível em: <http://www.codeblocks.org/>. Acesso em 05/04/2016
[2] Disponível em: <https://sourceforge.net/projects/portugolstudio/>. 12/04/2016
[3] Disponível em: <http://www.dei.estt.ipt.pt/portugol/>. Acesso em 12/04/2016
[4] Disponível em: <http://www.apoioinformatica.inf.br/produtos/visualg>. Acesso em 14/04/2016
[5] Disponível em: <https://vinyanalista.github.io/portugol/>. Acesso em 20/04/2016
[6] Disponível em: <https://sourceforge.net/projects/gpt.berlios/>. Acesso em 14/04/2016

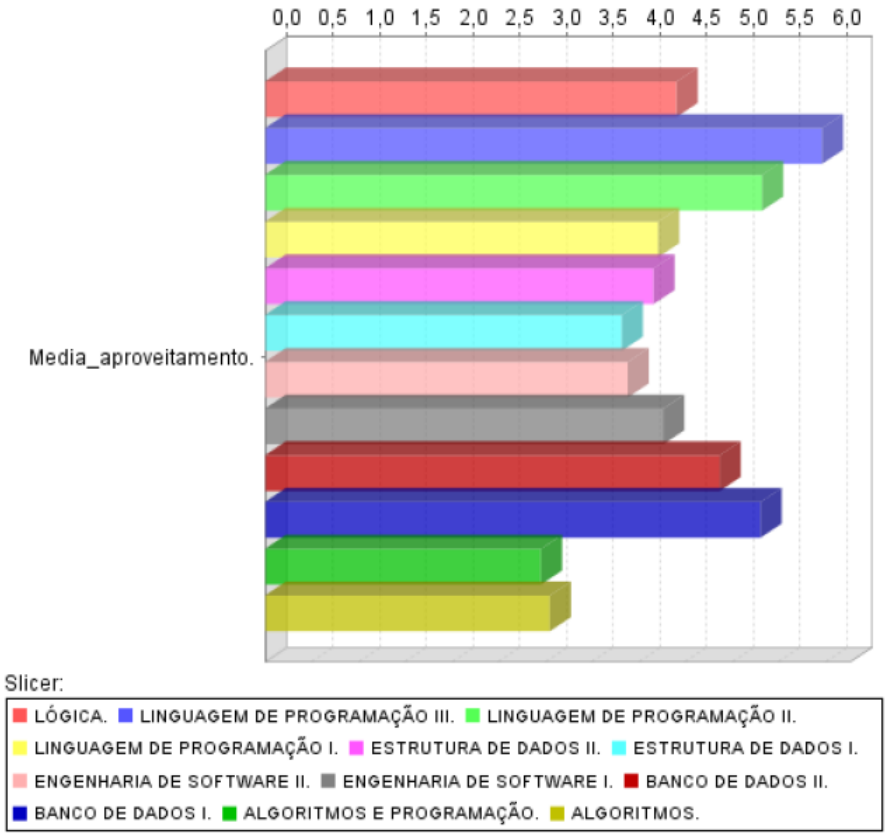


**Figura 1:** Média de Aproveitamento por Disciplinas

## 1.2 Trabalhos relacionados

Existem diversas aplicações para o Portugol, cada uma com suas peculiaridades, mas a maioria requer o *download* e a instalação das mesmas, o que se constitui em mais um obstáculo para a execução de códigos, muitas vezes pequenos e simples.

O Portugol Studio constitui-se de um ambiente de desenvolvimento construído para permitir a criação e a execução dos programas escritos em Portugol. Possui todos os recursos básicos de uma IDE: manipulação de arquivos de código-fonte, execução e interrupção de programas, área para entrada e saída de dados, área para exibição dos erros de compilação e de execução, destaque da sintaxe e árvore estrutural, que permite identificar e compreender os elementos que compõem o programa escrito, além de um depurador passo a passo do programa.

O Portugol IDE é um ambiente de desenvolvimento de algoritmos em Portugol, mais especificamente, trata-se de um simulador de linguagem algorítmica que visa o desenvolvimento do raciocínio lógico. Ele suporta comandos condicionais, de entrada/saída e de repetição, bem como os tipos básicos de dados, arrays e

constantes.

Portugol Viana é uma ferramenta que permite editar algoritmos em português (do Brasil e de Portugal), executá-los e também monitorá-los. O ponto forte do Portugol Viana é possibilidade de definir e invocar funções com parâmetros por valor e referência, bem como criar novos tipos de dados e até mesmos listas encadeadas.

O Visualg é um editor e interpretador de algoritmos que tem suporte para funções, procedimentos, comandos de repetição, condição, recursão e quatro tipos de dados: inteiro, real, caractere e lógico.

O G-Portugol é um dialeto da pseudolinguagem Portugol, muito usada para descrever algoritmos de forma livre e espontânea, em português. Atualmente possui um compilador, interpretador e um editor simples .

A comparação entre os trabalhos relacionados é apresentada na Tabela 1, na qual as colunas “interpretador” e “depurador” indicam se o compilador possui interpretador próprio e alguma funcionalidade de depuração, as colunas “tamanho” e “S.O.” indicam o tamanho do programa instalado e plataformas suportadas, sendo que as letras W, L e M se referem à Windows, Linux e Mac, respectivamente.

**Tabela 1:** Comparação dos Trabalhos Relacionados

| nome | interpretador | depurador | S.O. | tamanho |
|---|---|---|---|---|
| Portugol Studio | | X | W, L ou M | 50,8MB |
| Portugol IDE | X | X | W | 1,94MB |
| VisuAlg | X | X | W | 803 KB |
| G-Portugol | | | W e L | 1,2MB |
| Portugol Online | X | X | W, L ou M | Online |
| PortugolJS | X | X | W, L ou M | Online |

Os tipos de dados suportados pelas diferentes ferramentas são mostrados no quadro comparativo representado na Tabela 2, na qual cada coluna representa um tipo de dado, sendo “I” para inteiro, “R” para real, “L” para lógico, “C” para caractere, “S” para string, “A” para estruturas compostas homogêneas, “E” para estruturas compostas heterogêneas e “P” para ponteiro.

As funcionalidades relacionadas à subprogramas são apresentados na Tabela 3, sendo elas retorno (R), passagem por referência (PR), recursividade (REC) e níveis de escopo (NE).

**Tabela 2:** Quadro comparativo dos tipos de dados suportados

| NOME | I | R | L | C | S | A | E | P |
|---|---|---|---|---|---|---|---|---|
| Portugol Studio | X | X | X | X | X | X | | |
| Portugol IDE | X | X | X | X | X | X | | |
| VisuAlg | X | X | X | X | | X | | |
| G-Portugol | X | X | X | X | X | X | | |
| Portugol Online | X | X | X | | X | X | X | |
| PortugolJS | X | X | X | X | X | X | X | X |

**Tabela 3:** Funcionalidades relacionadas à subprogramas

| NOME | R | PR | REC | NE |
|---|---|---|---|---|
| Portugol Studio | X | X | X | 2 |
| Portugol IDE | X | | | 2 |
| VisuAlg | X | | | 2 |
| G-Portugol | X | | | 2 |
| Portugol Online | X | X | | 2 |
| PortugolJS | X | X | X | 10 |

Na Tabela 4 é ilustrado o suporte as principais estruturas de controle de fluxo do programa presentes nos compiladores clássicos.

**Tabela 4:** Comparação de estruturas de controle suportadas.

| NOME | SE/SENÃO | CASO | PARA | REPITA | ENQUANTO |
|---|---|---|---|---|---|
| Portugol Studio | X | X | X | X | X |
| Portugol IDE | X | X | X | X | X |
| VisuAlg | X | | X | X | X |
| G-Portugol | X | X | X | X | X |
| Portugol Online | X | | X | X | X |
| PortugolJS | X | X | X | X | X |

Cada um dos trabalhos relacionados segue sua própria sintaxe do português estruturado. Para efeito de comparação de sintaxe de cada ferramenta utilizamos uma

implementação do algoritmo de ordenação *BubbleSort*, que é ineficiente mas utiliza várias estruturas dentro de um código simples e compacto.

Implementação no PortugolJS:

```
programa bubblesort
var
  vet: arranjo[1..10] de inteiro
  i, j, aux: inteiro
inicio
  escrevaln('VETOR INICIAL')
  para i de 1 ate 10 faca
  inicio
    vet[i] := aleatorio(100)
    escrevaln(vet[i])
  fim
  para i de 1 ate 10 faca
  inicio
    para j de 1 ate 9 faca
    inicio
      se (vet[j] > vet[j+1]) entao
      inicio
        aux := vet[j]
        vet[j]:= vet[j+1]
        vet[j+1]:= aux
      fim
    fim
  fim
  escrevaln('VETOR ORDENADO: ')
  para i de 1 ate 10 faca
     escrevaln(vet[i])
fim
```

Implementação Portugol Studio:

```
programa{
  inclua biblioteca Util --> u
  funcao inicio(){
    inteiro vet[10]
    inteiro i, j, aux
    para (i = 0; i < 10; i++)
      vet[i] = u.sorteia(0, 100)
    para (i = 0; i < 10; i++)
      para (j = 0; j < 9; j++)
        se(vet[j] > vet[j+1])
            aux = vet[j]
            vet[j] = vet[j+1]
            vet[j+1] = aux
    escreva("VETOR ORDENADO:\n")
```

```
15      para (i = 0; i < 10; i++)
16          escreva(vet[i]+"\n")
17    }
18  }
```

Implementação no VisualAlg:

```
1  algoritmo "bubblesort"
2  var
3    vet:vetor[1..10] de inteiro
4    i, j , aux:inteiro
5  inicio
6    aleatorio on
7    para i:=1 ate 10 faca
8        leia(vet[i])
9    fimpara
10   aleatorio off
11   para j:=10 ate 1 passo -1 faca
12       para i:=1 ate j-1  faca
13           se (vet[i] > vet[i+1]) entao
14             aux:= vet[i]
15             vet[i]:=vet[i+1]
16             vet[i+1]:=aux
17           fimse
18       fimpara
19   fimpara
20   escreva("vetor ordenado: ")
21   para i:=1 ate 10 faca
22       escreva(vet[i])
23   fimpara
24 fimalgoritmo
```

Implementação no Portugol Online:

```
1  algoritmo
2  declare
3    vet[10] numerico
4    i, j, k, aux numerico
5      para i <- 1 ate 10 faca
6      inicio
7          leia vet[i]
8      fim
9      k <- 0
10     para i <- 1 ate 10 faca
11     inicio
12         para j <- 1 ate 9 faca
13         inicio
14             k <- j + 1
```

```
15              se vet[j] > vet[k] entao
16                  aux <- vet[j]
17                  vet[j] <- vet[k]
18                  vet[k] <- aux
19          fim
20      fim
21      para i <- 1 ate 10 faca
22      inicio
23          escreva vet[i]
24      fim
25  fim_algoritmo.
```

Implementação no Portugol IDE:

```
1   inicio
2       variavel real j , i , aux , vet [ 100 ]
3       para i de 1 ate 100
4            vet[i] <-  aleatorio()
5       proximo
6       para i de 1 ate 100
7           para j de 1 ate 99
8               se vet [ j ] > vet [ j + 1 ] entao
9                   aux <- vet [ j ]
10                  vet [ j ] <- vet [ j + 1 ]
11                  vet [ j + 1 ] <- aux
12              fimse
13          proximo
14      proximo
15      escrever "VETOR ORDENADO: \n"
16      para i de 1 ate 100
17          escrever vet [ j ]
18      proximo
19  fim
```

CAPÍTULO 2

# FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados os princípios básicos para a implementação de um compilador, especificação de gramáticas livre de contexto, etapas de compilação, geração de código e interpretação de instruções utilizando conceitos de pilha de execução. Na sequência, será apresentado o compilador utilizado como base para o desenvolvimento deste trabalho, a sintaxe suportada, funções pré definidas no compilador e tipos de dados suportados.

## 2.1 Linguagens Livre de Contexto

Uma linguagem é dita livre de contexto quando é especificada por uma gramática livre de contexto [MENEZES 1998].

Uma gramática é um conjunto de regras para a criação de uma linguagem através de mecanismos de geração utilizando regras de substituição. As regras de substituição são especificadas através de produções, cada produção representa uma regra da gramática, ela é escrita em pares ordenados do tipo x -> y sendo x um símbolo não-terminal e y uma cadeia de símbolos terminais ou não-terminais [NETO 1997].

O nome livre de contexto se baseia no fato de a gramática não depender do contexto onde está inserida para a derivação dos símbolos presentes na sentença, ou seja, são linguagens em que a derivação de um símbolo não-terminal não leva em conta os símbolos antecedentes ou que ainda serão analisados para gerar a derivação [MENEZES 1998].

Uma das vantagens deste tipo de gramática é a possibilidade de especificar linguagens com duplo balanceamento, recurso muito utilizado em linguagens de programação para delimitar blocos de instruções e parênteses. Considerando G uma gramática qualquer definida pelas regras abaixo e P as produções que podem ser geradas por esta gramática:

```
G = {{E}, {+, /, -, *, (, ) x, y}, P, E}, sendo:

P = {E -> E + E | E * E | E / E | E - E | ( E ) | x}.
```

A expressão aritmética “x * (y + x)” pode ser formada a partir desta gramática utilizando as derivações a seguir:

```
P -> E -> E * E -> x * E -> x * (E) -> x * (E + E) ->
x * (y + E) -> x * (y + x).
```

### 2.1.1 Descrevendo uma Linguagem Livre de Contexto

Atualmente a forma mais utilizada para descrever a sintaxe de uma linguagem de programação é através da Extended Backus-Naur Form, ou EBNF.

A EBNF é uma metalinguagem utilizada para descrever linguagens de programação desenvolvida na década de 1950 por John Backus e Peter Naur para a descrição do Algol 60. Posteriormente atualizada por Niklaus Wirth para a descrição da linguagem Pascal, a EBNF possui uma notação bastante natural para a especificação da sintaxe [SEBESTA 2005].

A EBNF utiliza abstrações para delimitar estruturas sintáticas da linguagem, sendo definida por pares ordenados nos quais o símbolo à esquerda é a abstração a ser definida e o símbolo à direita é um conjunto de símbolos terminais ou não terminais.

Considere-se a notação EBNF do comando de atribuição em Pascal:

```
<atribuição> ::= <var> := <expressão>; sendo:
<expressão> ::= (<expressão>) | <expressão> <op> <expressão> |
<var> | <literal>
<op> ::= + | - | * | /
<var> ::= x | y
<literal> ::= 1 | 5
```

Deste modo, a instrução “x := y * (y - 1) + 1;” poderia ser criada utilizando as seguintes derivações:

```
<atribuição> ::=  <var> := <expressão>;
::= x := <expressão> <op> <expressão>;
::= x := <expressão> <op> <expressão> <op> <expressão>;
  ::= x := y * ( <expressão> ) + <literal>;
  ::= x := y * (<expressão> <op> <expressão>) + 1;
  ::= x := y * (y - 1) + 1;
```

A EBNF utiliza **{** e **}** para especificar símbolos opcionais que podem ser repetidos por um indeterminado número de vezes, o símbolo **+** após **}** indica que o conjunto de símbolos entre chaves deve aparecer ao menos uma vez. Os caracteres **[** e **]** representam símbolos opcionais que não podem ser repetidos.

## 2.2 Compiladores

O fato de os computadores compreenderem apenas a linguagem de máquina, composta de 0s e 1s, tem motivado, desde os primórdios da computação, o desenvolvimento de linguagens que permitem a comunicação entre o programador e a máquina de uma forma mais intuitiva e natural.

A ferramenta responsável por facilitar esta comunicação e realizar a tradução de linguagens de programação para linguagens de máquina é o compilador. Um compilador é um programa que realiza a leitura de um código escrito em uma determinada linguagem, denominada linguagem fonte, e gera um código equivalente, chamada de linguagem alvo, em alguns casos, a linguagem alvo é a linguagem reconhecida pelo computador, chamada de linguagem de máquina. Há também a compilação de linguagens fonte para linguagens alvo que não são reconhecidas diretamente pelo computador, sendo que nestes casos pode ser utilizado um outro compilador para concluir a tradução para a linguagem de máquina ou utilizado um interpretador, que terá como papel realizar a tradução, em tempo de execução, da linguagem gerada pelo compilador para linguagem de máquina, para que as instruções possam ser executadas [AHO 2008].

Durante as etapas de compilação podem existir erros que são de natureza léxica, sintática ou semântica. Um papel importante do compilador é informar ao usuário os erros encontrados durante a leitura da linguagem fonte, e quando possível, sugerir dicas para solucionar o problema.

## 2.3 Etapas de Compilação

No decorrer do século XX o homem aperfeiçoou as técnicas utilizadas na implementação de compiladores e a divisão do processo de compilação em etapas ajudou a diminuir o elevado grau de complexidade do projeto.

Atualmente existem milhares de compiladores desenvolvidos, que reconhecem as mais variadas linguagens fonte, mas independente da finalidade para a qual o compilador foi desenvolvido, as tarefas básicas de um compilador podem ser representadas por meio do diagrama da Figura 2, no qual são ilustradas as principais etapas de compilação.

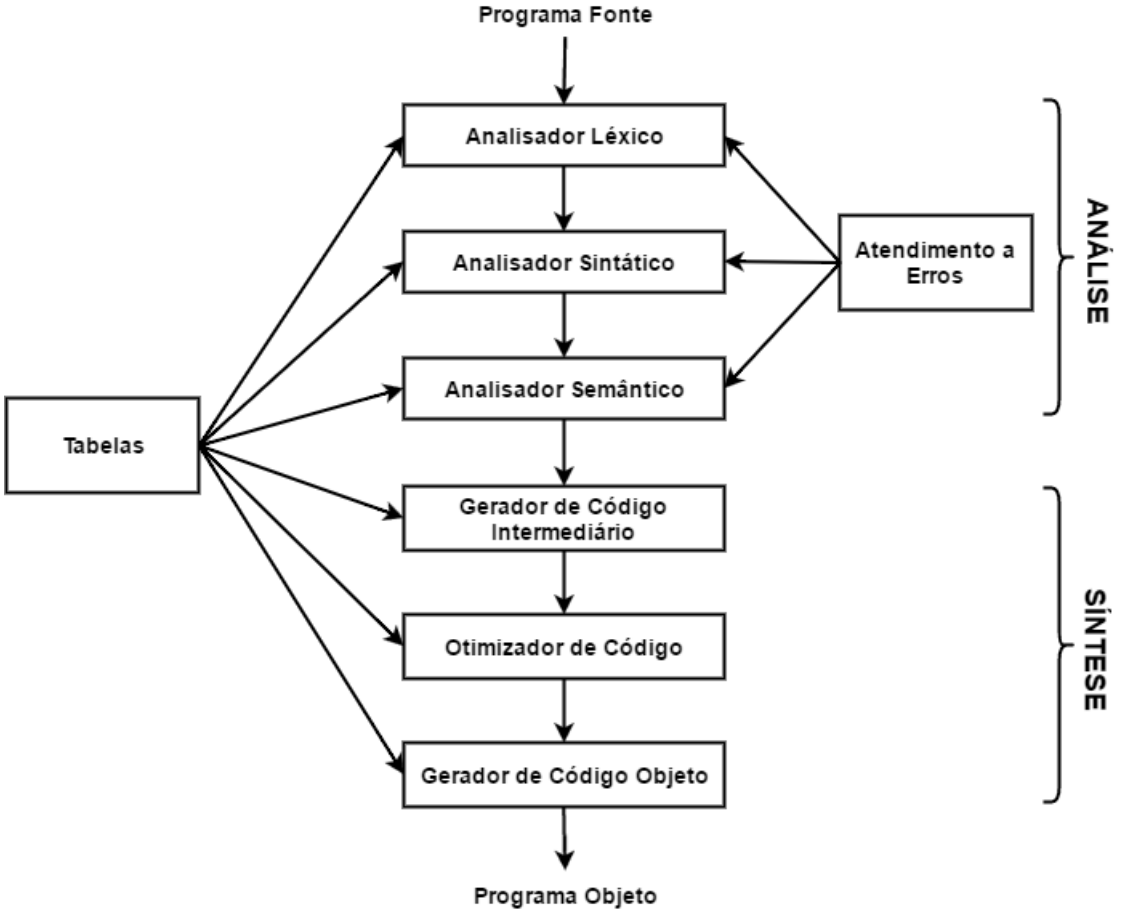


**Figura 2:** Etapas de um processo de compilação

### 2.3.1 Análise Léxica

A análise léxica é a primeira etapa do processo de compilação, sendo responsável por realizar a leitura da linguagem fonte e dividir o código em símbolos, denominados *tokens*. Este processo é realizado através da leitura caractere por caractere do código fonte, verificando se formam símbolos reconhecidos pela linguagem e informando ao analisador sintático o *token* encontrado e o lexema formado pelos caracteres lidos. Segundo o *Dicionário Michaelis da Língua Portuguesa* "Lexemas são palavras ou parte delas que servem de base ao sentido por ela expresso". Quando o processo de análise

léxica não consegue reconhecer um ou mais caracteres ou o símbolo lido não faz parte da linguagem, é necessário informar ao usuário o erro encontrado [SEBESTA 2003].

Os *tokens* são os símbolos terminais de uma linguagem de programação e, segundo Sebesta et al.(2003) os símbolos terminais são construções sintáticas de pequena escala da linguagem como operadores, valores literais, delimitadores e identificadores.

Considerando-se a análise léxica da instrução de atribuição em Linguagem Pascal "**A := (B * C) + 100;**", são mostrados na Tabela 5 o resultado da análise léxica e os símbolos e lexemas que são retornados ao analisador sintático.

**Tabela 5:** Símbolos gerados na análise léxica.

| Símbolo | Lexema |
|---|---|
| identificador | A |
| operador de atribuição | := |
| parêntese esquerdo | ( |
| identificador | B |
| operador de multiplicação | * |
| identificador | C |
| parêntese direito | ) |
| operador de adição | + |
| valor literal inteiro | 100 |
| ponto e vírgula | ; |

Em determinados momentos da leitura de caracteres, são encontrados dígitos numéricos e é papel do analisador léxico reconhecer a sequência de caracteres numéricos e convertê-los para constantes numéricas, sejam do tipo inteiro ou real, informando ao analisador sintático, juntamente com o símbolo referente à constante numérica, o valor lido do programa fonte. Os lexemas que são retornados com o símbolo podem ter três tipos: cadeia de caracteres, inteiro ou real.

O analisador léxico também realiza os saltos nos trechos de comentários e espaços em branco, mas em alguns compiladores, como Python, os espaços em branco à esquerda são utilizados para delimitar blocos de instruções.

O analisador léxico inicia a construção da tabela de símbolos que, segundo Aho et al.(2008), é uma estrutura de dados, na qual cada identificador contém um registro com seus atributos, sendo utilizada durante todo o processo de compilação como central de informações sobre os identificadores, que são nomes atribuídos a variáveis, constantes, tipos de dados etc. A diferenciação entre palavras chave da linguagem e identificadores pode ser complexa, por isso a maioria das linguagens de programação

tornam as palavras chave palavras reservadas que não podem ser utilizadas como identificadores, simplificando o processo de análise, pois quando uma cadeia de caracteres não se encaixa como uma palavra chave, assume-se que ela seja um identificador.

Em alguns casos no processo de leitura de caracteres, o analisador léxico precisa ler à frente do caractere atual para decidir qual símbolo retornar ao analisador sintático, uma vez que a maioria dos compiladores atualmente implementam operadores compostos por mais de um caractere. Em Pascal, por exemplo, o lexema **:=** representa o operador de atribuição e o lexema **>=** forma o operador relacional "maior ou igual". Porém, existem exceções, quando o caractere lido à frente não corresponde ao lexema do operador composto que está sendo avaliado, nesses casos ele deve retornar à cadeia de caracteres do código fonte, pois pertencerá a outro símbolo ainda não lido [AHO 2008].

### 2.3.2 Análise Sintática

Na segunda etapa do processo de compilação é realizada a análise sintática do programa fonte, os símbolos e lexemas são recebidos do analisador léxico e começa a ser gerada a árvore de análise sintática do programa. Quando um símbolo lido não corresponde ao esperado, um erro deve ser informado ao usuário. Para a análise sintática, existem duas principais técnicas utilizadas, a ascendente (*Bottom-Up*) e a descendente (*Top-Down*). Elas se diferenciam de acordo com a forma em que a árvore de análise sintática é construída, sendo que a técnica ascendente inicia-se pelas folhas e vai em direção à raiz, enquanto a técnica descendente inicia a construção da árvore pela raiz e vai em direção às folhas [SEBESTA 2003].

#### Análise Sintática Descendente

A análise sintática descendente é mais utilizada na implementação de compiladores para linguagens de programação, pois facilita a construção da árvore sintática manualmente. Considere-se o exemplo da operação de atribuição na Linguagem Pascal "A := (B * C) + 100;", cuja árvore sintática, construída de forma descendente, é mostrada na Figura 2 [AHO 2008].

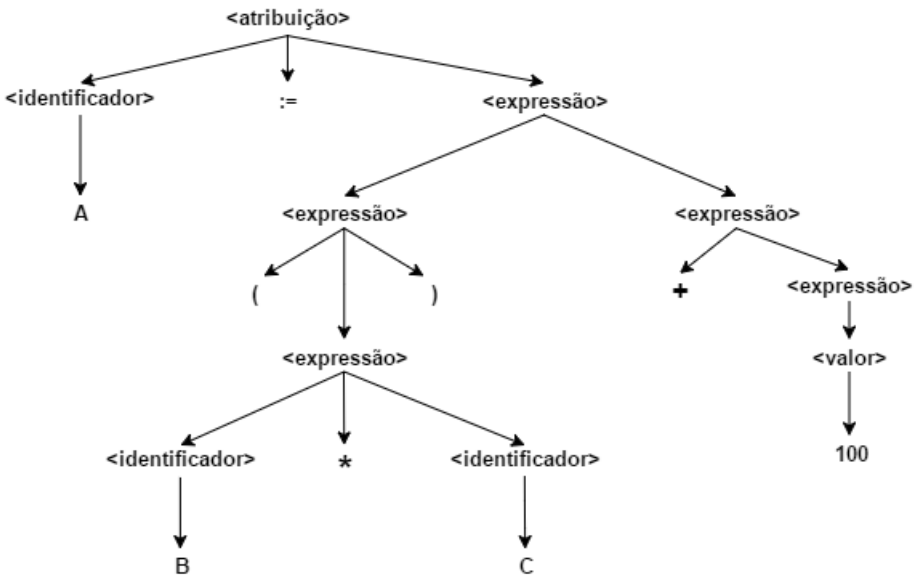


**Figura 3:** Árvore de análise sintática da instrução A := (B * C) + 100;

### 2.3.3 Análise Semântica

Na fase de análise semântica são verificados os erros semânticos no programa fonte e capturadas as informações de tipo para a fase subsequente de geração de código, utilizando a estrutura hierárquica determinada pela fase de análise sintática, para identificar os operadores e operandos das expressões e enunciados. Um importante componente da análise semântica é a verificação de tipos, nela o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte. Por exemplo, muitas definições nas linguagens de programação requerem que o compilador relate um erro a cada vez que um número real seja usado para indexar um arranjo. No entanto, a especificação da linguagem pode permitir algumas coerções de operandos, como, por exemplo, quando um operando aritmético binário é aplicado a um inteiro e a um real. Nesse caso, o compilador pode precisar converter o inteiro para real [AHO 2008].

A verificação estática, que acontece durante a compilação (diferente da verificação dinâmica que acontece durante a execução), assegura certas verificações como verificação de tipos, verificação do fluxo de controle, verificação de unicidade e verificações relacionadas aos nomes.

#### Verificação de tipos

Um compilador deve relatar um erro se um operador for aplicado a um operando incompatível, por exemplo: se uma variável do tipo arranjo for atribuída a uma variável do tipo inteiro.

#### Verificação do fluxo de controle

Os enunciados que fazem o fluxo de controle deixar uma construção precisam ter algum local para onde transferir o controle. Por exemplo, um enunciado *break* em C faz com que o controle deixe o *while*, *for* ou *switch* envolvente mais interno: um erro ocorre se tal enunciado envolvente não existir.

#### Verificações de unicidade

Existem situações nas quais um objeto precisa ser definido exatamente uma vez. Por exemplo, em Pascal, um identificador precisa ser declarado univocamente, os rótulos em enunciados *case* precisam ser distintos, e os elementos num tipo escalar não podem ser repetidos.

#### Verificações relacionadas aos nomes

Algumas vezes, o mesmo nome precisa figurar duas ou mais vezes. Por exemplo, em Ada, um laço ou bloco precisa ter um nome que apareça ao início e ao final da construção. O compilador precisa verificar se o mesmo nome é usado em ambos os locais [AHO 2008].

### 2.3.4 Geração de Código

O processo de geração de código é dividido em duas etapas: geração de código intermediário e geração de código objeto.

#### Geração de Código intermediário

Segundo Price e Toscani et al.(2001, p.115) a geração de código intermediário é a transformação da árvore de derivação em um segmento de código. Esse código

pode eventualmente ser o código objeto final. Mas, na maioria das vezes constitui-se em código intermediário, pois a tradução de código fonte para objeto em mais de um passo apresenta algumas vantagens:

- possibilita a otimização do código intermediário, de modo a obter-se o código objeto final mais eficiente;
- simplifica a implementação do compilador, resolvendo, gradativamente, as dificuldades da passagem de código fonte para objeto (alto-nível para baixo-nível), já que o código fonte pode ser visto como um texto condensado que "explode" em inúmeras instruções elementares de baixo nível;
- possibilita a tradução de código intermediário para diversas máquinas.

A desvantagem de gerar código intermediário é que o compilador requer um passo a mais. A tradução direta do código fonte para objeto leva a uma compilação mais rápida.

A grande diferença entre o código intermediário e o código objeto final é que o intermediário não especifica detalhes da máquina alvo, tais como quais registradores serão usados, quais endereços da memória serão referenciados, etc.

**Geração de Código Objeto**

Segundo Prince e Toscani(2001, p.187) os principais requisitos impostos a geradores de código objeto são os seguintes:

- o código gerado deve ser correto e de alta qualidade;
- o código deve fazer uso efetivo dos recursos da máquina;
- o código gerado deve executar eficientemente.

O problema de gerar código ótimo é indecidível como tantos outros. Na prática, devem ser usadas técnicas heurísticas que geram "bom" código, não necessariamente ótimo.

## 2.4 Tabela de Símbolos

A tabela de símbolos é uma estrutura de dados do compilador, que armazena as informações sobre os identificadores encontrados no código fonte. Deve permitir que novas entradas sejam adicionadas e que os dados presentes na tabela possam ser encontrados rapidamente, operações que podem ser feitas com eficiência O(1) quando se usa uma tabela *hash*.

Algumas linguagens de programação como Pascal exigem uma área específica para a declaração de novos identificadores, outras como C e Java permitem que uma variável seja criada a qualquer momento durante a leitura do código, mas sempre é necessário declarar uma variável antes de utilizá-la, no entanto, existem ainda linguagens como Javascript, que não exigem a declaração de variáveis antes de utilizá-las, ou seja, não existe campo específico para declaração e caso o nome lido na análise léxica não seja encontrado na tabela de símbolos e não seja uma operação de leitura, é criado um novo registro para armazenar o mesmo, essa forma de implementação é facilitada em linguagens com tipagem dinâmica, onde uma variável não possui tipo especificado em sua declaração [AHO 2008].

Durante a implementação dos compiladores, é ideal que a tabela de símbolos seja criada com tamanho dinâmico, para que possa armazenar quantos símbolos sejam necessários, quando isto não é possível, é importante especificar uma tabela de símbolos com um considerável tamanho para comportar códigos extensos [PRICE & TOSCANI 2000].

Uma tabela de símbolos pode ser implementada utilizando tabelas *hash*, árvores de busca binária, árvores AVL, árvores B ou listas lineares. As listas lineares permitem uma implementação simples e com um desempenho razoável para a compilação de algoritmos com tempo constante para inserção de novos elementos e tempo linear em relação ao tamanho da lista para a busca de elementos [LOUDEN 2004].

A tabela de símbolos armazena diversos atributos dos identificadores como: tipo, endereço na memória, limites(no caso de arranjos), nível de escopo, entre outros. Um problema típico na implementação de tabelas de símbolo em linguagens de baixo nível é a forma de armazenar o nome do identificador, em alguns casos é necessário criar uma estrutura auxiliar para armazenar os caracteres que formam o nome do identificador, nesse caso é possível economizar memória, pois o tamanho dos nomes é alocado dinamicamente e não é necessário armazenar espaço extra, mas a busca

dos identificadores pode se tornar mais demorada [PRICE & TOSCANI 2000].

## 2.5 Compilador Pascal S

No início dos anos 1970, o Departamento de Ciência da Computação do Instituto Federal de Tecnologia de Zurique (ETH Zurich) adotava o compilador Pascal padrão em seus mainframes, que, entre outras coisas, atendia um número grande e crescente de alunos das disciplina introdutórias de programação. O uso simultâneo por muitos usuários de um compilador completo e a execução dos programas por ele gerados consumia recursos computacionais excessivos para aquela época. Visando contornar este problema, o Prof. Nikalus Wirth, criador da Linguagem Pascal, desenvolveu o compilador e interpretador Pascal S (Pascal Subset), para trabalhar com um subconjunto da Linguagem Pascal.

A escolha dos tipos de dados suportados pelo compilador foi importante para desenvolver um compilador que auxiliasse no ensino da programação, porém com um custo computacional reduzido em relação ao compilador Pascal padrão. O compilador suporta os tipos de dados primitivos inteiro (*integer*), real (*real*), lógico (*boolean*) e caractere (*char*) e tipos de dados compostos arranjo (*array*) e registro (*record*). Embora o tipo *string* não esteja presente, é possível usar *strings* literais dentro dos comandos *write* e *writeln*.

O Pascal S suporta as mesmas estruturas de controle de fluxo de programa do Pascal padrão, com exceção das sentenças *with* e *goto*, sendo que esta última foi omitida para incentivar boas práticas de programação aos programadores iniciantes. Ele suporta procedimentos e funções, inclusive recursivos, sete níveis de escopo e passagem de parâmetros por referência. Assim como não há suporte a ponteiros, também não é possível passar funções e procedimentos como parâmetros [WIRTH 1975].

O Pascal S utiliza a técnica descendente recursiva para realizar a análise sintática, os símbolos do código fonte são reconhecidos através do procedimento **insymbol**. Os símbolos reconhecidos são armazenados, um a um, na variável global **sy**. As variáveis utilizadas para armazenar os valores literais lidos são: **id** para identificadores, **rnum** para números reais, **inum** para números inteiros e caracteres.

O processo de compilação é subdividido em duas partes: Declarações e

processamento de instruções e expressões. No processamento de declarações a tabela de símbolos é gerada e posteriormente usada processar instruções e expressões. A tabela de símbolos do compilador é armazenada na estrutura **tab**, mas também existem tabelas auxiliares: **btab** para blocos e registros e **atab** para arranjos.

A estrutura **tab** é um arranjo de registros contendo atributos ou informações sobre os identificadores presentes no código fonte. Os principais campos de cada registro são **obj**, **adr**, **lev**, **typ**, **ref**, **normal** e **name**.

O campo **obj** indica qual tipo de objeto do identificador: constante, variável, tipo, procedimento ou função. O campo **typ** armazena o tipo de dado associado aos identificadores, que caso sejam do tipo arranjo ou registro, o campo **ref** armazena referências às estruturas **atab** e **btab**, respectivamente. O campo **normal** identifica se um identificador armazena referência a uma posição de memória, utilizado em passagens de parâmetro por referência. O campo **lev** armazena o nível de escopo do identificador, O campo **adr** armazena diferentes valores de acordo com o tipo de objeto do identificador, no caso de variáveis, armazena a posição de memória no nível de escopo. Quando o identificador é uma constante de tipo inteiro, caractere ou lógico, armazena seu valor, para valores do tipo real, o campo **adr** armazena uma referência à estrutura de números constantes reais **rconst**, para procedimentos e funções, armazena o índice da primeira instrução do subprograma.

A estrutura **atab** armazena informações sobre cada arranjo definido no código fonte, os campos **eltyp** e **size** armazenam o tipo dos elementos do arranjo e seu tamanho total em *bytes* , respectivamente. Os limites inferior e superior do índice são armazenados nos campos **low** e **high** e o tipo de dado associado ao índice é armazenado no campo **inxtyp**.

Procedimentos, funções e registros encontrados no código tem informações extras armazenadas na varíavel **btab**, um arranjo de registros contendo os campos **last** (referência ao último campo do registro), **lastpar** (referência ao último parâmetro formal de um precedimento ou função) e **vsize** (tamanho total da estrutura em *bytes*).

Como código gerado para o interpretador, o compilador produz um conjunto de instruções que são armazenados em um arranjo de registros contendo um campo **f** (número entre 0 e 63, representando a instrução a ser executada) e dois campos auxiliares do tipo inteiro, **x** e **y**.

As instruções 0, 1, 2 e 3 utilizam os dois campos auxiliares, **x** e **y**. As instruções

de valor entre 8 e 30 utilizam apenas o campo **y**. As instruções entre 31 e 63 não utilizam campos auxiliares.

A necessidade de construir um compilador compacto, rápido e computacionalmente “leve” impôs algumas limitações como:
a) identificadores e palavras chaves são convertidos para minúsculo;
b) os identificadores são limitados a 10 caracteres;
c) a tabela de símbolos está limitada a 100 elementos.

CAPÍTULO 3

# DESENVOLVIMENTO DO PROJETO

Neste capítulo são descritas as tecnologias utilizadas na implementação do projeto, a especificação da linguagem suportada pelo compilador, alguns exemplos de código e as interfaces de utilização do compilador e do depurador.

## 3.1 Especificação do Dialeto Proposto

O português estruturado, ou Portugol, pode variar conforme o autor, mas sempre segue a linha de tradução livre da Linguagem Pascal com palavras reservadas, tipos de dados e comandos de entrada e saída traduzidos do inglês para o português. Sendo derivada de uma linguagem construída com base no paradigma estrutural, o português estruturado possui as estruturas básicas de decisão e de repetição das principais linguagens de programação.

O dialeto do Portugol desenvolvido possui duas estruturas de decisão, três estruturas de repetição e contempla a modularização dos programas por meio de funções e procedimentos, inclusive recursivos. A estrutura de blocos é a mesma do Pascal, porém há distinção entre letras minúsculas e maiúsculas para os identificadores e o ponto e vírgula não é necessário e nem admitido no final das sentenças.

### 3.1.1 Estrutura Léxica da Linguagem

A estrutura léxica é especificada através da notação EBNF, na qual os símbolos entre **{** e **}** podem ser repetidos, os símbolos envolvidos por **[** e **]** são opcionais, um conjunto de caracteres em sequência da tabela ASCII são especificados através de .. e os caracteres em negrito representam os símbolos terminais da linguagem.

Nesta notação, deve haver espaço simples, tabulação ou quebra de linha entre

duas sequências de caracteres que representem identificadores, palavras reservadas ou valores literais, sendo que em outros casos todos os espaços são desconsiderados. A sequência **(*** delimita o início de um comentário e os caracteres ***)** delimitam o seu final.

Os identificadores podem ter qualquer tamanho, devendo começar por uma letra ou pelo caractere _, os caracteres restantes podem ser letras ou números, o compilador diferencia letras maiúsculas de minúsculas; as palavra contantes na Tabela 7 não podem ser usadas como identificadores, na Tabela 6 tem-se a especificação gramatical de um identificador.

**Tabela 6:** Especificação de um identificador.

```
<identificador> ::= L { L | N }
L ::= a..z | A..Z | _
N ::= 0..9
```

As palavras reservadas são cadeias de caracteres que têm um significado especial e estrito para o compilador, portanto não podem ser usadas como identificadores. Elas estão listadas em ordem alfabética na Tabela 7.

**Tabela 7:** Palavras reservadas do sistema.

| | | | | |
|---|---|---|---|---|
| arranjo | ate | caso | const | de |
| div | e | enquanto | entao | faca |
| fim | funcao | inicio | mod | nao |
| ou | para | passo | procedimento | programa |
| ref | registro | repita | se | senao |
| tipo | var | | | |

O compilador também reconhece valores literais do tipo inteiro, real, caractere, string, lógico e ponteiro, literais do tipo string ou caractere são delimitados por aspas simples, os literais lógicos são **falso** e **verdadeiro**, enquanto **nulo** é o valor literal de ponteiro que indica ponteiro não referenciado, o compilador reconhece qualquer caractere da tabela ASCII para strings literais, o símbolo C representa os valores decimais de todos os caracteres reconhecidos com exceção de aspa simples. Segue a especificação dos valores literais suportados por cada tipo de dado.

```
1 C ::= 32..255-39
2 N ::= 0..9
3 inteiro ::= [+|-] N {N}
4 real ::= [+|-] N{N}  [. {N} [e[+|-] {N}]] | e[+|-] {N}
5 caractere ::= 'C'
```

```
6 string ::= '{C}'
7 logico ::= verdadeiro | falso
8 ponteiro ::= nulo
```

Os operadores aritméticos, relacionais e lógicos reconhecidos pelo compilador encontram-se listados na Tabela 8.

**Tabela 8:** Operadores da Linguagem.

| | |
|---|---|
| **+** | adição ou concatenação |
| **-** | subtração |
| * | multiplicação |
| / | divisão real |
| **div** | divisão inteira |
| **mod** | resto da divisão entre inteiros |
| **=** | igual a |
| **<>** | diferente de |
| **<** | menor que |
| **>** | maior que |
| **<=** | menor ou igual a |
| **>=** | maior ou igual a |
| **e** | e lógico |
| **ou** | ou lógico |
| **nao** | não lógico |

O operador unário ^ é usado após uma variável do tipo ponteiro para desreferenciá-la e o operador unário @ é usado antes de uma variável para retornar um ponteiro sem tipo, cujo valor é o endereço da variável na memória.

### 3.1.2 Declarações

Antes de serem referenciados ou usados, os identificadores devem ser declarados na seção de declarações, localizada após o cabeçalho do programa e antes de procedimentos funções e a palavra reservada **inicio**, ou seja, antes de qualquer instrução de execução nestes módulos. Nesta seção de declaração, os elementos devem ser declarados obrigatoriamente nesta ordem: constantes, tipos, variáveis e procedimentos ou funções.

A declaração de constantes não exige a definição explícita de um tipo de dado, porque o compilador reconhece automaticamente o tipo associado ao valor literal. A palavra reservada **const** deve ser utilizada no inicio do bloco de declarações de constantes e a atribuição de um valor a uma constante é feito utilizando o caractere **=**,

a especificação em EBNF da declaração de constantes é:

```
1 const
2   {<identificador> = <literal>}+
```

O compilador suporta a definição de novos tipos de dados baseados em um tipo de dado preexistente ou já definido, conforme a sintaxe:

```
1 tipo
2   {<identificador> = <tipo>}+
```

As variáveis devem ser declaradas, juntamente com o tipo de dado a qual pertence, na seção de declarações do módulo, o que lhes confere nível de escopo estático, consequentemente é possível a coexistência de variáveis homônimas, desde que declaradas em escopos ou módulos de código diferentes. São permitidos até 10 níveis de escopo ou abrangência, sendo que o programa principal tem o nível de espoco 1 e cada subprograma ou modulo declarado dentro do outro tem um nível de escopo a mais que o modulo ascendente hierárquico mais próximo.

Conforme a notação EBNF abaixo, a seção de declaração de variáveis é iniciada com a palavra reservada **var**, seguida por uma ou mais variáveis separadas por vírgula e, na sequência o símbolo **:** e o identificador de tipo de dado.

```
1 var
2   { <identificador> + { , <identificador> } : <tipo>}+
```

Procedimentos e funções, que são módulos de código também chamados de subprogramas, permitem realizar uma boa prática de programação denominada modularização, que consiste em dividir o código do programa em partes menores, facilitando o entendimento e a manutenção do mesmo.

Tal como em Pascal, por padrão, os parâmetros formais de subprograma são passados por valor, mas é possível passá-los por referência precedendo o parâmetro pela palavra reservada **ref** (equivalente **var** em Pascal). Qualquer expressão que retorne um tipo compatível com um parâmetro por valor pode ser passado ao invocar o subprograma, já na passagem por referência é exigida uma variável com tipo igual ao do parâmetro formal.

Funções são subprogramas que retornam um valor associado a um tipo como

resultado, enquanto procedimentos não retornam valor. Eis suas sintaxes de declaração:

```
1  <declara_procedimento> ::=
2  procedimento <identificador> [ ( [ ref ] <identificador>
3  { , <identificador>} : <tipo> + { ; [ ref ] <identificador>
4  {,<identificador>} : <tipo>} ) ]
5  <declara_var>
6  inicio
7    { <instrução> }+
8  fim
9
10 <declara_função> ::=
11 funcao <identificador> [ ( [ ref ] <identificador>
12 { , <identificador>} : <tipo> { ; [ ref ] <identificador>
13 {,<identificador>} : <tipo> } ) ] : <tipo>
14 <declara_var>
15 inicio
16   { <instrução> }+
17 fim
```

Toda função deve conter um tipo de retorno declarado em seu cabeçalho e realizar o retorno de um valor para a rotina que chamou a função, para isso é utilizado o comando **retorne**, que finaliza imediatamente a execução da função e retorna ao ponto de execução no qual foi invocada. Considere o exemplo abaixo de uma função que retorne o dobro do valor informado como parâmetro: e

```
1 funcao dobra (n: inteiro): inteiro
2 inicio
3   retorne 2 * n
4 fim
```

Diferentemente de algumas linguagens de programação como C e até mesmo Pascal, funções não podem ser invocadas como procedimentos, ou seja, ela deve fazer parte de uma expressão. Caso uma função não tenha parâmetros, o compilador não permite que ela seja invocada usando parênteses após seu nome.

### 3.1.3 Tipos de Dados

Programas de computador geram resultados através do processamento de dados, portanto é crucial para uma boa linguagem de programação que ela seja capaz de manipular diversos tipos de dados, sejam eles primitivos ou definidos pelo programador. Tipos de dados primitivos são aqueles suportados nativamente pelas

linguagens de programação, alguns compiladores permitem a definição de novos tipos de dados ou de tipos de dados agregados, a partir de tipos primitivos e também daqueles já definidos.

Além dos tipos de dados inteiro, real caractere, lógico, arranjo e registro, suportados no Pascal S, foi implementado o suporte ao tipo de dado *string*, para processamento de cadeia de caracteres, e ao tipo de dado ponteiro, para permitir a alocação dinâmica de memória e a definição de estruturas de dados dinâmicas mais complexas como listas encadeadas e árvores. Na Tabela 9 são mostrados os tipos de dados primitivos, seus tamanhos e faixas representáveis.

**Tabela 9:** Tipos de Dados.

| **Tipo** | **Tamanho** | **Faixa representável** |
|---|---|---|
| inteiro | 32 bits | -2 147 483 648 a 2 147 483 647 |
| real | 64 bits | $2.2 * 10^{-308}$ a $1.7 * 10^{308}$ |
| lógico | 8 bits | **verdadeiro** ou **falso** |
| caractere | 8 bits | 0 a 255 |
| string | 1024 bits | até 255 caracteres |
| ponteiro | 32 bits | 0 a 2 147 483 647 |

### Inteiro

Em geral, as linguagens de programação compiladas disponibilizam tipos inteiro nativos de 8, 16, 32, e 64 *bits*, sem e/ou com sinal, para o qual é utilizado o bit mais significativo. A presença do sinal e a consequente utilização de um bit não altera a quantidade de números representáveis, mas reduz a metade o valor do maior inteiro representável. Neste projeto optou-se pela disponibilização exclusiva do tipo inteiro de 32 *bits* com sinal.

### Real

O tipo real ou ponto flutuante é utilizado para armazenar números reais e normalmente é disponibilizado nos formatos de precisão simples (32 *bits*), precisão dupla (64 *bits*) e precisão extendida (80 *bits*). No projeto deste compilador, optou-se por armazenar números reais com precisão dupla, sendo 11 *bits* reservados para armazenar o expoente, 52 *bits* para armazenar a mantissa e 1 *bit* para armazenar o sinal da mantissa.

### Lógico

O tipo de dado lógico, ou *booleano*, é utilizado para armazenar apenas dois valores, verdadeiro ou falso. Algumas linguagens de programação como C permitem que expressões numéricas sejam usadas como lógicas onde o valor 0 é falso e qualquer valor diferente de 0 é reconhecido como verdadeiro. Neste compilador, uma variável do tipo lógico é armazenada na forma de um dado inteiro de 8 *bits* sem sinal, onde 0 é falso e 1 verdadeiro.

### Caractere

O tipo de dado caractere utiliza 8 *bits* de memória para armazenar um dado inteiro, sem sinal, que representa o índice de um caractere da tabela ASCII, cujos 128 primeiros caracteres são padronizados internacionalmente e representam o mesmo símbolo gráfico.

### String

O tipo *string*, ou cadeia de caracteres, se destina ao armazenamento de uma sequência de caracteres e há várias abordagens para sua implementação, como tamanho estático, tamanho variável limitado e tamanho variável ilimitado.

Neste projeto, optou-se pela implementação adotada em Pascal padrão para o tipo *string*, que usa um arranjo de 256 caracteres (1024 *bits*), em cuja posição zero se registra o número de caracteres armazenados a partir da posição 1 até 255. Em Pascal é possível definir-se subtipos *string*, com capacidades menores, de 1 até 254 caracteres, mas neste projeto a capacidade máxima é sempre 255 caracteres.

Um caractere em uma dada posição de uma *string* pode ser acessado utilizando-se *str*[pos], em que *str* é o identificador da *string* e *pos* é a posição, que deve ser positiva para acesso da esquerda para direita e negativo para acesso sentido inverso (direita para esquerda), com valor -1 para o último caractere. O interpretador gera um erro de execução, caso seja acessada a posição zero ou qualquer posição que seja maior que o número de caracteres da *string*.

### Ponteiro

O tipo de dado ponteiro é utilizado para referenciar endereços de memória, valores inteiros de 32 *bits*, geralmente a posição de uma variável ou de um bloco de memória alocado dinamicamente.

Implementou-se o tipo denominado ponteiro para ser usado com ponteiros não tipificados, enquanto os ponteiros tipificados podem ser declarados utilizando-se o símbolo ^ antes do identificador de tipo a que se referem.

O operador @ retorna o endereço da variável à sua direita e o operador ^ retorna o conteúdo apontado pelo ponteiro tipificado à sua esquerda, operação conhecida como desreferenciação. Enquanto não lhe for atribuído algum valor, um ponteiro armazena o valor literal nulo, o que significa uma referência inválida.

No exemplo abaixo, ilustra-se como declarar uma variável, atribuir um endereço memória a uma variável ponteiro e desreferenciar este endereço:

```
1 var
2   dado: real
3   ptr: ^real
4 inicio
5   ptr := @dado
6   ptr^ := 100.50
7   escreva(dado)    (*100.50*)
8 fim
```

### Arranjo

Arranjo é um tipo composto ou estruturado homogêneo de dado para armazenar valores associados a um mesmo tipo de dado, que são dispostos sequencialmente na memória e, portanto, podem ser acessados em tempo constante por meio dos índices ou posições que ocupam dentro da estrutura.

Algumas linguagens de programação especificam o limite inferior do arranjo como 0, logo, os índices do arranjo vão de 0 até tamanho - 1, em Pascal e neste compilador é possível especificar o limite inferior do índice, que podem ser, inclusive, valores negativos.

Os arranjos podem armazenar elementos de tipos primitivos ou definidos pelo usuário e devem ser declarados conforme a sintaxe em notação EBNF:

```
1 <índice> ::= {N}+ | C
2 <nome-variável> : arranjo[<índice>..<índice> ] de <tipo>
```

### Registro

Registro é um tipo composto estruturado concebido para agregar dados de tipos heterogêneos, que são dispostos sequencialmente na memória e podem ser acessados por meio de um identificador dado a um campo ou compartimento dentro da estrutura. Para declarar uma variável do tipo registro, utiliza-se o seguinte sintaxe especificada na notação EBNF:

```
1 <identificador> : registro
2 { <identificador> { , <identificador> } : <tipo>}
3 fim
```

Para acessar um determinado elemento ou campo de um registro, utiliza-se o caractere “ponto” e o nome do campo. A seguir, é mostrado um exemplo do registro aluno, contendo os campos nome, media e id declarados nas linhas 1 a 5, as linhas 7 a 9 fazem o acesso aos campos do registro e atribui valores.

```
1 aluno: registro
2   nome: string
3   media: real
4   id: inteiro
5 fim
6
7 aluno.nome := 'Aluno 1'
8 aluno.media := 6.7
9 aluno.id := 42992
```

## 3.1.4 Estrutura Sequencial

A estrutura sequencial, delimitada pelas palavras reservadas **inicio** e **fim**, define um bloco de instruções que são executados sequencialmente, uma após a outra, ainda que uma delas seja uma instrução de desvio.

O bloco de instruções principal é definido pela palavra reservada **programa**, segue abaixo a notação em EBNF do bloco de instruções principal.

```
1 programa  <identificador>
```

```
[<declarações>]
inicio
  <instruções>
fim

<instruções> ::= {<instrução>}

<instrução ::= <intrução_se> | <instrução_para> |
<instrução_enquanto> | <instrução_caso> | <instrução_repita> |
<instrução\_procedimento> | <atribuição>

<declarações> ::= <declara_tipo> |  <declara_const> |
<declara_var> | <declara_procedimento> | <declara_função>

<bloco_instruções> ::= <instrução> |
inicio <instrução> {<instrução>} fim
```

### 3.1.5 Estrutura Condicional

A estrutura condicional permite ao programa avaliar uma condição (expressão lógica ou relacional) e decidir se executa um, nenhum, ou grupo alternativo de ações.

#### Instrução se

A estrutura de decisão **se** avalia uma expressão relacional ou lógica, caso o resultado seja verdadeiro, executa o primeiro bloco de código. Opcionalmente pode-se definir um bloco de código a ser executado caso a expressão avaliada retorne um valor falso, este segundo bloco é declarado inserindo-se a cláusula **senao**.

Segue abaixo a especificação a notação em EBNF da estrutura se:
<instrução_se> ::=

```
se <condição> entao
  <bloco_instruções>
[senao  <bloco_instruções>]
```

#### Instrução caso

A estrutura de decisão **caso**/**de** avalia o valor presente em uma determinada expressão e compara com rótulos (valores literais dos tipos inteiro, caractere ou lógico)

especificados pelo programador; caso o valor resultante na expressão analisada seja igual ao valor de um destes rótulos, o bloco de instruções do rótulo é executado.

Os rótulos precisam ser constantes e podem ter três tipos: inteiro, caractere ou lógico. A expressão analisada e os rótulos precisam ser do mesmo tipo, não é permitido a declaração de dois ou mais rótulos iguais, mas é permitida a declaração de um mesmo bloco de instruções para mais de um rótulo. A cláusula facultativa **senao** pode ser utilizado no final da estrutura para especificar um bloco de instruções a ser executado caso o valor da variável não coincida com nenhum rótulo. Caso algum bloco de instruções dentro da estrutura **caso** seja executado, nenhuma outra verificação é feita, e ao final do bloco a estrutura é finalizada.

Segue abaixo a notação em EBNF da estrutura **caso**:

```
1 caso (<expressão>) de
2     {<rótulo>{,<rótulo>}: <bloco_instruções>}
3     senao <bloco_instruções>
4 fim
5 <rótulo> ::= inteiro | caractere | logico
```

### 3.1.6 Estruturas de Repetição

As estruturas de repetição, ou comandos estruturados de repetição, permitem repetir a execução de um trecho de código por um número pré-fixado (ou determinável) de vezes, ou podem ser controlados por uma condição.

#### Instrução para

O comando estruturado **para** utiliza uma variável de controle do tipo inteiro, cujo valor inicial resulta da expressão à esquerda de **ate** e cujo valor final não ultrapassa o resultado da expressão à direita de **ate**. Se a palavra **passo** for omitida, a variável é incrementada em 1 a cada iteração, mas caso esteja presente, pode-se definir qualquer valor positivo ou negativo, como incremento ou decremento, respectivamente. Quando o passo for positivo e o valor final for menor que o inicial não ocorre nenhuma iteração, o que também acontece quando o passo for negativo e o valor final for maior que o inicial.

```
1 para <var> de <expressão> ate <expressão> [passo <valor>] faca
2   <bloco_instruções>
```

#### Instrução enquanto

O comando estruturado **enquanto** avalia uma condição, representada por uma expressão relacional ou lógica, cujo valor verdadeiro implica na execução do código delimitado pela estrutura e em nova avaliação da condição. Quando a condição resultar em falso, o que pode acontecer já no início, a execução passa para a instrução seguinte à estrutura do comando.

```
1 enquanto <expressão> faca
2   <bloco_instruções>
```

#### Instrução repita

O comando estruturado **repita** repete um bloco de código até que uma condição seja verdadeira e, como a condição é testada ao final, o referido código sempre será executado pelo menos uma vez.

```
1 repita
2  { <instrução> }+
3 ate (<expressão>)
```

### 3.1.7 Instruções de Entrada e Saída

Neste compilador foi implementada uma interface para a entrada e a saída de dados denominada Janela do Console, pela qual os dados podem ser digitados e lidos pelo comando **leia** e na qual os dados podem ser exibidos pelos comandos **escreva** e **escrevaln**. O comando **leia** aceita uma lista de variáveis, separadas por vírgula, e requer o pressionamento da tecla <Enter> após a digitação de cada valor. O comando **escreva** imprime na tela uma ou mais expressões que resultem em valores pertencentes a tipos atômicos. Ao se deparar com a sequência de caracteres \n em uma *string*, o comando **escreva** passa a escrever o restante da saída na linha seguinte. Para quebrar a linha ao final da escrita, pode-se usar o comando **escrevaln**. Eis as sintaxes em EBNF dos referidos comandos:

```
1 escreva ([<expressão>|'{ C }']{ , <expressão>|'{ C }'})
2 escrevaln ([<expressão>|'{ C }']{ , <expressão>|'{ C }'})
```

```
1 leia ( <identificador> { , <identificador> } )
```

### 3.1.8 Subprogramas predefinidos

Subprogramas predefinidos são rotinas nativas do compilador para simplificar tarefas frequentes como o processamento de *strings*, o uso de funções matemáticas, a conversão entre tipos de dados, o gerenciamento de memória, entre outras. Nas Tabelas 10, 11, 12 e 13 são mostrados os cabeçalhos ou assinaturas destes subprogramas, bem como as descrições de cada um.

**Tabela 10:** Cabeçalhos de funções matemáticas

| **Cabeçalho** | **Descrição** |
|---|---|
| abs (x: inteiro): inteiro<br>abs (x: real): real | Retorna o valor absoluto do número. |
| sqr (x: inteiro): inteiro<br>sqr (x: real): real | Retorna o quadrado do número. |
| impar (x: inteiro): logico | Retorna verdadeiro se o número é impar. |
| chr (x : inteiro): caractere | Converte um valor inteiro em caractere. |
| ord (x: caractere): inteiro | Converte um caractere em inteiro. |
| succ (x: caractere): caractere | Retorna o caractere sucessor ao informado. |
| pred (x: caracetere): caractere | Retorna o caractere antecessor ao informado. |
| arred (x: real): inteiro | Arredonda o número real para o inteiro mais próximo. |
| trunca (x: real): inteiro | Retira a parte fracionária do valor real. |
| seno (x: real): real | Calcula o seno do ângulo informado. |
| cos(x: real) : real | Calcula o cosseno do ângulo informado. |
| exp (x: real): real | Calcula o exponencial do valor informado. |
| log (x: real): real | Calcula o logaritmo do valor informado. |
| raizq (x: real): real | Calcula a raiz quadrada do valor informado. |
| arctan (x: real): real | Calcula a arco tangente do ângulo informado. |

Para processamento de strings existe uma lista de subprogramas implementados que são descritos na Tabela 11 juntamente com a sintaxe e a finalidade de cada um. As funções **strmax**, **strmin** retornam uma string modificada mas não modifica a string passada por parâmetro, os procedimentos **strins** e **strdel** modificam a string passada por parâmetro.

No gerenciamento de memória a função **aloca** recebe como parâmetro uma variável, ou um tipo de dado ou um valor inteiro representando o número de *bytes* a serem alocados e retorna um ponteiro referenciando o bloco da memória alocado. O procedimento **desaloca** recebe um ponteiro como parâmetro e libera a região da memória referenciada por ele.

Ainda existem mais duas funções predefinidas do sistema para medição de tempo e geração de números aleatórios. A função **aleatorio** pode receber um ou nenhum

**Tabela 11:** Cabeçalhos de sub-programas para operações com string

| **Cabeçalho** | **Descrição** |
|---|---|
| strmax(str: string):string | Retorna a string convertida para maiúsculo. |
| strmin (str: string): string | Retorna a string convertida para minúsculo. |
| strtmo (str: string): inteiro | Retorna o número de caracteres da string. |
| strbusca (str: string; substr: string): inteiro | Retorna a posição de substr em str ou zero, caso não encontre. |
| strins(ref: string; pos: inteiro; substr: string) | Insere substr na posição pos de str. |
| strdel (ref:string; pos: inteiro; n: inteiro) | Exclui n caracteres de str, a partir da posição pos. |

**Tabela 12:** Cabeçalhos de sub-programas para gerenciamento de memória.

| **Cabeçalho** | **Descrição** |
|---|---|
| bytes (ptr: ponteiro): inteiro | Retorna o número de bytes requeridos por *ptr^*. |
| bytes (var: tipo): inteiro | Retorna o número de bytes utilizados por *var*. |
| bytes (tpo: tipo): inteiro | Retorna o número de bytes associados ao tipo *tpo*. |
| aloca (ptr: ponteiro): ponteiro | Aloca um bloco de memória de tamanho bytes(*ptr*). |
| aloca (tpo: tipo): ponteiro | Aloca um bloco de memória de tamanho bytes(*tpo*). |
| aloca (tam: inteiro): ponteiro | Aloca um bloco de memória de tamanho *tam*. |
| desaloca (ptr: ponteiro ) | Desaloca o bloco de memória referenciado por *ptr*. |

parâmetro, caso seja chamada sem parâmetros, ela retorna um valor aleatório maior ou igual a 0 e menor que 1 e caso seja chamada com um número inteiro, ela retorna um valor aleatório maior ou igual a 0 e menor que o valor informado. A função **tempo** retorna os 31 bits menos significativos da quantidade de milissegundos desde 1 de janeiro de 1970. Na Tabela 13 estão descritos os cabeçalhos das duas funções. O procedimento **semente** modifica a semente utilizada pela função aloca para a geração de valores pseudoaleatórios.

**Tabela 13:** Funções para geração de números aleatórios e tempo.

| **Cabeçalho** | **Descrição** |
|---|---|
| aleatorio: real | Retorna um valor aleatório em [0, 1). |
| aleatorio(n: inteiro): inteiro | Retorna um valor aleatório em [0, n). |
| semente(s: inteiro) | Altera para s a semente da funçao aleatório. |
| tempo: inteiro | Retorna o tempo relativo em milissegundos. |

### 3.1.9 Avaliação de Expressões

Expressões são o meio fundamental de processar dados computacionais, geralmente são divididas em três categorias: aritmética, lógica ou relacional. Essas categorias dizem respeito a quais operadores são utilizados e quais tipos de dados se esperam como operandos e como resultado. Os operadores podem ser unários, binários ou ternários, essa variação corresponde a quantos operandos o operador necessita para realizar a operação, a lista de operadores está descrita na Tabela 5, o compilador objeto deste trabalho não implementa operadores ternários [SEBESTA 2003].

A sintaxe em notação ENBF das expressões suportadas pelo compilador está descrita abaixo:

```
1  <expressão> ::= <expressão_simples>
2  [<relação> <expressão_simples>]
3
4  <relação> ::= = | <> | > | < | >= | <=
5
6  <expressão_simples> ::= [+|-] <termo> { (+|-|ou) <termo>}
7
8  <termo> ::= <fator> { ( * | / | div | e | mod) <fator>}
9
10 <fator> ::= [ @ ]<variável> [ ^ ] |
11
12 <valor> | ( <expressão> ) | nao <fator>
13
14 <variável> ::= <identificador>
15
16 <valor> ::= 1 | 4.4 | verdadeiro | 'C'
```

#### Precedência de Operadores

Um aspecto importante para a avaliação de uma expressão é a ordem em que os operadores são avaliados, sendo que na maioria das linguagens de programação, a ordem de precedência é derivada da matemática. É importante se atentar para a ordem precedência dos operadores em expressões aritméticas pois é comum erros de resultados de expressões pela não observância desta ordem e a consequente ausência ou mal posicionamento de parênteses. Na tabela 14 é listada a ordem de avaliação dos operadores, cada linha possui uma precedência maior que as linhas

inferiores, caso exista operadores de mesma ordem de precedência na expressão, o operador mais à esquerda é processado primeiro.

**Tabela 14:** Ordem de Precedência de Operadores.

| Precedência | Operadores |
|---|---|
| Alta | (, ), **nao**, @, ^ |
| | *, /, **div**, **mod**, **e** |
| | +, -, **ou** |
| Baixa | =, <>, <, >, <=, >= |

O operador +, utilizado na adição aritmética e na concatenação de strings, retorna o valor de acordo com o tipo de seus operandos; quando os tipos são diferentes, a operação retorna o tipo que ocupa mais espaço na memória. Por exemplo, a adição de um número real e um inteiro, resulta em um número real. Utilizando-se o operador + para concatenação, a operação deve receber como operandos valores do tipo caractere ou string e sempre retorna um valor do tipo *string*.

O operador "/" retorna sempre o resultado da divisão real, mesmo que ambos os operandos sejam valores inteiros ou mesmo que o divisão não deixe resto. Para a divisão inteira, ou euclidiana, deve-se utilizar o operador **div**, que resulta no quociente de dois operados do tipo inteiro.

A tabela verdade dos operadores lógicos **e**, **ou** e **nao** está ilustrada na Tabela 15.

**Tabela 15:** Tabela Verdade dos Operadores Lógicos

| a | b | a **e** b | a **ou** b | **nao** a | **nao** b |
|---|---|---|---|---|---|
| F | F | F | F | V | V |
| F | V | F | V | V | F |
| V | F | F | V | F | V |
| V | V | V | V | F | F |

## 3.2 Compilador PortugolJS

A partir do compilador e interpretador Pascal S, descrito na seção 2.5, implementou-se o ambiente de desenvolvimento integrado PortugolJS para compilar, executar e depurar códigos escritos no dialeto de Português Estruturado, especificado no Capítulo 3. Nesta seção são abordados os três módulos do sistema: compilador, interpretador e depurador.

O projeto contendo todos os arquivos e código fonte está disponível em: github.com/r4faelferreir4/ide-pseudocodigo.

### 3.2.1 Compilador

No desenvolvimento deste projeto foi necessário implementar modificações no compilador original, bem como novas funcionalidades e tipos de dados, para permitir uma ferramenta mais completa para o ensino da programação de acordo com a grade curricular das universidades brasileiras.

A única funcionalidade retirada do compilador original foi o suporte a arquivos, tendo em vista que a linguagem Javascript, utilizada como base, é limitada quanto à operações com arquivos e impossibilitou a implementação de funções para operações de leitura/escrita no disco rígido.

Por outro lado, foram adicionadas ao compilador funcionalidades importantes como a avaliação mínima, ou de curto circuito, na conjunção e disjunção lógica, o suporte ao tipo de dado string e várias funções nativas para o seu processamento. Porém o acréscimo mais relevante e desafiador foi o suporte ao tipo de dados ponteiros, que possibilida o uso de alocação dinâmica de memória e, principalmente, a definição de estruturas de dados mais complexas. O suporte a alocação dinâmica exigiu a implementação de um mecanismo de gerenciamento de memória, descrito a seguir.

#### Gerenciamento de Memória

A maioria dos compiladores baseados no paradigma estrutural utilizam uma pilha para armazenar os dados do programa, onde os últimos dados a serem armazenados são os primeiros a sair, essa é uma abordagem simples e eficiente mas possui algumas limitações. Uma função poderia retornar um endereço de uma variável local e essa região de memória, por ser temporária, estaria desprotegida e sujeita a alterações indevidas [LOUDEN 2004].

A alocação de memória dinâmica é uma alternativa para solucionar os problemas de uma pilha, pois permite ao programador alocar e liberar memória durante a execução em uma estrutura chamada *heap* e é feita de forma explícita através de subprogramas predefinidos. Neste projeto o gerenciamento de memória é feito com

auxilio da estrutura *Blocks*, que é uma lista de registros com os campos *start*, *size* e *isAvailable* para armazenar o inicio, tamanho e disponibilidade do bloco.

Algumas linguagens implementam o gerenciamento automático do heap ou coleta automática, por meio de técnicas como a contagem de referências e a de marcar e varrer, que também podem movimentar os blocos alocados para uma extremidade do heap, visando evitar a fragmentação [AHO 2008].

Além da complexidade da implementação de um mecanismo eficiente de coleta de lixo e a queda desempenho do interpretador, vislumbrou-se desde o início que a desalocação de memória devia ser feita explicitamente pelo usuário, o que viria a contribuir para consolidar boas práticas de programação.

O interpretador realiza a alocação da memória através da função *MemoryAlloc* que recebe um valor inteiro referente ao número de *bytes* a serem alocados, realiza uma busca linear na procura do primeiro bloco disponível que comporte o número de bytes informado, ao encontrar, marca o bloco como indisponível. Caso o bloco encontrado seja maior que o necessário, então o gerenciador verifica se o bloco anterior está indisponível, se estiver, ele será expandido para acoplar a nova alocação e o bloco disponível que foi encontrado será reduzido.

A função *MemoryFree* recebe como parâmetros o endereço de inicio e o número de *bytes* a serem liberados e realiza uma busca binária na lista de blocos, caso o bloco seja menor que o tamanho de *bytes* a serem liberados a função continua a liberação de forma recursiva, caso o bloco seja maior que o espaço liberado, o gerenciador divide o bloco em duas ou três partes, sempre os blocos de mesma disponibilidade são unificados.

Embora esta forma de implementar o desalocador de memória seja um pouco diferente da padrão, optou-se por ela para reduzir o número de blocos do gerenciador e a utilização de memória. Ela também mostrou-se útil na operação de exclusão intermediária em strings e a consequente liberação de espaço no espaço ao final da mesma.

#### Geração de Números Aleatórios

Para a geração de números aleatórios implementou-se o algoritmo linear congruencial, que é um dos mais antigos e eficientes, inclusive utilizado nos compiladores Delphi e Borland C/C++. A função que retorna um valor aleatório está

descrita abaixo.

X$n$= ($c$ * X$n+1$ + $i$) **mod** $m$

X representa um valor inicial, ou semente, e é modificada a cada número gerado, $c$ representa uma constante numérica, geralmente utiliza-se números primos, $i$ representa o incremento realizado a cada iteração e $m$ representa o módulo para divisão do valor multiplicado, a partir da geração de X$n$, se aplica a seguinte função:

X$n$ / D * N

D representa o maior valor de um inteiro de 32 bits sem sinal, ou seja, 4294967296. Se N = 1, o algoritmo gera valores reais no intervalo [0, 1), mas se N > 1 o algoritmo gera valores inteiros que variam de 0 a N-1, inclusive [VIEIRA 2004].

### Medição do Tempo

Para a medição do tempo, o interpretador utiliza como base uma função nativa do JavaScript chamada Date.now(), que retorna o valor em milissegundos desde 01/01/1970. Como esta função retorna um inteiro de aproximadamente 40 *bits*, a função **time32** considera apenas os 31 *bits* menos significativos do valor retornado, já que o 32° *bit* é representada o sinal de um tipo inteiro.

### Alterações na Sintaxe

Pelo fato de a Linguagem Pascal ter sido idealizada para facilitar o ensino e boas práticas de programação, ela tem estrutura e sintaxe muito parecida com o pseudocódigo estruturado, qualquer que seja o idioma. Por isso, ao se escolher o Compilador Pascal S como base para este projeto, além da simples tradução para o português das palavras reservadas e sentenças em Pascal, as alterações na sintaxe proposta exigiram relativamente poucas mudanças durante a tradução do código do compilador para Javascript.

Uma dessas alterações foi quanto ao ponto e vírgula, que tanto não é mais necessário como não é permitido, tendo em vista que a maioria dos livros de algoritmos não o utiliza para delimitar final de instrução. Outro motivo para a supressão do ponto e vírgula é que a ausência dele, geralmente por esquecimento do programador iniciante, é umas das causas mais comuns de erros detectados pelos compiladores das linguagens de programação que o empregam como finalizador de

sentenças.

A sintaxe da estrutura **para** foi alterada para permitir incrementos e decrementos diferentes de 1, ao inserir-se a palavra reservada **passo**. Alterou-se a também a estrutura de seleção **caso**, adicionando-lhe a possibilidade do uso facultativo da cláusula **senão**, que já existe em Pascal, mas não foi implementada no compilador Pascal S. Se esta cláusula estiver presente e nenhum rótulo for selecionado, então o código delimitado por ela será executado.

#### Avaliação miníma ou de curto circuíto em expressões lógicas

Na operação lógica **e** (conjunção), quando o primeiro operando é falso, o segundo operando não precisa ser avaliado, porque o resultado fica determinado como falso. Analogamente, na disjunção lógica (**ou**), se o primeiro operando é verdadeiro, o resultado é verdadeiro e não depende do valor do segundo operando, que obviamente não precisa ser avaliado. Ao tirar proveito disso, os compiladores evitam não apenas uma avaliação desnecessária, mas também, em muitos casos, evitam causar uma exceção, ao realizar a segunda avaliação. Não fosse a avaliação mínima, a exceção causada pela segunda avaliação só poderia ser evitada com o um comando condicional, aumentando o tamanho e diminuindo a legibilidade do código.

### 3.2.2 Interpretador

O interpretador é o responsável por executar as instruções descritas nas Tabelas 16, 17 e 18, sendo que estas instruções são geradas durante o processo de compilação. Para leitura de dados do usuário, o interpretador é finalizado momentaneamente até que o usuário insira um dado na caixa de entrada e tecle <Enter>, quando então a função do interpretador é chamada novamente sem que as variáveis de controle sejam inicializadas, mantendo-se as mesmas de quando foi finalizado. Desta forma o interpretador retorna à instrução de leitura para armazenar o dado informado pelo usuário na memória. O diagrama de execução do interpretador está ilustrado na Figura 3.

O interpretador utiliza variáveis globais para o controle da execução do programa, por isso ele pode ser parado e retomado de onde parou sem a perda de dados. As principais variáveis de controle são: **kode**, **pc**, **display**, **s**, **ir**, **t**, **ps**, **b**.

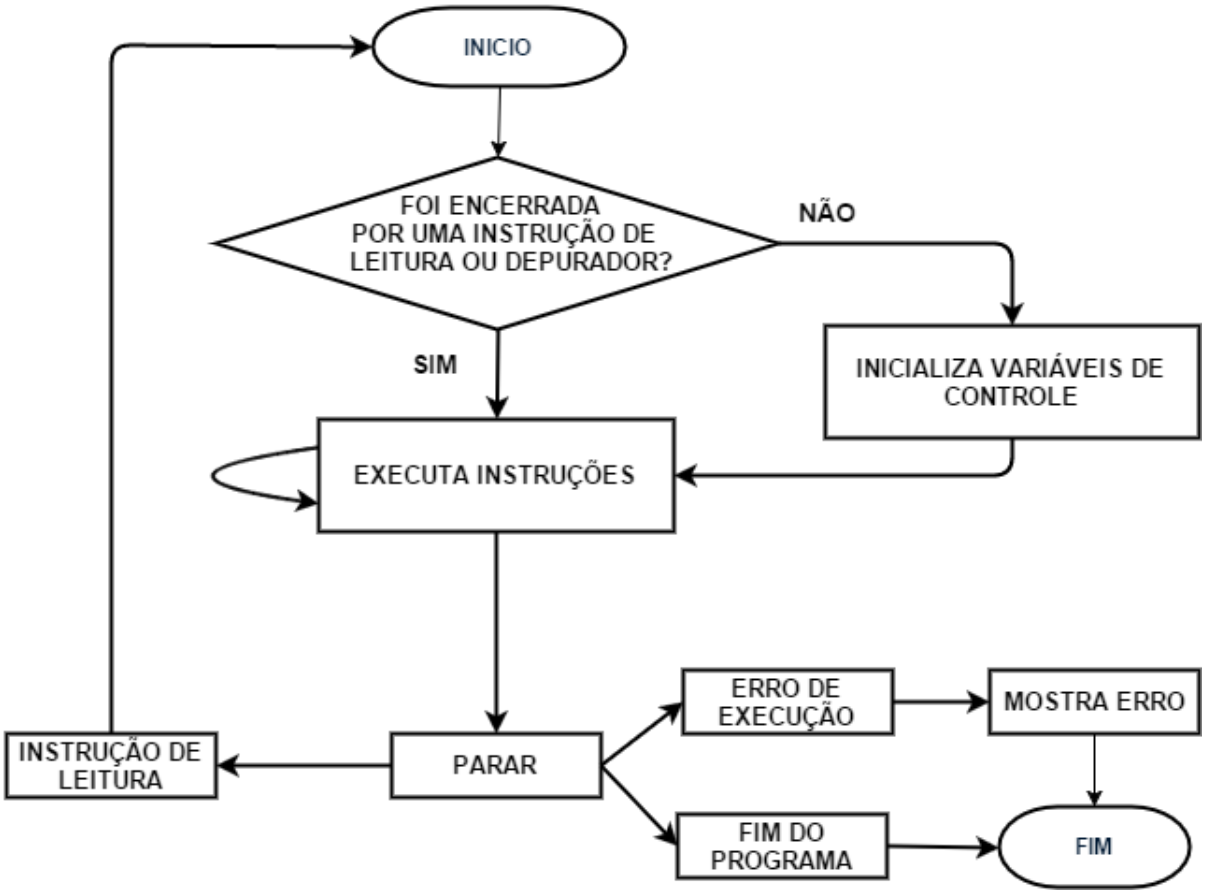


**Figura 4:** Diagrama de Execução do Interpretador

As instruções geradas pelo compilador, no sentido de leitura do código, são armazenadas no arranjo **kode**, cujos elementos são registros do tipo *order*. Durante a interpretação, a variável **pc** mantém o índice da instrução corrente, que após ser executada provoca o incremento de **pc**, com o qual se lê a próxima instrução de **kode** para o registro **ir**.

A variável **display** é um arranjo de números inteiros que armazenam a posição inicial na pilha de execução para cada nível de escopo. O topo da pilha é representado pela variável **t** que armazena a primeira posição disponível para empilhamento de dados, a variável **b** armazena o fim da área base da memória e inicio da área temporária. A variável **ps** armazena o estado de execução do programa e erros.

### Pilha de Execução

Uma das alterações realizadas no interpretador em relação ao original foi na pilha de execução, responsável por armazenar os dados temporários do programa. Na implementação original, foi utilizado um arranjo de registros variantes (os campos se

superpõem) com a seguinte estrutura:

```
registro
    c:caractere
    i:inteiro
    r:real
    b:logico
fim
```

Cada informação era armazenada em uma única posição do arranjo, utilizando um dos quatro campos do registro, dependendo do tipo de dado. O primeiro problema que isso gera é o desperdício de memória, pois para armazenar um dado lógico, de 8 *bits*, por exemplo, eram utilizados 48 *bits* de memória, associados ao campo *r*, que é o maior deles. A nova versão da pilha de execução armazena os dados em um vetor de *bytes*, de modo que cada dado utiliza apenas da quantidade de *bits* necessária.

Outro problema decorrente da pilha de execução implementada no compilador Pascal S é a implementação de alocação de memória para ponteiros, pois dificulta o controle da quantidade de memória realmente alocada, pois cada dado utilizava uma única posição da pilha .

O diagrama de uso da memória está ilustrado na Figura 5 , na qual a **Base** armazena as variáveis globais do programa, portanto tem tamanho variável, que estende até o início da pilha de execução, que é uma área temporária usada no processamento de expressões e no armazenamento de variáveis locais de subprogramas. A pilha de execução se estende até o início do *heap*, cujo tamanho é 5Mbytes (25%) e cuja finalidade é prover uma área de memória para ser alocada dinamicamente, durante a execução.

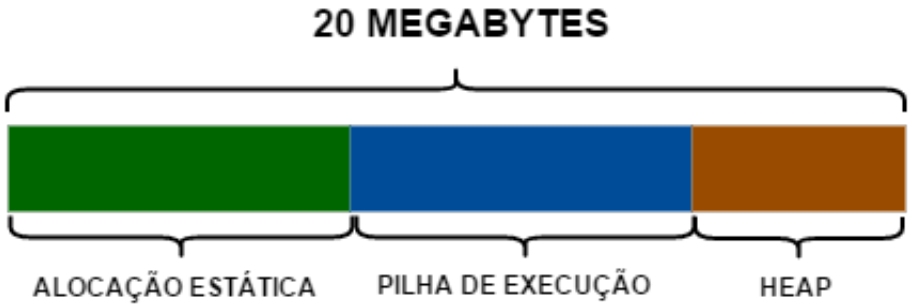


**Figura 5:** Diagrama de uso da memória

### Instruções Suportadas

A estrutura **order**, usada para registrar as informações de cada instrução nos seus campos **f**, **x** e **y**, ganhou o campo **line**, para armazenar a linha de código como informação para depuração e o campo **z** para registrar informações associadas ao tipo de dado que a instrução está utilizando e até mesmo o seu tamanho. As instruções usadas em operações matemáticas estão listadas na Tabela 16.

**Tabela 16:** Instruções de Operações Matemáticas do Interpretador

| Código | Descrição |
|---|---|
| 24 | Carrega valor na pilha. |
| 25 | Converte caractere para *string*. |
| 26 | Converte inteiro para real. |
| 35 | Não lógico.(**nao**) |
| 36 | Muda sinal.(-) |
| 39 | igual.(=) |
| 40 | diferente.(<>) |
| 41 | menor.(<) |
| 42 | menor ou igual.(<=) |
| 43 | maior.(>) |
| 44 | maior ou igual.(>=) |
| 51 | Ou lógico.(**ou**) |
| 52 | Adição ou concatenação.(+) |
| 53 | Subtração.(-) |
| 56 | E lógico.(**e**) |
| 57 | Multiplicação.(*) |
| 58 | Divisão.(/) |
| 59 | Resto da divisão entre inteiros.(**mod**) |
| 68 | Avaliação curto-circuíto ou. |
| 69 | Avaliação curto-circuíto e. |
| 72 | Gera números aleatórios. |
| 73 | Altera semente de geração de números aleatórios. |

O compilador utiliza subprogramas predefinidos para realizar operações com strings, porém também utiliza operadores padrões como o operador + para a concatenação e os caracteres [ e ] para acesso a um caractere específico, as

instruções que realizam as operações estão listadas na Tabela 17.

**Tabela 17:** Instruções de Operações com Strings.

| Código | Descrição |
|---|---|
| 61 | Insere uma substring em uma string. |
| 62 | Seleciona um caractere e uma string. |
| 63 | Escreve um caractere em uma string. |
| 64 | Retorna o tamanho de uma string. |
| 65 | Converte uma string para maiúsculo. |
| 66 | Converte uma string para minúsculo. |
| 67 | Busca uma substring em uma string. |
| 75 | Remove caracteres de uma string. |

Na Tabela 18, estão listadas as instruções de controle do interpretador, que são responsáveis pelo controle de fluxo da interpretação, chamadas de funções, carregamentos de dados e endereços no topo da pilha, a entrada e saída de dados do usuário, alocação e desalocação de memória e medição do tempo de interpretação.

**Tabela 18:** Instruções de Controle do Interpretador.

| **Código** | **Descrição** |
|---|---|
| 0 | Carrega endereço na área temporária. |
| 1 | Carrega valor na área temporária. |
| 2 | Carrega valor referenciado na área temporária. |
| 3 | Atualiza display. |
| 8 | Funções pré-definidas. |
| 9 | Acesso a atributos de um registro. |
| 10 | Desvio incondicional. |
| 11 | Desvio condicional. |
| 12 | Comparação de rótulos da estrutura **caso**. |
| 14, 15 | Incremento na estrutura **para**. |
| 16, 17 | Decremento na estrutura **para**. |
| 18 | Marca pilha para chamada de função. |
| 19 | Chamada de função. |
| 20, 21 | Acesso à posições de arranjos. |
| 22 | Carrega bloco de dados na pilha. |
| 23 | Copia bloco de dados da pilha. |

**Tabela 18:** Instruções de Controle do Interpretador.

| | |
|---|---|
| 27 | Recebe dado do usuário. |
| 28 | Imprime valor literal do tipo string. |
| 29 | Imprime valor armazenado na memória. |
| 30 | Imprime valor com espaçamento. |
| 31 | Fim do programa. |
| 32, 33 | Fim de procedimento ou função. |
| 34 | Desreferencia endereço da pilha. |
| 37 | Imprime valor real com espaçamento e fração. |
| 38 | Armazena valor na base da pilha. |
| 70 | Libera memória. |
| 71 | Aloca memória. |
| 74 | Retorna um tempo relativo em milissegundos. |

### Tela de Execução e Erros

Ao se executar um programa é exibida a Janela de Entrada e Saída de Dados, que pode ser ocultada por meio do botão Esconder, mas que ficará visível novamente se ocorrer uma instrução de leitura de dados. A área de saída de dados, usada pelos comandos **escreva** e **escrevaln**, é representada pela porção em fundo escuro da janela. Logo abaixo dela, uma pequena caixa de texto é usada para leitura de dados por meio do comando **leia**. A referida janela exibindo a saída de um programa que soluciona o problema da Torre de Hanoi é mostrada na Figura 6.

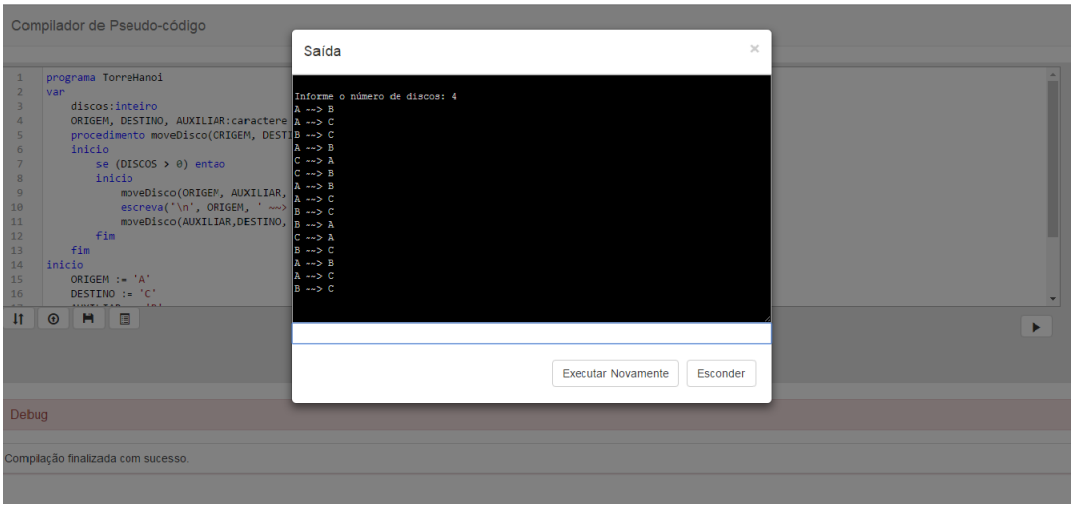


**Figura 6:** Tela de Execução de um Programa

O interpretador também tem a responsabilidade de retornar ao usuário os erros de execução, como por exemplo a divisão por zero. Os erros de execução suportados pelo interpretador estão listados na Tabela 19.

**Tabela 19:** Erros de Execução

| |
|---|
| Divisão por 0 |
| Erro na estrutura caso. |
| Erro no índice de arranjo. |
| Erro de índice da string. Índice maior que o tamanho da string. |
| Erro de índice da string. Índice 0 não existe |
| Estouro de pilha. |
| String literal para impressão muito longa. |

### 3.2.3 Depurador

Como o compilador Pascal S não possui qualquer mecanismo de depuração, foi necessário projetar e implementar o suporte às seguintes funcionalidades durante a execução em modo de depuração:

a) Execução passo a passo entrando em rotinas;

b) Execução passo a passo saindo de rotinas;

c) Execução passo a passo pulando rotinas;

d) Execução até a linha do cursor;

e) Visualização da pilha de chamadas de funções

f) Visualização do número de execuções de cada linha de código;

g) Visualização das variáveis ativas e seus valores;

h) Alteração dos valores de variáveis em tempo de execução;

i) Execução até o final;

j) Finalização da execução.

Todas as funcionalidades citadas acima utilizam como ponto de parada o valor da variável **stopln**, que armazena o número da linha de código referente aos pontos de parada, determinados por cada opção escolhida pelo usuário.

A execução passo a passo entrando em rotinas ou subprogramas é controlada pela função **inRoutine**, que ao verificar que a próxima instrução é a chamada de subprograma, busca o número da primeira linha executável do subprograma e atribui-o à **stopln**. Então o interpretador executa as instruções de alocação de memória, passagem de parâmetro e desvio de fluxo de execução para a primeira instrução do

subprograma chamado, após isso o depurador finaliza a execução do interpretador e aguarda um novo comando. A variável **indebug** sinaliza ao interpretador que o mesmo deve ser executado no modo passo a passo entrando em rotinas.

A função **byRoutine** efetua a execução passo a passo saltando rotinas, ou seja, considera chamadas de subprogramas como instruções únicas. Com o interpretador pausado, ela atribui o número da próxima linha executável do escopo corrente à variável **stopln** e reinicia o interpretador, que executa as instruções normalmente até encontrar uma instrução cuja linha seja igual a **stopln**. A variável **bydebug** indica ao interpretador que ele está sendo executado através da função **byRoutine**.

A função **outRoutine** define a variável **outdebug** como verdadeira e retoma a execução pelo interpretador, que deve executar as instruções normalmente até encontrar os códigos de instrução 31, 32 ou 33, que sinalizam os finais de programa, procedimento e função, respectivamente. Quando encontrados, verifica se o nível de escopo coincide com o definido no momento da execução de **outRoutine** e então após desviar o contador de programa **pc** para a rotina chamadora finaliza o interpretador.

A operação executar até o cursor é feita através da função **RunToCursor**, que obtém o número da linha onde o cursor do editor está posicionado, verifica se há instrução executável nesta linha e, em caso afirmativo, atribui este número à **stopln**. Na sequencia, a variável **CursorRun** é definida como verdadeira, o interpretador é retomado e executa as instruções normalmente até atingir a linha cujo número coincida com o valor de **stopln**.

A função **stopDeb** desvia a execução do programa para a última instrução, armazenada em **finalInst**, e finaliza o interpretador e o depurador. A última linha executada é armazenada em **linecount**, com o avanço da execução e mudança de linha das instruções, a nova linha é incrementada no editor e armazenada como última linha executada na variável **linecount**.

Algumas informações das variáveis ativas como tipo, nome, nível de escopo e endereço são armazenadas na estrutura **arrayObjetoTabela**, por meio da qual a função **atualizaVariavel** carrega os valores informados nos campos de texto do depurador e atualiza os valores na pilha de execução.

A Figura 7 ilustra um programa para armazenar uma sequência de caracteres em uma lista encadeada sendo executado passo a passo pelo depurador, que pode ser iniciado pelas teclas de atalho <F7> ou <F8>.

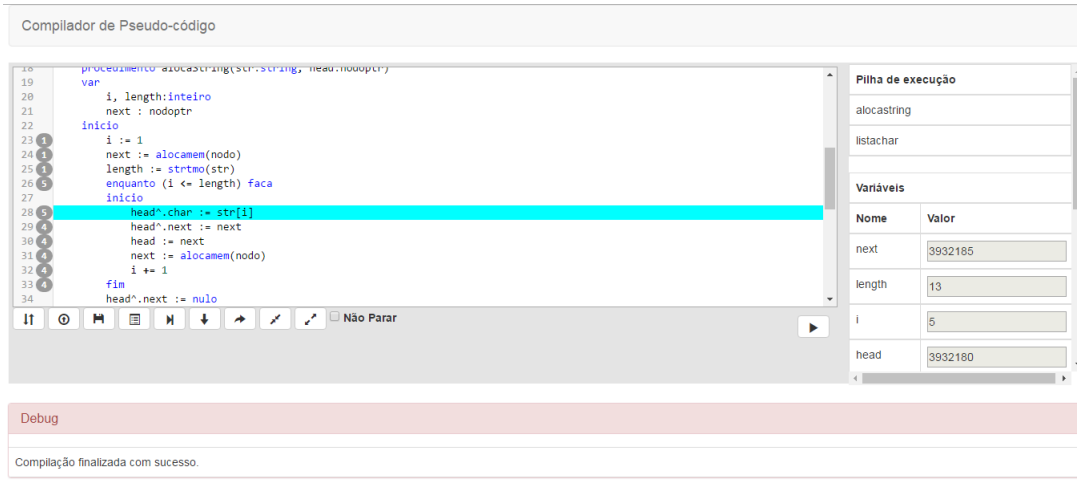


**Figura 7:** Modo de depuração do compilador.

Na Tabela 20 estão listadas as teclas de atalho que podem ser usadas para simplificar e agilizar o acesso às funcionalidades do depurador.

**Tabela 20:** Teclas de Atalho de Depuração.

| Tecla de Atalho | Função |
|---|---|
| F6 | Inicia modo de depuração. |
| F4 | Executa até o cursor. |
| F7 | Executa passo a passo entrando em rotinas. |
| F8 | Executa passo a passo pulando rotinas. |
| Ctrl + F8 | Executa passo a passo saindo de rotinas. |
| Ctrl + F9 | Executar até o final. |
| Ctrl + F2 | Finalizar depuração. |

## 3.3 Ferramentas Utilizadas

Para o desenvolvimento deste trabalho, foram utilizadas algumas ferramentas para programação *web* e *frameworks* da linguagem Javascript. Optou-se pela implementação deste compilador na linguagem Javascript pelos seguintes motivos:

a) É uma linguagem para desenvolvimento *web*;

b) Todo o processamento é realizado no lado cliente;

c) Linguagem leve e rápida;
d) Está em constante evolução e atualização;
e) Possui uma gama de ferramentas para auxílio no desenvolvimento de sistemas;
f) Pode ser executada em qualquer *browser*, inclusive *offline*.

### 3.3.1 CodeMirror

CodeMirror é um editor de texto versátil implementado em JavaScript para os navegadores. Quando um determinado tipo de código é inserido no editor, o texto terá sua sintaxe realçada, de acordo com a linguagem definida para o editor, facilitando assim a identificação dos diferentes elementos presentes no mesmo e poderá contar ainda com numeração de linhas. Atualmente o CodeMirror é um projeto open-source compartilhada pela licença MIT, seu código é mantido em um repositório na plataforma GitHub. O CodeMirror fornece uma rica API de programação e um sistema de temas em CSS para personalizar o layout e se ajustar ao seu aplicativo.

### 3.3.2 Bootstrap

Bootstrap é um framework JavaScript de código aberto que destina-se a facilitar a criação de sites e aplicações web . É uma combinação de HTML, CSS e código JavaScript projetado para ajudar usuários a construir os componentes da interface como formulários, botões, navegação e outros componentes da interface.

### 3.3.3 JavaScript

JavaScript é uma linguagem de script orientada a objetos, multiplataforma, pequena e “leve”. Dentro de um ambiente de host (por exemplo, um navegador web) o JavaScript pode ser ligado aos objetos deste ambiente para prover um controle programático sobre eles.

JavaScript tem uma biblioteca padrão de objetos, como: Array, Date, e Math, e um conjuntos de elementos que formam o núcleo da linguagem, tais como: operadores, estruturas de controle e declarações. O núcleo do JavaScript pode ser estendido para uma variedade de propósitos, complementando assim a linguagem:

O lado cliente do JavaScript estende-se do núcleo da linguagem, fornecendo objetos para controlar um navegador web e seu Document Object Model (DOM). Por exemplo, as extensões do lado do cliente permitem que uma aplicação coloque elementos em um formulário HTML e responda a eventos do usuário, como cliques do mouse, entrada de formulário e de navegação da página. O lado do servidor do JavaScript estende-se do núcleo da linguagem, fornecendo objetos relevantes à execução do JavaScript em um servidor. Por exemplo, as extensões do lado do servidor permitem que uma aplicação se comunique com um banco de dados, garantindo a continuidade de informações de uma chamada para a outra da aplicação, ou execute manipulações de arquivos em um servidor.

CAPÍTULO 4

# CONCLUSÃO

O objetivo geral deste trabalho foi o desenvolvimento de um software como instrumento de apoio ao ensino de algoritmos e lógica de programação por meio de português estruturado. Como resultado, obteve-se o software denominado PortugolJS, que compreende um ambiente de desenvolvimento integrado, constituído de editor de código, compilador, interpretador e depurador. O dialeto de Portugol reconhecível pelo compilador foi especificado em notação EBNF e contempla as principais estruturas de controle de fluxo do programa.

A utilização do compilador e interpretador Pascal S como base para este projeto mostrou-se uma escolha adequada, especialmente pela clareza e didática do seu código, o que facilitou em muito a compreensão dos mecanismos de compilação e interpretação. Como consequência, tanto a conversão do seu código, de Pascal para Javascript, como a incorporação de novos recursos tornaram-se tarefas, se não mais fáceis, menos desafiadoras.

Dentre os novos recursos de programação, o suporte ao tipo de dados ponteiro passa a permitir a alocação de memória em tempo de execução, assim como a definição de estruturas de dados dinâmicas mais complexas como listas encadeadas e árvores binárias.

A avaliação de cinco ferramentas similares ao PortugolJS mostrou que apenas duas delas (40%) permitem a passagem de parâmetros por referência, que é um conceito importante a ser trabalhado dentro do tópico “modularização”, geralmente presente nas ementas das disciplinas introdutórias de algoritmos e programação. Embora, comumente, a recursividade não seja ensinada nestas disciplinas, é desejável que ela esteja disponível e possa ser explorada por aqueles alunos que costumam ser mais curiosos ou estão acima da média. O suporte à subprogramas recursivos está presente apenas no Portugol Studio, a mais completa ferramenta similar avaliada. Portanto, por permitir tanto a passagem de parâmetros por referência como a recursividade, o PortugolJS surge com este diferencial em relação à maioria

das ferramentas atualmente disponíveis.

Acredita-se que outro ponto de destaque do PortugolJS é o seu mecanismo de depuração, que permite executar o programa passo a passo, inspecionar e alterar os valores das variáveis do escopo, bem como observar a pilha de chamadas a subprogramas. A depuração conta ainda com um recurso inédito, inexistente mesmo em IDEs comerciais, qual seja, a indicação do número vezes que cada linha de código foi executada, o que pode ser bastante útil na avaliação rápida da eficiência de processos recursivos ou dentro de laços controlados por uma expressão lógica.

Finalmente, conclui-se que este trabalho mostrou ser factível a implementação de uma ferramenta simples, que requer apenas um browser atualizado, mas que conta com recursos avançados como a capacidade de depuração de algoritmos.

Como sugestões para uma eventual continuidade deste trabalho, visando melhorias na interface de utilização e agregação de novas funcionalidades, têm-se:
a) Implementação de tipos de dados enumerados;
b) Implementação do tipo de dados conjunto;
c) Implementação de registros variantes;
d) Suporte a passagem de funções como parâmetros;
e) Suporte a geração de arquivo de instruções;
f) Criação de diagramas de execução do código;
g) Definição de palavras reservadas em outros idiomas;
h) Melhorias na interface gráfica do ambiente, incluindo a personalização;
i) Incorporar pontos de paradas como mecanismo de depuração;
j) Incorporar uma mini biblioteca de algoritmos exemplo.

CAPÍTULO 5

# REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V.; SETHI, R.; LAM, M. S. Compiladores: Principios, Técnicas e Ferramentas. Longman do Brasil. 2008.

COOPER, K.; TORCZON, L. Engineering a Compiler. Elsevier. 2011.

GRUNE, Dick and VAN REEUWIJK, K. BAL, H.; JACOBS, C. JH; LANGENDOEN, K. Modern Compiler Design. Springer Science & Business Media. 2012.

LOUDEN, K. C. Compiladores: Princípios e práticas. Pioneira Thomson Learning. 2004

MEDEIROS, A. V. M. Um interpretador online para a linguagem Portugol. 2015. 117 p. Trabalho de Conclusão de Curso (Graduação) Curso de Ciência da Computação, Universidade Federal de Sergipe, São Cristóvão - SE, 2015.

NETO, J. J. Contribuições à metodologia de construção de compiladores. São Paulo: Tese de Livre Docência, USP. 1993.

NOSCHANG, L. F.;PELZ, F.; JESUS, E. A.; RAABE, A. L. A.Portugol Studio: Uma IDE para Iniciantes em Programação.XXXIV Congresso da Sociedade Brasileira de Computação. 2014.

O'BRIEN, s. Turbo Pascal 6 : Completo e Total. Makron Books. São Paulo. 1992.

PRICE, A. M. A.; TOSCANI, S. S. Implementação de linguagens de programação: compiladores. Sagra-Luzzatto. 2000.

SEBESTA, R. W. Conceitos de Linguagens de Programação. Porto Alegre, Bookman, 5 Edição. 2003.

TOMAZELLI, G, C. Implementação de um Compilador para uma Linguagem de Programação com Geração de Código Microsoft .NET Intermediate Language. Trabalho de Conclusão de Curso (Graduação) - Curso de Ciência da Computação, Universidade Regional de Blumenau, Blumenau, 2004.

TRINDADE, M. H. Modelagem de Dados com Data Warehouse e OLAP: Um estudo de caso.2014. 45p. Trabalho de Conclusão de Curso (Graduação) - Curso de Sistemas de Informação, Universidade Federal da Grande Dourados, Dourados-MS. 2014.

VELOSO, P. A. S. Estruturação e verificação de programas com tipos de dados. São Paulo. Edgar Blucher, 1987

VIEIRA, C. E. C; RIBEIRO, C. C.; SOUZA, R. C. Geradores de Números Aleatórios. Rio de Janeiro. 2004

WIRTH, N. Compiler Construction. Addison-Wesley Reading. 1996.

WIRTH, N. Pascal-S: A Subset and its Implementation. Zurich. 1975.

.