UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Antônio Vinícius Menezes Medeiros
Orientador: Prof. Dr. Michel dos Santos Soares

UM INTERPRETADOR ONLINE PARA A LINGUAGEM PORTUGOL

São Cristóvão
2015

UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Antônio Vinícius Menezes Medeiros

UM INTERPRETADOR ONLINE PARA A LINGUAGEM PORTUGOL

Trabalho de Conclusão de Curso submetido ao Departamento de Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Michel dos Santos Soares

São Cristóvão

2015

Medeiros, Antônio Vinícius Menezes
Um interpretador online para a linguagem Portugol / Antônio Vinícius Menezes Medeiros – São Cristovão: UFS, 2015.

117 p.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal de Sergipe, Curso de Ciência da Computação, 2015.

1. Algoritmos. 2. Linguagens de Programação - TCC. 3. Ciência da Computação. I. Título.

Antônio Vinícius Menezes Medeiros

UM INTERPRETADOR ONLINE PARA A LINGUAGEM PORTUGOL

Trabalho de Conclusão de Curso submetido ao corpo docente do Departamento de Computação da Universidade Federal de Sergipe (DCOMP/UFS) como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

São Cristóvão, 27 de março de 2015.

BANCA EXAMINADORA:

_____

Prof. Michel dos Santos Soares, Doutor
Orientador
DCOMP/UFS

_____

Prof. Breno Piva Ribeiro, Mestre
DCOMP/UFS

_____

Prof. Rogério Patrício Chagas do Nascimento, Doutor
DCOMP/UFS

Esta página foi intencionalmente deixada em branco.

Dedico esse trabalho a todas as pessoas que contribuíram para minha formação e minha carreira na área de Informática, me guiando, me inspirando ou me ajudando, mas em especial, dedico esse trabalho:

à minha mãe, pelo apoio incondicional em todos os momentos da minha vida;

às minhas tias Deise e Denise, por terem me apresentado o computador, sem elas nada disso teria sequer começado;

à minha tia Inês, por ter sido minha primeira guru e por ter me presenteado com meu primeiro computador;

ao meu amigo Heron, por ser minha referência como técnico profissional, competente, sério, honesto, dedicado e autodidata;

ao meu professor Marcus Aurelius, por ter me feito descobrir o gosto pela programação e por ter me ensinado a programar com qualidade e bom senso;

aos meus orientadores Heli, Marcus Aurelius, Rogério, Alberto e Michel, por terem me guiado em momentos importantes da minha vida acadêmica;

ao meu professor Kalil, cujas aulas de Compiladores me motivaram a fazer este trabalho;

aos meus colegas de classe e amigos *nerds* do IFS e da UFS, pela palavra amiga, pela força nos momentos difíceis e pela alegria da sua companhia nos momentos de lazer;

aos meus colegas de trabalho e amigos do Cinform, da Moobi Tech e do Ministério Público de Sergipe, pela troca de experiências e pelos ambientes de trabalho prazerosos; e

à minha amada Gicélia, por acreditar, apostar e investir em mim, sempre me instigando a melhorar como aluno, como profissional e como pessoa.

Dedico este trabalho também a todos aqueles que dedicaram seu tempo a lê-lo.

Esta página foi intencionalmente deixada em branco.

AGRADECIMENTOS

Quero agradecer a Deus, que é um Pai verdadeiro, que nada deixa faltar, que provê tudo a seu tempo e sua hora, que protege dos perigos e escuta nos momentos de dificuldade, é graças a ele que nada tenho a reclamar da vida;

ao professor Michel, que me permitiu desenvolver como Trabalho de Conclusão de Curso exatamente o que eu desejava, que aceitou me acompanhar nessa aventura – ou loucura, como muitos preferiram classificar – de desenvolver um interpretador como Trabalho de Conclusão de Curso e me guiou com sua paciência, sabedoria e experiência, garantindo, assim, que esse trabalho atingisse seus objetivos;

aos professores Rogério e Breno, pelo pronto aceite à participação na Banca;

aos demais professores da UFS, pela amizade e companheirismo durante esta jornada;

a meus familiares, em especial minha mãe Silvia e meu irmão Ítalo, pelo amor, compreensão e apoio incondicionais;

à minha namorada Gicélia por todo carinho, apoio e compreensão necessários para a realização deste trabalho; e

a todos aqueles que dedicaram um momento por menor que fosse das suas vidas corridas para contribuir com meu trabalho visitando o *site*, testando o interpretador e fornecendo suas opiniões.

Esta página foi intencionalmente deixada em branco.

“Missão dada, é missão cumprida.”

(Coronel do BOPE, no filme Tropa de Elite)

Esta página foi intencionalmente deixada em branco.

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, 2015.

RESUMO

O processo de ensino-aprendizagem de programação é marcado por dificuldades e desafios, em especial devido à carência, por parte dos alunos, de habilidades cognitivas como a abstração e o raciocínio lógico-matemático e devido à dificuldade, por parte dos professores, de elaborar e executar estratégias efetivas para contornar esses problemas. Além disso, o uso de linguagens de programação e ambientes de desenvolvimento profissionais exige dos alunos um esforço maior do que o necessário para aprender a lógica fundamental à programação. A linguagem Portugol, próxima da língua portuguesa, foi concebida com o objetivo de facilitar o ensino de programação a estudantes brasileiros. Este trabalho objetiva analisar e formalizar a linguagem Portugol e desenvolver uma ferramenta para suportá-la, um ambiente de desenvolvimento simples com interpretador integrado que dispensa instalação, capaz de ser executado pela Internet. O presente texto apresenta o referencial teórico necessário e a metodologia adotada para o desenvolvimento da ferramenta, a especificação da linguagem Portugol, as ferramentas utilizadas na implantação da aplicação e os resultados deste trabalho.

Palavras-chave: algoritmos, programação, interpretador, Portugol.

Esta página foi intencionalmente deixada em branco.

ABSTRACT

The programming teaching-learning process is marked by difficulties and challenges, especially due to the deficit, on students' part, of cognitive skills such as abstraction and logical and mathematical reasoning and due to the difficulty, on teachers' part, to formulate and implement effective strategies to circumvent those problems. In addition, the use of professional programming languages and development environments requires students more efforts than necessary to learn programming logic. The Portugol language, close to the Portuguese language, was conceived aiming to facilitate teaching Brazilian students to program. This work aims to analyze and formalize the Portugol language and develop a tool to support it, a simple development environment with an integrated interpreter that requires no installation and can be run from the Internet. This text presents the necessary theoretical framework and the methodology adopted for developing the tool, the Portugol language specification, the tools used to implement the application and the results of that work.

Keywords: algorithms, programming, interpreter, Portugol.

Esta página foi intencionalmente deixada em branco.

LISTA DE ILUSTRAÇÕES

Esta página foi intencionalmente deixada em branco.

LISTA DE QUADROS

LISTA DE TABELAS

Esta página foi intencionalmente deixada em branco.

SUMÁRIO

1 INTRODUÇÃO

> Desde o início de sua existência, o homem procurou criar máquinas que o auxiliassem em seu trabalho, diminuindo esforço e economizando tempo. Dentre essas máquinas, o computador vem se mostrando uma das mais versáteis, rápidas e seguras. (ASCENCIO; CAMPOS, 2007)

Segundo o dicionário Michaelis (2014), **computar** significa calcular, avaliar, contar. Os computadores foram criados, a princípio, para facilitar operações matemáticas, fornecendo seus resultados com rapidez e precisão. Um dos primeiros computadores, o ENIAC (*Electronic Numerical Integrator And Computer*), que data de 1945, foi desenvolvido com o objetivo de acelerar cálculos balísticos para o Exército americano (MONTEIRO, 2007).

Desde então, os computadores não pararam de evoluir tecnologicamente. Hoje, eles não apenas realizam cálculos, mas são capazes de fazer quase tudo que se possa imaginar: textos, imagens, vídeos, músicas, jogos, aplicações multimídia, pesquisas, comunicação instantânea com qualquer lugar do mundo a qualquer dia e qualquer hora.

O que conferiu ao computador tamanha versatilidade foi um componente introduzido pela arquitetura proposta por John von Neumann para aperfeiçoar o ENIAC: o **programa armazenado** em memória. O computador trabalha conforme as instruções contidas no programa (MONTEIRO, 2007). Quando um usuário de computador deseja que ele realize uma computação, deve, portanto, fornecer-lhe um programa.

Um **programa** contém uma sequência de instruções ordenadas de maneira lógica que o computador deve executar para cumprir uma tarefa específica. Elaborar um programa consiste em representar a solução de uma tarefa por meio de operações que a máquina possa realizar. O raciocínio embutido na sequência de instruções é conhecido por **algoritmo** (ASCENCIO; CAMPOS, 2007; BINI; KOSCIANSKI, 2009; MONTEIRO, 2007).

Um exemplo de algoritmo para somar dois números é apresentado no Quadro 1.

Quadro 1 - Exemplo de algoritmo para somar dois números escrito em português

```
1 Início
2 Obter do usuário, através do teclado numérico, o valor do primeiro operando
3 Obter do usuário, através do teclado numérico, o valor do segundo operando
4 Somar os dois operandos
5 Informar ao usuário, através da tela, o resultado da operação
6 Fim
```

(autoria própria)

Qualquer ser humano que compreenda a língua portuguesa é capaz de compreender

esse algoritmo. No entanto, para que o computador o compreenda e execute, o usuário deve escrevê-lo em uma linguagem que tanto o computador quanto o usuário entendam. Essa linguagem é chamada de **linguagem de programação** (ASCENCIO; CAMPOS, 2007).

O algoritmo apresentado em português no Quadro 1 pode ser escrito usando, por exemplo, a linguagem de programação Pascal como mostra o Quadro 2.

Quadro 2 - Exemplo de algoritmo para somar dois números escrito em Pascal

```
1 program soma;
2 var x, y, z: integer;
3 begin
4      read(x, y);
5      z := x + y;
6      write(z);
7 end.
```

(autoria própria)

A linguagem Pascal foi desenvolvida no final da década de 1960 por Niklaus Wirth, na Suíça, como uma ferramenta para o ensino de programação estruturada. Por isso, ela foi muito utilizada nos cursos técnicos de nível médio e nos cursos superiores de áreas tecnológicas para ensinar programação (ASCENCIO; CAMPOS, 2007; BINI; KOSCIANSKI, 2009).

O processo de ensino-aprendizagem de programação corresponde a parte considerável da grade desses cursos, especialmente nos primeiros semestres, e normalmente é considerado desafiador para alunos e professores por diversos motivos (BINI; KOSCIANSKI, 2009; MOREIRA; FAVERO, 2009; RAABE; SILVA, 2005; ROCHA et al, 2010; SOUZA, 2009):

a) carência de habilidades cognitivas necessárias à disciplina, como a capacidade de abstração, o raciocínio lógico-matemático e o desenvolvimento de estratégias para solucionar problemas;
b) hábitos de estudo pouco disciplinados e focados em memorização;
c) diferentes experiências prévias de manuseio de tecnologias e de programação;
d) utilização de materiais, ferramentas e linguagens não adaptadas pedagogicamente;
e) diferentes ritmos de aprendizagem;
f) dificuldade em entender os conteúdos apresentados;
g) dificuldade em estabelecer relações com conteúdos já apreendidos, visto que a lógica algorítmica é abstrata e totalmente nova para a maioria dos alunos;
h) dificuldade em entender os enunciados dos problemas propostos e solucioná-los, ou até mesmo compreender as soluções propostas pelos colegas ou pelo professor;

i) abordagens pouco motivadoras; e
j) dificuldade, por parte dos professores, de diagnosticar e sanar as dificuldades de aprendizagem de cada aluno, dada a grande quantidade de alunos por turma.

Essas dificuldades podem ocasionar desinteresse e aversão às disciplinas de programação e acarretam uma quantidade expressiva de reprovações e evasões ao final de cada semestre (RAABE; SILVA, 2005).

Ainda que passe nas disciplinas iniciais, se o aluno não aproveitá-las adequadamente, pode enfrentar dificuldade em disciplinas posteriores, que dependem dos conceitos básicos de programação. Há muitos alunos que passam pelo curso sem adquirir um conhecimento mínimo adequado de programação (MOREIRA; FAVERO, 2009; ROCHA et al, 2010).

Uma abordagem tradicional para ensinar programação consiste em usar apenas lápis e papel para elaborar algoritmos. Para a maioria dos alunos, essa abordagem é muito abstrata, pois não conseguem verificar imediatamente a corretude das suas soluções executando-as no computador. Assim, se torna difícil associar o algoritmo à maneira como ele é executado pelo computador, assim como perceber como sua execução é afetada pela utilização de estruturas de controle e de repetição (BINI; KOSCIANSKI, 2009; SOUZA, 2009).

Diante do exposto, conclui-se que as disciplinas de programação exigem um processo de ensino-aprendizagem diferenciado daquele que utiliza apenas recursos estáticos como transparências, textos ditados, diagramas, entre outros. A habilidade de programar computadores não pode ser adquirida sem grande esforço em atividades práticas de laboratório (MOREIRA; FAVERO, 2009; SOUZA, 2009).

Muitas ferramentas têm sido propostas para tentar eliminar, ou ao menos minimizar, as dificuldades apresentadas e vêm demonstrando resultados positivos (BINI; KOSCIANSKI, 2009; RAABE; GIRAFFA, 2006; RAABE; SILVA, 2005; ROCHA et al, 2010; SANTIAGO; PEIXOTO; SILVA, 2011; SOUZA, 2009; VIEIRA; RAABE; ZEFERINO, 2010).

A atividade de programação tradicionalmente requer o uso de *softwares* de apoio, como, por exemplo, editores, compiladores, interpretadores e depuradores. Os **ambientes integrados de desenvolvimento** (do inglês *integrated development environments*, IDEs) reúnem esses recursos de forma a facilitar as atividades do programador. Esses ambientes são vantajosos especialmente para os programadores iniciantes, pois minimizam a ocorrência de erros comuns, como erros de sintaxe ou mesmo de lógica (RUSSI; CHARÃO, 2011).

O uso de ambientes integrados de desenvolvimento também é bem-aceito por professores. Existem ambientes concebidos especificamente para o ensino de linguagens de programação, como, por exemplo, o BlueJ, o Greenfoot e o DrJava, destinados ao ensino-

aprendizagem da linguagem Java. Ambientes de desenvolvimento profissionais também podem ser utilizados para fins educacionais, embora não sejam recomendados para alunos novatos em programação (NOSCHANG et al, 2014; RUSSI; CHARÃO, 2011).

Além disso, os formalismos das linguagens de programação profissionais representam uma preocupação adicional para os alunos novatos em programação, que podem desviar sua atenção do principal objetivo das disciplinas de programação, que é desenvolver soluções para os problemas propostos em sala (BINI; KOSCIANSKI, 2009; SOUZA, 2009).

Ascencio e Campos (2007) propõem uma linguagem acadêmica para a escrita de algoritmos que elas chamam de **pseudocódigo** ou **Portugol**. Assim como a linguagem Pascal é próxima da língua inglesa, a linguagem Portugol é próxima da língua portuguesa, o que a torna vantajosa no ensino de programação a estudantes brasileiros.

O algoritmo para somar dois números, apresentado em português no Quadro 1, pode ser escrito usando a linguagem Portugol como mostra o Quadro 3.

Quadro 3 - Exemplo de código para somar dois números escrito em Portugol

```
1 ALGORITMO
2 DECLARE x, y, z NUMERICO
3 LEIA x, y
4 z <- x + y
5 ESCREVA z
6 FIM_ALGORITMO.
```

(autoria própria)

Os conceitos básicos de programação, apreendidos mais facilmente com o uso de uma linguagem como o Portugol, podem ser transpostos posteriormente para linguagens profissionais (SOUZA, 2009). Um estudante de programação que conheça a linguagem Pascal, por exemplo, ao ler e compreender o algoritmo em Portugol do Quadro 3, pode produzir seu equivalente em Pascal, mostrado no Quadro 2.

O propósito deste trabalho é analisar a linguagem Portugol, como proposta por Ascencio e Campos (2007), e tornar possível a execução de programas escritos em Portugol pelo computador, através do projeto de um ambiente de desenvolvimento voltado para o meio acadêmico, em especial para os alunos das disciplinas de programação e seus professores.

Vale observar que a linguagem Portugol já é utilizada, com variações, nos cursos brasileiros de computação (SOUZA, 2009). Zanini e Raabe (2012 apud NOSCHANG et al, 2014) apontam que a maioria dos livros sobre introdução à programação adotados pelas universidades brasileiras utilizam alguma variação de Portugol como recurso didático.

2 METODOLOGIA

Segundo Shaw (2002), as pesquisas científicas podem ser caracterizadas: (i) pelos tipos de perguntas que elas fazem; (ii) pelos tipos de resultados que elas produzem; e (iii) pelos tipos de validação que elas utilizam para verificar seus resultados. Geralmente, as pesquisas em engenharia de *software* buscam melhores maneiras de desenvolver e avaliar *softwares*. Elas são motivadas por problemas práticos e comumente objetivam pesquisar a qualidade, o custo e a atualidade de produtos de *software*.

Este trabalho busca uma melhor maneira de ensinar programação aos iniciantes. Como resultado, propõe uma linguagem de programação mais próxima do convívio dos estudantes (mais próxima do português, ao invés do inglês) e uma ferramenta simples, de fácil utilização, mas eficiente, que possa auxiliá-los no aprendizado da linguagem.

A validação de trabalhos semelhantes, nos quais se propõe a utilização de uma ferramenta para auxiliar o ensino-aprendizagem de algoritmos (HOSTINS; RAABE, 2007; KOLIVER, DORNELES, CASA, 2004; RAABE, SILVA, 2005; VIEIRA; RAABE; ZEFERINO, 2010), consiste em utilizar a ferramenta proposta em uma, duas ou até três turmas de uma disciplina sobre introdução à programação, no mesmo semestre ou em semestres diferentes, e avaliar o desempenho dos alunos durante ou após o uso da ferramenta. Há diversas ameaças à validade dos experimentos que devem ser controladas, como relatam Vieira, Raabe e Zeferino (2010).

Como o desenvolvimento da ferramenta demandou muito tempo, como é mostrado no cronograma adotado (Tabela 1), verificou-se que seria inviável realizar um experimento que demonstrasse a validade da ferramenta proposta por este trabalho. Então, optou-se por procedimentos mais simples para verificar seu funcionamento: realizar testes utilizando exemplos do livro durante todo o desenvolvimento e, após a implantação da aplicação na Internet, divulgá-la e colher opiniões dos usuários acerca de suas experiências utilizando-a.

O desenvolvimento da ferramenta ocorreu de forma iterativa e incremental, como prevê a metodologia ágil SCRUM (SOARES, 2004), com atualizações diárias do código-fonte, trocas de e-mails para reportar o andamento do trabalho e ciclos de desenvolvimento de duas semanas, que envolveram pesquisa, documentação, implementação e testes para cada uma das atividades planejadas. Obedeceu-se rigorosamente o cronograma da Erro: Origem da referência não encontrada.

Os requisitos da aplicação foram definidos a partir da pesquisa e identificação das funcionalidades mais essenciais e comuns aos ambientes de desenvolvimento atuais (RUSSI;

CHARÃO, 2011; SOUZA, 2009; VIEIRA; RAABE; ZEFERINO, 2010). A análise dos requisitos culminou na seguinte lista:

a) Instalação: a instalação deve ser fácil e rápida, para que o usuário possa começar a utilizar a ferramenta o mais breve possível;
b) Interface: a ferramenta deve apresentar interface intuitiva, limpa e amigável;
c) Edição: a ferramenta deve possuir um editor de código-fonte com funcionalidades comuns, incluindo salvar, abrir, recortar, copiar, colar, desfazer, refazer, localizar, substituir;
d) Numeração de linhas: o editor de código-fonte deve numerar as linhas, visando facilitar a localização de eventuais erros reportados;
e) Realce de sintaxe (*syntax highlighting*): o editor de código-fonte deve exibir os elementos da linguagem de programação (palavras reservadas, variáveis, valores, cadeias de caracteres) com formatações diferentes, facilitando a visualização e localização desses elementos no código;
f) Compleção de código: a ferramenta deve auxiliar o programador na digitação, reduzindo o esforço para digitar todo o código-fonte e a probabilidade de erros;

Tabela 1 - Cronograma

Tarefa / ciclo	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Documentação													
Introdução	X	X	X										
Referencial teórico	X	X	X	X	X	X	X	X	X	X			
Especificação da linguagem	X	X	X	X	X	X	X	X	X	X			
Conclusão e revisão											X	X	
Implementação													
Analisador léxico	X	X											
Analisador sintático			X	X									
Analisador semântico				X	X	X							
Executor						X	X	X					
Interface gráfica									X	X			
Implantação na Internet									X	X			
Testes	X	X	X	X	X	X	X	X	X	X	X	X	
Apresentação													X

(autoria própria)

g) Endentação: a ferramenta deve permitir a fácil endentação do código-fonte, uma prática comum para organizá-lo e facilitar sua leitura;
h) Indicação de erros: a ferramenta deve indicar ao programador quaisquer erros encontrados no código-fonte;
i) Execução: a ferramenta deve possibilitar a execução dos códigos-fonte simulando o console do computador, pelo qual o usuário vai interagir com o programa por ele fornecido;
j) Plataformas: a ferramenta deve funcionar nas plataformas mais utilizadas (Windows, Linux e Mac OS X); e
k) Licença de uso: a ferramenta deve ser distribuída sob uma licença de *software* livre, de modo que seja mais acessível.

O referencial teórico foi constituído por livros e artigos consolidados pela academia sobre os temas necessários ao desenvolvimento da aplicação: linguagens de programação e tradutores.

Esta página foi intencionalmente deixada em branco.

3 REFERENCIAL TEÓRICO

Um algoritmo descrito informalmente em português, como o exemplo do Quadro 1, não pode ser processado por uma máquina. A tarefa de programar um computador requer o uso de uma linguagem intermediária que possibilite a comunicação entre programador e computador, chamada linguagem de programação (ASCENCIO; CAMPOS, 2007).

3.1 LINGUAGENS DE PROGRAMAÇÃO

As instruções reconhecidas por um computador representam basicamente operações matemáticas e lógicas, além de operações de busca, leitura e gravação de dados na memória. Juntas, elas formam a linguagem do computador, denominada **linguagem de máquina**. Uma instrução definida por essa linguagem é denominada **instrução de máquina**.

Cada instrução de máquina é pois uma sequência de zeros e uns que indica ao processador a operação a ser realizada e os dados que serão utilizados nessa operação. Uma instrução enviada ao computador para somar dois números, por exemplo, deverá conter a sequência de bits que identifica a operação de soma e outra sequência de bits que indica os valores que devem ser somados, ou a posição deles na memória, se eles estiverem armazenados na memória (MONTEIRO, 2007).

O algoritmo para somar dois números, apresentado em português no Quadro 1, poderia ser escrito em linguagem de máquina de maneira análoga à apresentada no Quadro 4 (os textos em português descrevem o significado das instruções hipotéticas).

Quadro 4 - Exemplo de algoritmo para somar dois números em linguagem de máquina

```
1 0000 0010 1100          Início do programa
2 0010 0111 1100          Obter do usuário o valor do primeiro operando
3 0010 1000 1101          Obter do usuário o valor do segundo operando
4 0100 0111 1100          Armazenar na memória o primeiro operando
5 0110 1100 1101          Somar com o segundo operando
6 0101 1100 1101          Gravar o resultado na memória
7 0011 1110 1100          Informar ao usuário o resultado da operação
8 0001 0101 1000          Fim do programa
```

(autoria própria)

Com pouco tempo do surgimento do computador, criar programas em linguagem de máquina se mostrou uma tarefa muito difícil, demorada, tediosa e suscetível a erros (LOUDEN, 1997; MONTEIRO, 2007).

Para tornar os programas mais inteligíveis ao programador, foi desenvolvida, ainda para os primeiros computadores, uma linguagem simbólica, que representa as instruções por símbolos alfanuméricos em vez de apenas números, a **linguagem *assembly***. Programas escritos em *assembly* apresentam uma linha para cada instrução, tendo, portanto, uma relação de 1 para 1 com os programas escritos em linguagem de máquina. Ainda assim, são mais simples de compreender e manter. Isso só é possível, no entanto, se o programador conhecer os mnemônicos definidos pela linguagem (como ADD, por exemplo) e entender a arquitetura da máquina que pretende programar (LOUDEN, 1997; MAK, 2009; MONTEIRO, 2007).

O algoritmo para somar dois números, apresentado em português no Quadro 1, poderia ser escrito em linguagem *assembly* de maneira análoga à apresentada no Quadro 5 (novamente, os textos em português descrevem as instruções hipotéticas).

Quadro 5 - Exemplo de algoritmo para somar dois números em linguagem *assembly*

```
1 PROG    SOMA          Início do programa
2 READ    X             Obter do usuário o valor do primeiro operando
3 READ    Y             Obter do usuário o valor do segundo operando
4 LDA     X             Armazenar na memória o primeiro operando
5 ADD     Y             Somar com o segundo operando
6 STR     Z             Gravar o resultado na memória
7 WRITE   Z             Informar ao usuário o resultado da operação
8 END     SOMA          Fim do programa
```

(autoria própria)

Programas escritos em linguagem *assembly* não podem ser executados pelo computador, porque ele não entende símbolos, apenas instruções de máquina. Para executar um programa escrito em linguagem *assembly*, é necessário utilizar um **montador** (*assembler*), programa que traduz as instruções em linguagem *assembly* em correspondentes instruções de máquina, que podem ser executadas pelo computador (LOUDEN, 1997; MONTEIRO, 2007).

Um passo mais significativo no sentido de melhorar a linguagem de comunicação com o computador foi o desenvolvimento de linguagens que refletem os procedimentos utilizados na solução de problemas, com instruções mais simples e concisas, mais próximas do entendimento do ser humano, mas ainda passíveis de conversão para instruções de máquina. Foi então que surgiram as linguagens de programação (LOUDEN, 1997; MONTEIRO, 2007).

Uma linguagem de programação é uma notação para descrever computações para pessoas e máquinas (AHO et al, 2006). Existem milhares de linguagens de programação, seja para escrever programas em geral, seja para desenvolver programas que resolvem tipos mais

específicos de problemas. São exemplos de linguagens de programação: COBOL, FORTRAN, LISP, Pascal, C, BASIC e Java (MONTEIRO, 2007).

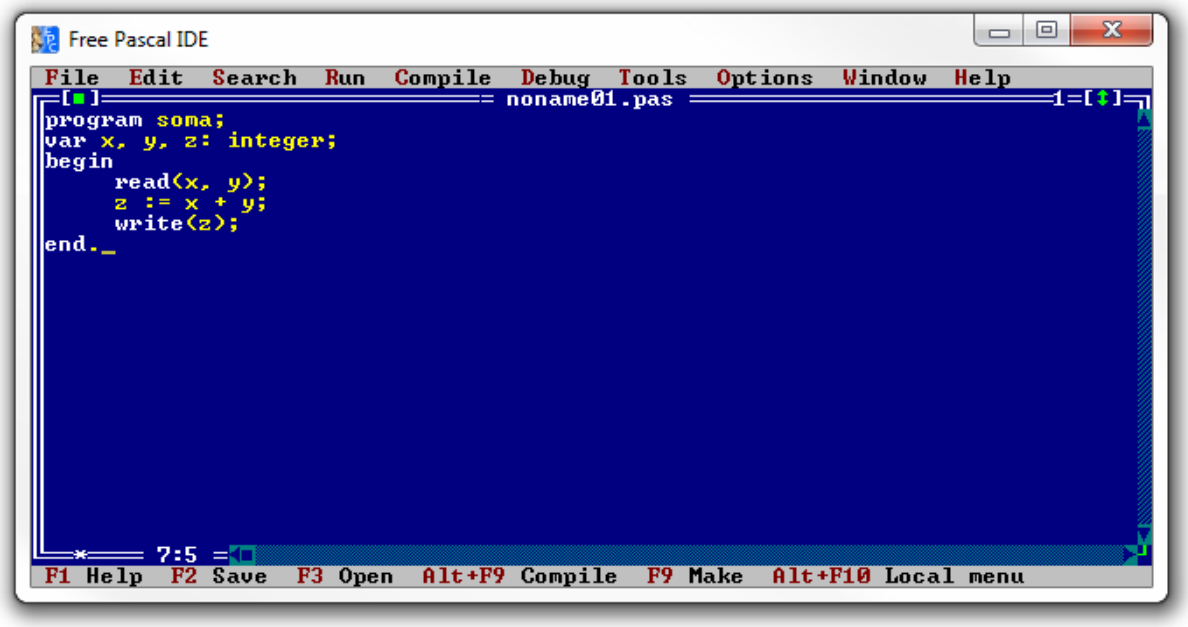
Uma instrução definida por uma linguagem de programação é comumente chamada de **comando**. Os programas são escritos em linguagens de programação, que são mais legíveis do ponto de vista do programador e apresentam características semelhantes às linguagens usadas por seres humanos, e depois são convertidos, por meio de um processo chamado **tradução**, para linguagem de máquina, para que possam ser executados pelo computador. Um programa completo, quando escrito em uma linguagem de programação, é chamado de **código-fonte**, ou simplesmente **código** (MONTEIRO, 2007).

O algoritmo para somar dois números, apresentado em português no Quadro 1, pode ser escrito usando, por exemplo, a linguagem Pascal, como mostra o Quadro 2.

As linguagens de programação facilitam a atividade do programador não apenas simplificando a codificação do algoritmo. Os projetistas de linguagens de programação fornecem junto com a sua especificação, dentre outros recursos, **tradutores**, programas que realizam a tradução do código em linguagem de programação para instruções em linguagem de máquina, e ambientes integrados de desenvolvimento, ferramentas que facilitam a codificação, tradução, execução e testes de programas.

A Figura 1 mostra o ambiente de desenvolvimento integrado Free Pascal, que é um *software* livre semelhante em diversos aspectos ao Turbo Pascal (proprietário) da Borland.

Figura 1 - Ambiente de desenvolvimento integrado Free Pascal



(autoria própria)

3.2 TRADUTORES

Um tradutor é um programa capaz de ler um programa em uma linguagem – a **linguagem fonte** – e traduzi-lo em um programa equivalente em outra linguagem – a **linguagem destino** ou **linguagem-objeto**. Eventualmente, durante o processo de tradução, o tradutor pode se deparar com erros no programa original e deve reportá-los (AHO et al, 2006; LOUDEN, 1997).

Há dois tipos principais de tradutores: compiladores e interpretadores (AHO et al, 2006).

Um **compilador** analisa um código escrito em uma linguagem-fonte, o código-fonte ou **programa-fonte**, e gera, a partir dele, um programa equivalente em uma linguagem-objeto, o **código-objeto** ou **programa-objeto**. O programa produzido é, então, chamado de **código executável** ou **programa executável**. Comumente, a linguagem-fonte é uma linguagem de programação e a linguagem-objeto é a linguagem binária de máquina, que o computador é capaz de executar diretamente. O código-fonte normalmente tem a forma de um arquivo de texto, e o código executável, de um arquivo binário (AHO et al, 2006; LOUDEN, 1997; MAK, 2009; MONTEIRO, 2007).

Uma vez produzido, o programa executável pode ser invocado pelo usuário e executado pelo computador, que obtém dados do usuário, processa esses dados conforme as instruções de máquina contidas no programa e retorna uma saída para o usuário, sem necessidade de auxílio do compilador (AHO et al, 2006; MONTEIRO, 2007).

Utilizando a compilação, para que um programa possa ser executado, é necessário que o código-fonte tenha sido traduzido por inteiro para código executável. A compilação não compreende a execução do programa, apenas a tradução (MONTEIRO, 2007). Há, então, dois momentos distintos, esquematizados na Figura 2: (i) o momento em que o programa executável é gerado a partir da compilação do código-fonte e (ii) o momento em que o programa executável é de fato executado.

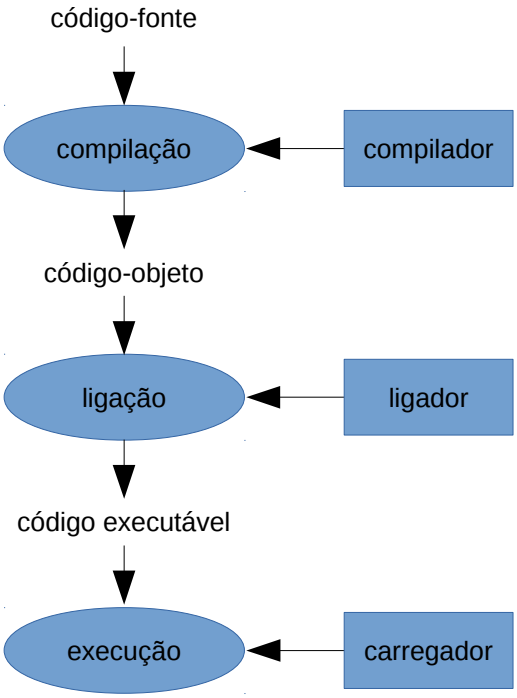
Na prática, além do compilador, outros programas podem ser necessários para criar um programa executável, como mostrado na Figura 3.

Se o código-fonte estiver dividido em vários arquivos, a tarefa de unir o conteúdo desses arquivos e passá-lo para o compilador pode ser atribuída a um programa a parte, chamado **preprocessador** (*preprocessor*). Esse programa é invocado pelo compilador antes de iniciar a tradução propriamente dita e também pode, além de unir arquivos de código-fonte, eliminar comentários, espaços em branco e substituir macros.

Figura 2 - Compilação e execução ocorrem em momentos distintos

(AHO et al, 2006)

Figura 3 - Programas adicionais necessários à compilação



(MONTEIRO, 2007)

Se o código-objeto gerado pelo compilador for, na verdade, um código em linguagem *assembly*, que é mais fácil de produzir e de traduzir para linguagem de máquina, esse código deverá ser processado por um montador para a obtenção do código executável.

Como grandes programas são compilados em partes separadas, pode ser necessário um **ligador** (*linker*) para resolver as referências de um código a outro e um **carregador** (*loader*) para carregar para a memória todas as peças de código executável antes de começar a execução (AHO et al, 2006; LOUDEN, 1997; MAK, 2009; MONTEIRO, 2007).

Um **interpretador**, em vez de produzir um programa executável como resultado da tradução (como faz o compilador), obtém do usuário o código-fonte do programa e os dados que lhe devem ser fornecidos, analisa, traduz e executa cada um dos comandos na sequência em que aparecem no código-fonte, e retorna uma saída para o usuário. O interpretador realiza, portanto, compilação, ligação e execução comando a comando do código-fonte (AHO et al, 2006; MONTEIRO, 2007). Pode-se dizer que um interpretador traduz um programa nas ações que resultam de sua execução (MAK, 2009).

A interpretação compreende a tradução e a execução do programa: cada comando do código-fonte é traduzido pelo interpretador e imediatamente executado, antes que o próximo comando seja considerado. A interpretação não gera um produto intermediário, como o código-objeto ou o código executável gerado pela compilação (MONTEIRO, 2007). Esse processo é esquematizado na Figura 4.

Figura 4 - A interpretação compreende a tradução e a execução do programa

(AHO et al, 2006)

Em princípio, qualquer linguagem de programação pode ser interpretada ou compilada. Porém, um ou outro método de tradução pode ser preferido a depender da linguagem utilizada e da situação em que é empregada (LOUDEN, 1997).

Tanto a compilação quanto a interpretação apresentam vantagens e desvantagens. O executável produzido pelo compilador apresenta um desempenho melhor que o interpretador traduzindo e executando o código-fonte, uma vez que o computador é melhor em executar um programa na sua linguagem nativa. No entanto, um interpretador pode fornecer um melhor diagnóstico sobre eventuais erros encontrados no código-fonte, uma vez que ele controla a execução do código-fonte comando a comando (AHO et al, 2006; MAK, 2009). Os interpretadores são muito utilizados no ensino de programação e no desenvolvimento de *software*, situações em que os programas são traduzidos repetidas vezes (LOUDEN, 1997).

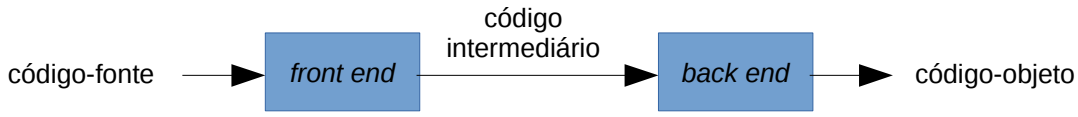
3.2.1 Processo de compilação

O processo de compilação (Figura 5) pode ser compreendido em duas etapas: (i) a **análise** ou **etapa inicial**, em que o compilador analisa os componentes do código-fonte e verifica se eles seguem as regras da linguagem, reportando ao programador quaisquer erros encontrados ou gerando uma **representação intermediária** do código-fonte; e (ii) a **síntese** ou **etapa final**, em que o código-objeto é gerado a partir da representação intermediária.

A etapa de análise depende apenas da linguagem-fonte, enquanto a etapa de síntese depende da linguagem-objeto (AHO et al, 2006; LOUDEN, 1997).

De maneira ideal, os compiladores podem ser decompostos em duas partes distintas, implementadas separadamente (Figura 5): o ***front end***, que realiza a etapa de análise do código-fonte, e o ***back end***, que realiza a etapa de síntese do programa executável (AHO et al, 2006; LOUDEN, 1997).

Figura 5 - O processo de compilação



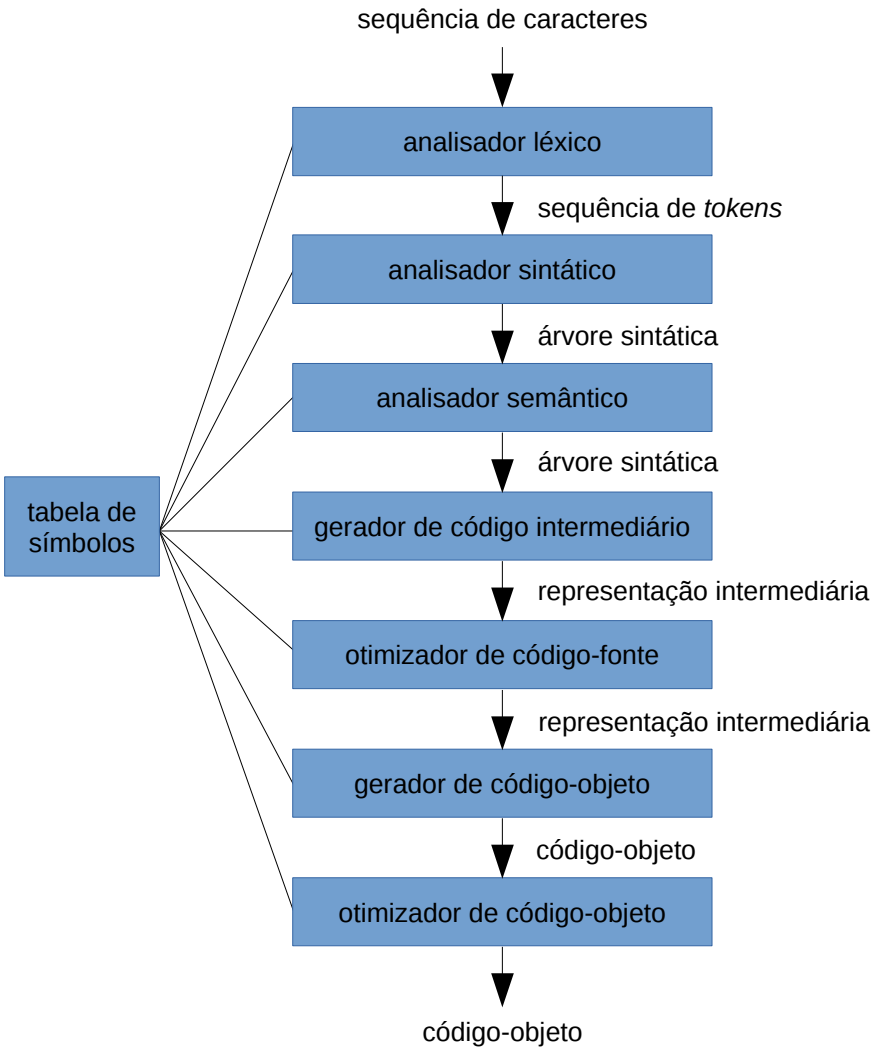
(LOUDEN, 1997)

Examinando o processo de compilação em mais detalhes, percebe-se que ele pode ser organizado logicamente como uma sequência de **fases**. Cada fase transforma uma representação do código-fonte em outra, como mostrado na Figura 6. As fases de análise léxica, análise sintática e análise semântica pertencem à etapa inicial de análise, enquanto a geração de código pertence à etapa final de síntese. Fases de otimização de código são comumente definidas, mas opcionais (AHO et al, 2006; LOUDEN, 1997).

Essas fases podem ser implementadas como peças separadas que compõem o compilador, como ilustrado na Figura 6, ou, como acontece mais comumente, podem ser agrupadas em **passadas** que processam um arquivo de entrada e produzem um arquivo de saída, dispensando a construção explícita das representações intermediárias entre as fases agrupadas (AHO et al, 2006; LOUDEN, 1997).

Vale observar que, na prática, os compiladores são muito diferentes em termos de organização. Assim, as fases descritas representam generalizações e estão presentes de alguma forma em quase todos os compiladores (LOUDEN, 1997).

Figura 6 - Componentes de um compilador



(AHO et al, 2006; LOUDEN, 1997)

3.2.2 Analisador léxico

O **analisador léxico** ou ***scanner*** realiza a leitura de fato do código-fonte, que é apresentado para ele como uma sequência, um fluxo (*stream*) de caracteres. Sobre essa sequência de caracteres, ele executa o que se conhece como **análise léxica** ou ***scanning***: o agrupamento de caracteres em unidades significativas elementares chamadas **lexemas**, os quais são **palavras** pertencentes à linguagem-fonte, análogas às palavras de uma linguagem natural como o inglês ou o português. Assim, a análise léxica é semelhante a uma análise morfológica ou ortográfica (AHO et al, 2006; LOUDEN, 1997).

Para cada lexema encontrado no código-fonte, o analisador léxico produz como saída um ***token***, que contém a classificação do lexema e outros atributos (como linha e coluna onde ele aparece no código-fonte) e é passado para o componente seguinte do processo de compilação, o analisador sintático (AHO et al, 2006).

Por exemplo, seja a linha a seguir, que poderia ser parte de um código-fonte em C.

Quadro 6 - Exemplo de linha de código escrito em C

```
a[posicao] = valorInicial + taxa * 60;
```

(autoria própria)

Esse código contém 38 caracteres, mas apenas 11 *tokens*: **a** (identificador), **[** (colchete esquerdo), **posicao** (identificador), **]** (colchete direito), **=** (operador de atribuição), **valorInicial** (identificador), **+** (operador de soma), **taxa** (identificador), ***** (operador de multiplicação), **60** (número inteiro) e **;** (ponto e vírgula).

As palavras pertencentes a uma linguagem-fonte são definidas usando um formalismo matemático chamado expressão regular (LOUDEN, 1997), que será apresentado a seguir.

De maneira análoga às expressões aritméticas, que descrevem operações matemáticas, as expressões regulares definem linguagens. O valor de uma expressão aritmética é um número, enquanto o valor de uma expressão regular é uma linguagem (SIPSER, 2005).

Um **alfabeto** Σ é um conjunto finito de símbolos (por exemplo, letras, dígitos e caracteres de pontuação). Uma palavra ou **cadeia** (*string*) s sobre um alfabeto Σ é uma sequência finita de símbolos pertencentes a esse alfabeto. O **tamanho** da cadeia s é o número de símbolos em s. A **cadeia vazia** (*empty string*) ε é um caso particular de cadeia que não contém caracteres, portanto, seu tamanho é igual a zero. Uma **linguagem** L sobre um alfabeto Σ é qualquer conjunto contável de cadeias formadas sobre esse alfabeto (AHO et al, 2006; GAGNON, 1998; LOUDEN, 1997; SIPSER, 2005).

A Tabela 2 resume as possíveis operações sobre linguagens.

Expressões regulares representam padrões de palavras e são definidas por (AHO et al, 2006; GAGNON, 1998; LOUDEN, 1997; SIPSER, 2005):

a) ε é uma expressão regular que denota a linguagem $L(\varepsilon)=\{\varepsilon\}$, um conjunto que contém apenas a cadeia vazia;

b) se c é um símbolo pertencente ao alfabeto Σ, então c é uma expressão regular que denota a linguagem $L(c)=\{c\}$, um conjunto que contém apenas a cadeia c (uma cadeia de tamanho um); e

c) se r e s são expressões regulares que denotam as linguagens $L(r)$ e $L(s)$, respectivamente, então:

- (r) é uma expressão regular que denota a linguagem $L(r)$;
- $(r)|(s)$ é uma expressão regular que denota a linguagem $L(r) \cup L(s)$;
- $(r)(s)$ é uma expressão regular que denota a linguagem $L(r)L(s)$; e
- $(r)*$ é uma expressão regular que denota a linguagem $(L(r))*$.

Tabela 2 - Operações sobre linguagens

Operação	Definição
União ($L \cup M$)	$L \cup M = \{s \mid s \in L$ ou $s \in M\}$
Concatenação (LM)	$LM = \{st \mid s \in L$ e $t \in M\}$
Fechamento de Kleene ($L*$)	$L* = \{x_1 x_2 \ldots x_k \mid k \geq 0$ e cada $x_i \in L\}$

(AHO et al, 2006; GAGNON, 1998; LOUDEN, 1997; SIPSER, 2005)

Na prática, algumas regras são adicionadas para simplificar a notação de expressões regulares. Ao operador $*$ é conferida a maior procedência, seguido do operador de concatenação e depois do operador de união. Essas regras de precedência visam reduzir a utilização de parênteses. Além disso, são definidos mais dois operadores: $+$ e $?$, onde $r+ = rr*$ e $r? = r|\varepsilon$. A esses operadores é concedida a mesma precedência do operador $*$ (AHO et al, 2006; GAGNON, 1998; LOUDEN, 1997; SIPSER, 2005).

São exemplos de expressões regulares (GAGNON, 1998; LOUDEN, 1997; SIPSER, 2005):

a) $a*$, que denota a linguagem $\{\varepsilon, a, aa, aaa, \ldots\}$;

b) $a+ \equiv aa* = \{a, aa, aaa, \ldots\}$;

c) $a?b \equiv ab|b = \{ab, b\}$;

d) $d = 0|1|2|3|4|5|6|7|8|9 = \{0,1,\ldots,9\}$ é a expressão regular que denota dígitos decimais; e

e) $n = (+|-|\varepsilon)d+$ é a expressão regular que denota números inteiros. Exemplos de cadeias denotadas por essa expressão regular são: **`-1`**, **`0`**, **`2`**, e **`+30`**.

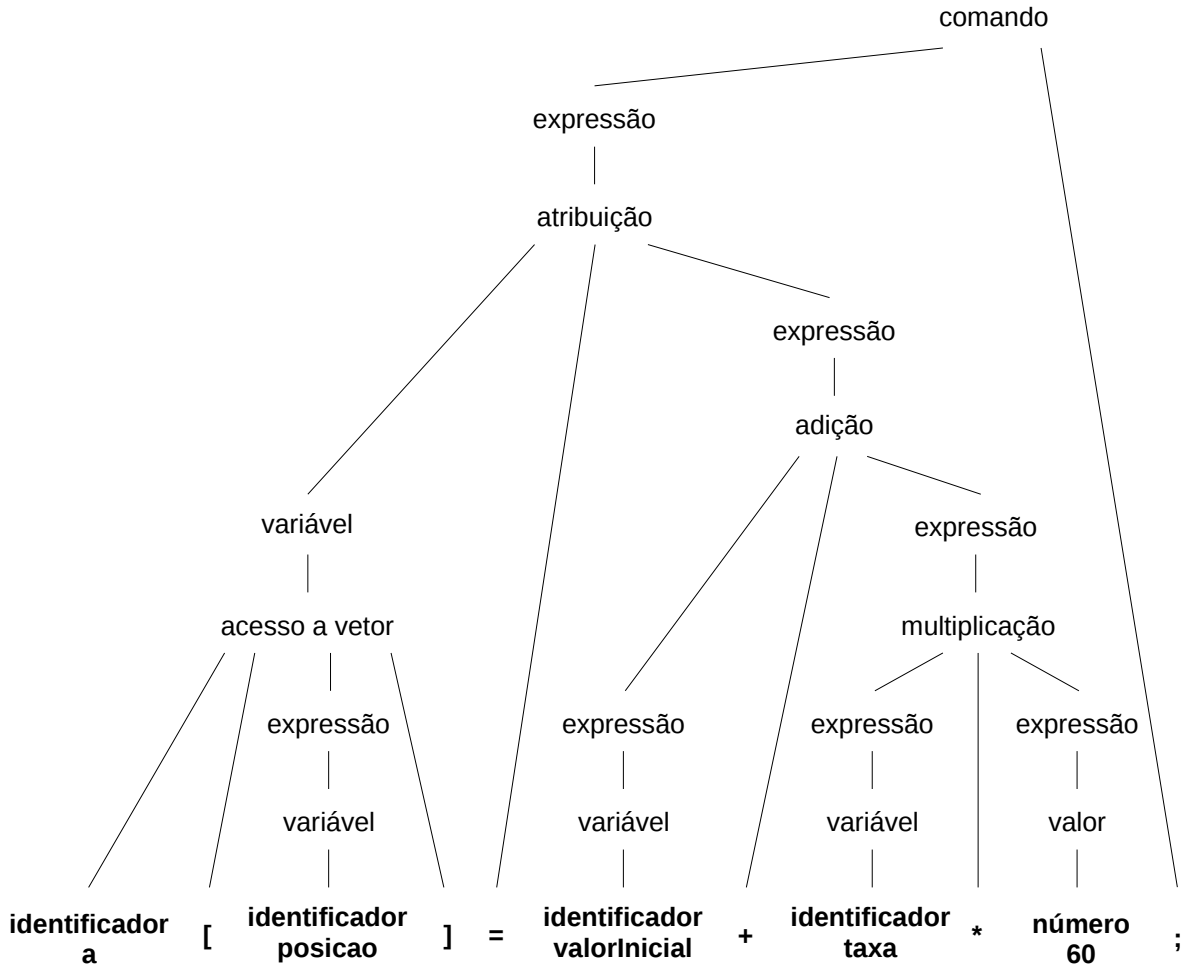
A partir da especificação dos *tokens* de uma linguagem de programação na forma de expressões regulares, há ferramentas que geram automaticamente o analisador léxico para a linguagem especificada. Mais informações sobre a construção automatizada de compiladores são apresentadas na seção 3.2.9.

3.2.3 Analisador sintático

O **analisador sintático** ou ***parser*** recebe do analisador léxico o código-fonte representado na forma de uma sequência de *tokens* e realiza, sobre essa sequência de *tokens*, a **análise sintática** ou ***parsing***, que consiste em determinar os elementos estruturais do programa (como expressões aritméticas, comandos de atribuição ou declarações de procedimentos) e as relações entre eles, definindo assim a estrutura do programa e garantindo que ele esteja sintaticamente correto. Comumente, o resultado da análise sintática é representado como uma **árvore sintática** (AHO et al, 2006; LOUDEN, 1997; MAK, 2009).

Considerando o exemplo visto anteriormente da linha de código em C, a árvore sintática produzida para aquela sequência de *tokens* é exibida na Figura 7.

Figura 7 - Exemplo de árvore sintática



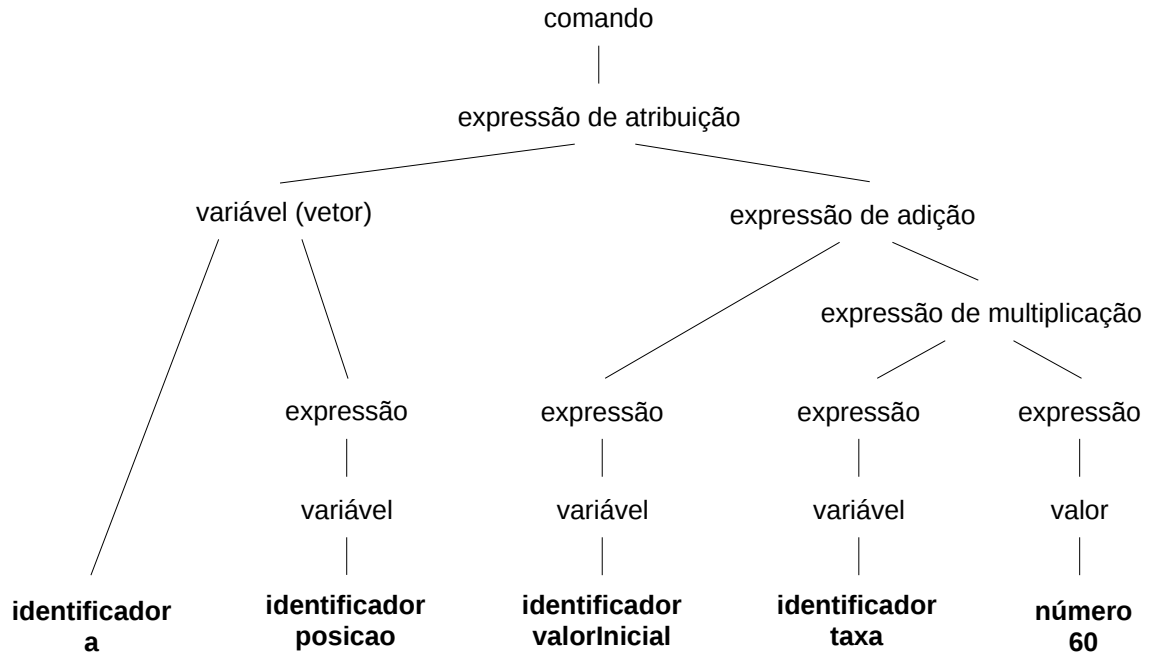
(autoria própria)

A ordem em que as operações aritméticas são consideradas na árvore sintática é consistente com as convenções matemáticas (AHO et al, 2006), que determinam, por exemplo, que a multiplicação tem maior precedência que a adição.

Vale observar que os nós internos de uma árvore sintática representam estruturas gramaticais, enquanto os nós folhas da árvore representam a sequência de *tokens* que compõem o código-fonte. Essa representação é útil para visualizar e compreender a sintaxe do código-fonte, mas não é uma representação eficaz da sua estrutura gramatical. Assim, para representar o resultado da análise sintática é mais comum utilizar, em vez da árvore sintática, a **árvore sintática abstrata**, que é uma representação mais abstrata e condensada da estrutura gramatical do código-fonte (LOUDEN, 1997).

A Figura 8 mostra a árvore sintática abstrata para o exemplo da linha de código em C visto anteriormente. Nessa representação, muitos nós desaparecem, porque podem ser subentendidos. Por exemplo, não é necessário armazenar o operador de soma se é sabido que determinada expressão é uma expressão de soma.

Figura 8 - Exemplo de árvore sintática abstrata



(autoria própria)

A sintaxe de uma linguagem de programação é definida geralmente usando um formalismo matemático chamado gramática livre de contexto (LOUDEN, 1997).

Uma **gramática livre de contexto** (ou simplesmente **gramática**) é uma quádrupla

$G=(V_N, V_T, P, S)$, na qual (AHO et al, 2006; GAGNON, 1998; SIPSER, 2005):

a) V_N é um conjunto finito de **variáveis**, chamadas **não terminais**;

b) V_T é um conjunto finito de *tokens*, chamados **terminais**;

c) $P \subseteq V_N \times (V_N \cup V_T)^*$ é um conjunto finito de **regras de substituição** ou **produções**, em que cada produção consiste de um não terminal e uma sequência de terminais e/ou não terminais; e

d) $S \in V_N$ é a **variável inicial**.

Comumente, as gramáticas são especificadas apenas por suas produções. As variáveis podem ser identificadas como os símbolos que aparecem do **lado esquerdo** e os terminais aparecem apenas do **lado direito** das produções. Por convenção, as produções cujo lado esquerdo corresponde à variável inicial são listadas primeiro (AHO et al, 2006; SIPSER, 2005). O Quadro 7 apresenta um exemplo de gramática livre de contexto.

Quadro 7 - Exemplo de gramática livre de contexto

```
A → 0A1
A → B
B → #
```

(SIPSER, 2005)

Também por convenção, produções com o mesmo lado esquerdo são normalmente agrupadas e uma barra vertical (**|**) é usada para separar os lados direitos (AHO et al, 2006; GAGNON, 1998; LOUDEN, 1997; SIPSER, 2005).

Assim, a gramática do Quadro 7 pode ser apresentada como no Quadro 8.

Quadro 8 - Exemplo de gramática livre de contexto reescrita

```
A → 0A1 | B
B → #
```

(SIPSER, 2005)

Uma gramática G descreve uma linguagem $L(G)$ a partir da geração de cada uma das suas cadeias. Inicia-se o processo pela variável inicial e repetidamente substitui-se uma variável pelo lado direito de uma produção cujo lado esquerdo contém a variável, até que não haja mais variáveis a substituir. A sequência de substituições para obter uma cadeia é chamada **derivação** (AHO et al, 2006; LOUDEN, 1997; SIPSER, 2005).

A gramática apresentada no Quadro 8 gera, por exemplo, a cadeia **000#111**. Uma derivação possível dessa cadeia naquela gramática é apresentada a seguir.

Quadro 9 - Exemplo de derivação

A ⇒ 0A1 ⇒ 00A11 ⇒ 000A111 ⇒ 000B111 ⇒ 000#111

(SIPSER, 2005)

Todas as cadeias geradas dessa maneira constituem a **linguagem da gramática**. Qualquer linguagem que possa ser gerada por uma gramática livre de contexto é chamada de **linguagem livre de contexto** (AHO et al, 2006; LOUDEN, 1997; MAK, 2009; SIPSER, 2005). A maior parte da sintaxe das linguagens de programação atuais pode ser especificada utilizando uma gramática livre de contexto (GAGNON, 1998). A árvore sintática exibida na Figura 7 é obtida por um processo análogo ao de derivação exibido no Quadro 9.

A partir da especificação da gramática de uma linguagem de programação, há ferramentas que geram automaticamente o analisador sintático para a linguagem especificada. Mais informações sobre a construção automatizada de compiladores são apresentadas na seção 3.2.9.

3.2.4 Analisador semântico

A **semântica** de um código é seu “significado” e determina seu comportamento no momento em que for executado. Os códigos escritos na maioria das linguagens de programação apresentam características que podem ser avaliadas antes da execução (ou seja, ainda durante a compilação). São exemplos comuns dessas características as declarações e os tipos de dados (LOUDEN, 1997).

O **analisador semântico** recebe do analisador sintático a árvore sintática (ou, como é mais comum, a árvore sintática abstrata) e verifica se o código-fonte está de acordo com as regras semânticas da linguagem-fonte. Ele também coleta informações extras necessárias à etapa de síntese, chamadas **atributos**, e as armazena na própria árvore sintática ou na tabela de símbolos (ver seção 3.2.8) para uso posterior (AHO et al, 2006; LOUDEN, 1997).

Uma importante tarefa do analisador semântico é realizar a **verificação de tipos**, que consiste em verificar, para cada operação, se os operandos são compatíveis entre si e se a operação é possível. Se uma violação às regras semânticas da linguagem é detectada, o

compilador interrompe a verificação e reporta um erro (AHO et al, 2006; MAK, 2009).

Por exemplo, uma linguagem de programação pode exigir que os operandos de uma operação de soma aritmética sejam de tipos numéricos, não sendo possível, por exemplo, a soma de um número com um texto.

As regras semânticas da linguagem podem permitir conversões implícitas de tipos, chamadas **coerções**. Por exemplo, a especificação da linguagem pode exigir que a soma aritmética seja efetuada: (i) entre dois números inteiros, resultando em um número inteiro; ou (ii) entre dois números reais, resultando em um número real. Caso verifique no código-fonte uma tentativa de somar um número inteiro e um número real (ou vice-versa), o compilador pode converter ou coagir o número inteiro em um número real, transformando essa soma em uma soma de números reais (AHO et al, 2006).

Ao analisar a linha de código em C considerada como exemplo, o analisador semântico verificaria que (supondo que **`a`** tenha sido declarado como um vetor de números reais; **`posicao`**, um número inteiro; e **`valorInicial`** e **`taxa`**, números reais): (i) **`a`** representa um vetor de números reais; (ii) **`posicao`** representa um número inteiro; (iii) a expressão **`a[posicao]`** é válida, e se refere a uma posição da memória que deve armazenar um número real; (iv) **`taxa`** representa um número real; (v) **`60`** é um número inteiro; (vi) a multiplicação **`taxa * 60`** é possível e resulta em um número real, contanto que **`60`** seja convertido para um número real; (vii) a soma **`valorInicial + taxa * 60`** é possível e resulta em um número real; (viii) a atribuição é possível, porque os tipos dos operandos correspondem.

3.2.5 Gerador de código intermediário

Após a análise semântica do código-fonte, muitos compiladores convertem a árvore sintática, mais apropriada à etapa de análise, em uma representação mais próxima da linguagem de máquina, mais apropriada à etapa de síntese. Essa representação deve ser, por um lado, simples de produzir a partir da árvore sintática, e por outro, simples de traduzir para linguagem de máquina. Por isso, é chamada de **representação intermediária**, e pode ser vista como um programa para uma máquina abstrata. Essa representação é produzida pelo **gerador de código intermediário** (AHO et al, 2006).

Existem muitas formas de representações intermediárias. A forma de **árvore em memória**, oriunda da análise sintática, pode ser mantida (MAK, 2009), mas a representação intermediária mais comum é o **código de três endereços** (*three-address code*), que consiste

em uma representação do código-fonte como uma sequência de instruções que possuem, no máximo, três operandos, semelhantes às instruções da linguagem *assembly*. Cada operando pode atuar como uma posição de memória (AHO et al, 2006; LOUDEN, 1997).

A representação como código de três endereços da linha de código em C utilizada como exemplo pode ser parecida com a exibida no Quadro 10.

Com relação a essa representação, vale observar que: (i) cada instrução possui no máximo um operador do lado direito das atribuições; (ii) a sequência das instruções determina, de maneira correta, a execução das operações (no exemplo, a multiplicação precede a adição); e (iii) o compilador deve gerar para cada instrução uma variável temporária para armazenar o resultado da operação que ela define (AHO et al, 2006).

Quadro 10 - Exemplo de código de três endereços

```
t1 = inttofloat(60)
t2 = taxa * t1
t3 = valorInicial + t2
a[posicao] = t3
```

(autoria própria)

3.2.6 Otimizador de código-fonte

Os compiladores podem incluir diversas fases de **otimização** (melhoramento) de código em diversos momentos do processo de compilação. As otimizações e os momentos em que elas são realizadas variam de compilador para compilador.

O primeiro momento em que uma otimização é possível é logo após a análise semântica. Nesse momento, pode-se realizar otimizações na representação intermediária que independem da linguagem-objeto. Normalmente, as otimizações objetivam aumentar a velocidade do código executável produzido, mas outras melhorias podem ser desejadas, como reduzir seu tamanho ou consumo de energia (AHO et al, 2006; LOUDEN, 1997).

Sobre o código intermediário gerado para o exemplo, um **otimizador de código-fonte** poderia deduzir que a conversão do número inteiro **`60`** para um número real pode ser realizada em tempo de compilação, de modo a dispensar a realização dessa conversão em tempo de execução cada vez que o código resultante é executado. Além disso, ele poderia deduzir também que a variável **`t3`** não é realmente necessária, pois é utilizada somente para transmitir seu valor para **`a[posicao]`**. Um otimizador de código-fonte poderia então transformar a

representação do Quadro 10 na representação mais curta do Quadro 11.

Quadro 11 - Exemplo de código de três endereços otimizado

```
t1 = taxa * 60.0
a[posicao] = valorInicial + t1
```

(autoria própria)

3.2.7 Gerador de código-objeto

O **gerador de código-objeto** recebe como entrada a representação intermediária do código-fonte do programa e produz sua tradução de fato para código na linguagem-objeto. Nessa etapa da compilação, as regras da linguagem-objeto passam a ser mais importantes que as regras da linguagem-fonte, cuja verificação já foi realizada nas etapas anteriores. Para gerar o código-objeto, o compilador toma como base, dentre outras informações, quais instruções são suportadas pela máquina que executará o código-objeto e como ela gerencia o armazenamento de dados na memória (LOUDEN, 1997).

Na maioria dos compiladores, a linguagem-objeto corresponde à linguagem de máquina, mas há compiladores cuja linguagem-objeto é a linguagem *assembly*. Esses geram código que deve ser traduzido por um montador para linguagem de máquina para que possa ser executado. Se a linguagem-objeto é a linguagem de máquina, o compilador determina como o código-objeto utilizará os registradores do processador e as células da memória principal para armazenar os valores das variáveis (AHO et al, 2006; LOUDEN, 1997).

Por exemplo, a representação intermediária do Quadro 11 poderia originar o código em linguagem *assembly* do Quadro 12.

Quadro 12 - Exemplo de código-objeto

```
MOVF    R0      taxa
MULF    R0      60.0
MOVF    R1      valorInicial
ADDF    R0      R1
MOV     R1      posicao
MUL     R1      4
MOV     R2      &a
ADD     R2      R1
MOVF    *R2     R0
```

(autoria própria)

Após a geração de código-objeto, pode haver uma fase opcional de **otimização de código-objeto**, na qual o **otimizador de código-objeto** tenta melhorar o código-objeto produzido pelo gerador de código-objeto (LOUDEN, 1997).

3.2.8 Tabela de símbolos

Um compilador deve registrar os nomes dados às variáveis, funções, constantes e tipos de dados no código-fonte, assim como determinar os atributos de cada nome. Esses atributos podem corresponder, por exemplo, a como o valor de uma variável deve ser armazenado, seu tipo e escopo (em quais locais no programa ele pode ser utilizado) ou, no caso de nomes de funções, quantos e de que tipos são seus parâmetros, como eles devem ser passados (por valor ou por referência) e o tipo de dado que retornam. Essas informações são armazenadas durante o processo de compilação na tabela de símbolos (AHO et al, 2006; LOUDEN, 1997).

A **tabela de símbolos** é uma estrutura de dados que armazena um registro para cada nome utilizado no código-fonte. Cada registro possui campos para armazenar os atributos de cada nome. As informações que são armazenadas na tabela de símbolos são coletadas durante todas as fases da etapa de análise – análise léxica, análise sintática e análise semântica – e são utilizadas ainda na etapa de análise e também na etapa de síntese – geração de código-objeto (AHO et al, 2006; LOUDEN, 1997).

Uma vez que a tabela de símbolos é acessada com frequência durante todo o processo de compilação, ela deve ser projetada de modo que o compilador possa rapidamente encontrar o registro correspondente a determinado nome e armazenar ou obter informações desse registro também rapidamente, de preferência em tempo constante (AHO et al, 2006; LOUDEN, 1997). Uma estrutura de dados comumente utilizada para construir a tabela de símbolos é a tabela *hash*. Pode-se utilizar também uma estrutura em formato de árvore (LOUDEN, 1997).

Pode ser necessário manter várias tabelas de símbolos organizadas em uma lista ou pilha, como durante a análise de funções, que podem apresentar variáveis cujo escopo se restringe às funções em que são declaradas (LOUDEN, 1997; MAK, 2009).

3.2.9 Compiladores de compiladores

O desenvolvedor de compiladores pode se beneficiar de ferramentas específicas criadas com o objetivo de auxiliar o desenvolvimento de compiladores. Essas ferramentas permitem especificar os componentes de um compilador e usam algoritmos sofisticados para implementá-los. As melhores são aquelas que abstraem o processo de implementação e geram componentes que podem ser facilmente integrados aos demais componentes do compilador (AHO et al, 2006).

Um **compilador de compilador** (*compiler-compiler*) é um *software* que recebe como entrada a definição de uma linguagem-fonte e produz partes de um compilador para essa linguagem, comumente o analisador léxico (*scanner*) e/ou o analisador sintático (*parser*). A entrada pode ser, por exemplo, a gramática da linguagem como um arquivo de texto, e a saída, código-fonte em uma linguagem de programação como Java ou C. De posse do código gerado, o desenvolvedor do compilador pode lê-lo e modificá-lo, se achar necessário, e por fim integrá-lo às demais partes para formar o compilador completo (MAK, 2009).

Exemplos de compiladores de compiladores conhecidos são: JavaCC, que recebe como entrada uma gramática e gera os códigos-fonte em Java do *scanner* e do *parser* correspondentes; Lex, capaz de produzir código-fonte de um *scanner* em C, dado um arquivo com definições de *tokens*; e Yacc, capaz de produzir código-fonte de um parser em C com base em um arquivo contendo produções (GAGNON, 1998; MAK, 2009). Também merece destaque o SableCC que, semelhante ao JavaCC, é capaz de gerar *scanners* e *parsers* em Java (APPEL; PALSBERG, 2002; GAGNON, 1998).

3.2.10 Processo de interpretação

De maneira análoga à compilação, a interpretação pode ser compreendida em duas etapas e os interpretadores podem ser decompostos em duas partes correspondentes: a análise, realizada pelo *front end*, e a síntese, pelo *back end*. Seguindo o princípio da reutilização de *software*, compiladores e interpretadores podem compartilhar o mesmo *front end*, mas seus *back ends* são diferentes (MAK, 2009; WU, NARANG; CABRAL, 2014).

Um compilador traduz um código-fonte em código-objeto e o principal componente do seu *back end* é o gerador de código-objeto. Um interpretador, por sua vez, traduz um programa nas ações que resultam de sua execução. Portanto, o principal componente do seu

back end é o executor (MAK, 2009).

O **executor** utiliza a representação intermediária e a tabela de símbolos produzidas na etapa de análise para interpretar e executar o programa (MAK, 2009). Durante a execução, podem ocorrer erros que não puderam ser verificados nas fases anteriores, como, por exemplo, divisão por zero ou tentativa de acesso a uma posição inexistente em um vetor. Diante da ocorrência desses erros, o executor deve tratá-los, possivelmente interrompendo a execução e exibindo uma mensagem de erro.

Como estrutura de dados auxiliar, o executor mantém uma **pilha de execução**, na qual ele insere e da qual ele remove registros de ativação à medida que inicia e termina, respectivamente, a execução de procedimentos e funções (MAK, 2009).

Um **registro de ativação** mantém informações a respeito da invocação (de um procedimento, de uma função ou do próprio programa principal) que está sendo executada. No caso dos procedimentos e funções, o registro de ativação mantém também os valores dos parâmetros e das variáveis locais. Se ocorrem invocações recursivas de um procedimento ou função, a pilha de execução deve conter um registro de ativação para cada invocação, de modo a manter isoladas as variáveis de cada invocação (MAK, 2009).

Cada registro de ativação contém um **mapa de memória**, que é formado por diversas células de memória. Cada célula armazena o valor associado a uma variável da invocação corrente. A célula de um vetor ou matriz contém uma célula para cada elemento do vetor ou matriz. A célula de um registro contém um mapa de memória cujas células correspondem aos campos do registro. Mantendo registros de ativação em uma pilha de execução, um executor é capaz de acessar tanto variáveis locais quanto variáveis não locais pertencentes a invocações anteriores (MAK, 2009).

4 ESPECIFICAÇÃO DA LINGUAGEM PORTUGOL

A definição formal da linguagem foi feita com base nas definições e exemplos apresentados no início e nos exercícios resolvidos ao final dos capítulos do livro Fundamentos da Programação de Computadores (ASCENCIO; CAMPOS, 2007).

4.1 ESTRUTURA SEQUENCIAL

Um **programa sequencial** (sem sub-rotinas definidas pelo programador) escrito em Portugol apresenta a seguinte estrutura básica:

Quadro 13 - Estrutura básica de um programa sequencial em Portugol

```
ALGORITMO
declaracoes
bloco_de_comandos
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 16)

declaracoes corresponde às declarações de variáveis (ver seção 4.1.1) e **bloco_de_comandos** representa todos os comandos que compõem o programa em sequência. As sub-rotinas são declaradas após o algoritmo principal (ver seção 4.6).

4.1.1 Declaração de variáveis

As **variáveis globais** são declaradas após a palavra reservada **DECLARE**, sendo uma linha para cada tipo de variável. Variáveis de um mesmo **tipo de dado** podem ser declaradas em uma mesma linha, com seus identificadores separados por vírgula e seu tipo informado após a lista de identificadores (ASCENCIO; CAMPOS, 2007, p. 16).

Quadro 14 - Declaração de variáveis em Portugol

```
DECLARE
     x NUMERICO
     y, z LITERAL
     teste LOGICO
```

(ASCENCIO; CAMPOS, 2007, p. 16)

As regras para a formação de **identificadores** em Portugol são (ASCENCIO; CAMPOS, 2007, p. 9):

a) os identificadores podem conter números, letras maiúsculas, letras minúsculas e o caractere sublinhado;
b) o primeiro caractere deve ser uma letra ou o caractere sublinhado;
c) não são permitidos espaços em branco nem caracteres especiais, como **!**, **@**, **#**, **$**, **%**, **&**, **+**, **-**, dentre outros; e
d) os identificadores não podem coincidir com as palavras reservadas da linguagem de programação.

As **palavras reservadas** da linguagem Portugol são apresentadas a seguir.

Tabela 3 - Palavras reservadas da linguagem Portugol

ALGORITMO	**FACA**	**LITERAL**	**REGISTRO**
ATE	**FALSO**	**LOGICO**	**REPITA**
DECLARE	**FIM**	**NAO**	**RETORNE**
E	**FIM_ALGORITMO**	**NUMERICO**	**SE**
ENQUANTO	**FIM_SUBROTINA**	**OU**	**SENAO**
ENTAO	**INICIO**	**PARA**	**SUB-ROTINA**
ESCREVA	**LEIA**	**PASSO**	**VERDADEIRO**

(autoria própria)

Os identificadores definidos pelo usuário não podem coincidir com nenhuma dessas palavras, assim como não podem coincidir com os nomes das sub-rotinas pré-definidas pela linguagem, apresentadas na seção 4.1.7.

Por essas regras, são exemplos de identificadores válidos: **x**, **y**, **media**, **nota1**, **salario_atual**, **SalarioMinimo**, **NOME**. São exemplos de identificadores inválidos: **1anota** (por começar com número), **nota 1** (por conter um espaço em branco), **x-y** (por possuir um caractere especial), **algoritmo** (por coincidir com palavra reservada).

Os tipos de dados definidos para a linguagem Portugol são (ASCENCIO; CAMPOS, 2007, p. 8-9,16):

a) **numérico:** usado para variáveis que devem armazenar números, como **-23**, **-23.45**, **0**, **98**, **346.89**;
b) **lógico:** usado para variáveis que devem assumir apenas os valores **VERDADEIRO** ou **FALSO**; e

c) **literal:** usado para armazenar um ou mais caracteres (letras maiúsculas, minúsculas, números e caracteres especiais) em sequência.

Convencionou-se que a linguagem é **insensível à capitalização**, ou seja, não faz distinção entre letras maiúsculas e minúsculas (ASCENCIO; CAMPOS, 2007, p. 17). Portanto, o identificador **`NUM`** equivale ao identificador **`num`**. Apenas como recurso gráfico, neste documento as palavras reservadas são grafadas em maiúsculas, tal como no livro.

4.1.2 Comando de atribuição

O **comando de atribuição** é utilizado para armazenar valores ou resultados de operações em variáveis. Em Portugol, ele é representado pelo símbolo ← (ASCENCIO; CAMPOS, 2007, p. 16). Considerando as variáveis declaradas no exemplo do Quadro 14, alguns exemplos de comandos de atribuição são mostrados no Quadro 15.

Quadro 15 - Comando de atribuição em Portugol

```
x ← 4
x ← x + 2
y ← “aula”
teste ← FALSO
```

(ASCENCIO; CAMPOS, 2007, p. 16)

Na linguagem Portugol, as constantes do tipo literal são representadas entre aspas duplas e os números reais utilizam o ponto como separador decimal (ASCENCIO; CAMPOS, 2007, p. 8).

Como não é direto inserir o símbolo ← pelo teclado, escolheu-se representar o operador de atribuição por um sinal de menor (**`<`**) seguido por um traço (**`-`**).

4.1.3 Comando de entrada de dados

O **comando de entrada** é utilizado para instruir o computador a receber dados digitados pelo usuário e armazená-los nas variáveis. Em Portugol, o comando de entrada é iniciado pela palavra reservada **`LEIA`**, seguida de uma ou mais variáveis (nesse caso, separadas por vírgulas) cujos valores deseja-se obter do usuário (ASCENCIO; CAMPOS, 2007, p. 16).

Considerando as variáveis declaradas no exemplo do Quadro 14, para executar o comando mostrado no Quadro 16, o computador deve receber um número do usuário e armazená-lo na variável **x**. Um erro deve ser informado se o usuário não digitar um número.

Quadro 16 - Exemplo de comando de entrada de dados em Portugol

```
LEIA x
```

(ASCENCIO; CAMPOS, 2007, p. 16)

Ainda considerando aquelas variáveis, para executar o comando a seguir, o computador deve ler um ou vários caracteres digitados pelo usuário e armazená-los na variável **y**.

Quadro 17 - Outro exemplo de comando de entrada de dados em Portugol

```
LEIA y
```

(ASCENCIO; CAMPOS, 2007, p. 16)

4.1.4 Comando de saída de dados

O **comando de saída** é utilizado para instruir o computador a mostrar dados na tela. Em Portugol, o comando de saída é iniciado pela palavra reservada **ESCREVA**, seguida de uma ou mais expressões (nesse caso, separadas por vírgulas), que podem ser valores constantes ou variáveis (ASCENCIO; CAMPOS, 2007, p. 17).

Considerando as variáveis declaradas no exemplo do Quadro 14, para executar o comando a seguir, o computador deve mostrar na tela o número armazenado na variável **x**.

Quadro 18 - Exemplo de comando de saída de dados em Portugol

```
ESCREVA x
```

(ASCENCIO; CAMPOS, 2007, p. 17)

Ainda considerando aquelas variáveis, para executar o comando a seguir, o computador deve exibir na tela a mensagem “Conteúdo de y = ” e, logo em seguida, a sequência de caracteres armazenados na variável **y**.

Quadro 19 - Outro exemplo de comando de saída de dados em Portugol

```
ESCREVA "Conteúdo de y = ", y
```

(ASCENCIO; CAMPOS, 2007, p. 17)

Convencionou-se que cada comando de saída produz uma linha na tela. Assim, ao executar um comando de saída, o computador deve exibir na tela, sem mudar de linha, o valor de cada uma das expressões em sequência e, ao final da execução do comando, passar o cursor para a próxima linha.

4.1.5 Comentários

Comentários são textos que podem ser inseridos em códigos-fonte com o objetivo de documentá-los. Os comentários são ignorados pelo compilador durante a análise do código-fonte (ASCENCIO; CAMPOS, 2007, p. 24).

Ascencio e Campos (2007) não definem comentários para a linguagem Portugol. Por ser esse um recurso comumente encontrado nas linguagens de programação, decidiu-se permitir **comentários de linha**, em que a região de um comentário é iniciada por duas barras em sequência (`//`) e encerrada automaticamente ao final da linha, a semelhança dos comentários de linha definidos para as linguagems C e C++ (ASCENCIO; CAMPOS, 2007, p. 24). O Quadro 20 apresenta um exemplo de comentário válido na linguagem Portugol.

Quadro 20 - Comentário em Portugol

```
LEIA x // Obtém o valor de x do usuário
```

(autoria própria)

4.1.6 Operadores

Os operadores aritméticos definidos para a linguagem Portugol são mostrados na Tabela 4 e são aplicáveis somente aos dados numéricos.

Os operadores relacionais definidos para a linguagem Portugol são mostrados na Tabela 5 e são aplicáveis aos dados numéricos. Os operadores de igualdade e diferença são aplicáveis também a dados de mesmo tipo.

Os operadores lógicos definidos para a linguagem Portugol são mostrados na Tabela 6 e são aplicáveis somente aos dados lógicos.

Tabela 4 - Operadores aritméticos da linguagem Portugol

Operador	Exemplo	Descrição
`+`	`x + y`	Soma o valor de **x** e de **y**
`-`	`x - y`	Subtrai o valor de **x** do valor de **y**
`*`	`x * y`	Multiplica o valor de **x** pelo valor de **y**
`/`	`x / y`	Obtém o quociente da divisão de **x** por **y**
`+`	`+x`	Equivale a multiplicar **x** por `+1`
`-`	`-x`	Equivale a multiplicar **x** por `-1`

(autoria própria)

Tabela 5 - Operadores relacionais da linguagem Portugol

Operador	Exemplo	Descrição
`=`	`x = y`	O valor de **x** é igual ao valor de **y**
`<>`	`x <> y`	O valor de **x** é diferente do valor de **y**
`<`	`x < y`	O valor de **x** é menor que o valor de **y**
`<=`	`x <= y`	O valor de **x** é menor ou igual ao valor de **y**
`>`	`x > y`	O valor de **x** é maior que o valor de **y**
`>=`	`x >= y`	O valor de **x** é maior ou igual ao valor de **y**

(autoria própria)

Tabela 6 - Operadores lógicos da linguagem Portugol

Operador	Exemplo	Descrição
`OU`	`p OU q`	O valor de **p** ou o valor de **q**
`E`	`p E q`	O valor de **p** e o valor de **q**
`NAO`	`NAO p`	Nega o valor de **p**

(autoria própria)

Quando dois ou mais operadores aparecem em sequência em uma expressão, as operações são executadas na ordem estabelecida pelas convenções matemáticas (a multiplicação é executada antes da adição, por exemplo). As regras de precedência adotadas para os operadores são semelhantes às regras definidas para a linguagem C (MICROSOFT, 2015), comuns a muitas linguagens de programação.

As regras de precedência para a linguagem Portugol encontram-se resumidas na Tabela 7. Os operadores estão ordenados quanto à precedência de maneira crescente: os operadores situados nas linhas mais abaixo da tabela apresentam maior precedência que os operadores nas linhas mais acima; operadores em uma mesma linha apresentam a mesma precedência, sendo avaliados na sequência determinada pela sua associatividade.

Tabela 7 - Regras de precedência dos operadores da linguagem Portugol

Operadores	Associatividade
ou (disjunção)	Esquerda para direita
e (conjunção)	Esquerda para direita
= (igualdade) e **<>** (diferença)	Esquerda para direita
< (menor), **<=** (menor ou igual), **>** (maior) e **>=** (maior ou igual)	Esquerda para direita
+ (soma) e **-** (subtração)	Esquerda para direita
***** (multiplicação) e **/** (divisão)	Esquerda para direita
+ (positivo), **-** (negativo) e **nao** (negação)	Direita para esquerda

(autoria própria)

De maneira análoga à Matemática, a linguagem Portugol permite alterar a ordem em que as operações são executadas utilizando parênteses. Assim, a expressão **`1 + 5 * 3`**, por exemplo, resulta no valor **16**, enquanto a expressão **`(1 + 5) * 3`** resulta no valor **18**.

Um exemplo de programa completo escrito em Portugol é apresentado no Quadro 21.

Quadro 21 - Exemplo de programa em Portugol

```
// Faça um algoritmo para mostrar o resultado da multiplicação de dois números.

ALGORITMO
DECLARE n1, n2, m NUMERICO
ESCREVA "Digite dois números:"
LEIA n1, n2
m <- n1 * n2
ESCREVA "Multiplicação = ", m
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 4-5)

4.1.7 Sub-rotinas pré-definidas

A linguagem Portugol possui sub-rotinas predefinidas destinadas a cálculos matemáticos, apresentadas na Tabela 8.

Tabela 8 - Sub-rotinas predefinidas destinadas a cálculos da linguagem Portugol

Sub-rotina	Descrição	Exemplo

`arredonda(x)`	**Parâmetro:**
	• **x**: número real a ser arredondado
	Retorno:
	• o arredondamento do número real **x**
		`DECLARE i NUMERICO`
		`i <- arredonda(3.6)`
		`// Equivale a`
		`// i <- 4`

`cosseno(x)`	**Parâmetro:**
	• **x**: ângulo representado em radianos
	Retorno:
	• o cosseno do ângulo **x**
		`DECLARE ang, rad, cos NUMERICO`
		`ang <- 60`
		`rad <- ang * 3.14 / 180`
		`cos <- cosseno(rad)`
		`// Equivale a`
		`// cos <- 0.5`

`parte_inteira(x)`	**Parâmetro:**
	• **x:** número real do qual se deseja obter a parte inteira
	Retorno:
	• a parte inteira do número real **x**
		`DECLARE i NUMERICO`
		`i <- parte_inteira(3.6)`
		`// Equivale a`
		`// i <- 3`

`potencia(a, b)`	**Parâmetros:**
	• **a:** a base
	• **b:** o expoente
	Retorno:
	• o número **a** elevado ao número **b**
		`DECLARE p NUMERICO`
		`p <- potencia(2, 10)`
		`// Equivale a`
		`// p <- 1024`

`raiz_enesima(n, x)`	**Parâmetros:**
	• **n:** índice da raiz
	• **x:** número do qual se deseja obter a raiz **n**
	Retorno:
	• a raiz **n** do número **x**
		`DECLARE r3 NUMERICO`
		`r3 <- raiz_enesima(3, 8)`
		`// Equivale a`
		`// r3 <- 2`

`raiz_quadrada(x)`	**Parâmetro:**
	• **x:** número do qual se deseja obter a raiz quadrada
	Retorno:
	• a raiz quadrada do número **x**
		`DECLARE r2 NUMERICO`
		`r2 <- raiz_quadrada(4)`
		`// Equivale a`
		`// r2 <- 2`

`resto(x, y)`	**Parâmetros:**
	• **x:** dividendo
	• **y:** divisor
	Retorno:
	• o resto da divisão do número **x** pelo número **y**
		`DECLARE r NUMERICO`
		`r <- resto(8, 3)`
		`// Equivale a`
		`// r <- 2`

`seno(x)`	**Parâmetro:**
	• **x:** ângulo representado em radianos
	Retorno:
	• o seno do ângulo **x**
		`DECLARE ang, rad, sen NUMERICO`
		`ang <- 30`
		`rad <- ang * 3.14 / 180`
		`sen <- seno(rad)`
		`// Equivale a`
		`// sen <- 0.5`

(autoria própria)

Além dessas, foram definidas outras sub-rotinas, apresentadas na Tabela 9.

Tabela 9 - Outras sub-rotinas predefinidas da linguagem Portugol

Sub-rotina	Descrição	Exemplo
limpar_tela()	apaga todos os caracteres da tela	**limpar_tela()**
obtenha_ano(data)	**Parâmetro:** • **data:** data obtida pela sub-rotina **obtenha_data()** **Retorno:** • o ano da **data** fornecida	**DECLARE hoje, ano NUMERICO** **hoje <- obtenha_data()** **ano <- obtenha_ano(hoje)**
obtenha_data()	**Retorno:** • a diferença, medida em milissegundos, entre a data atual e 01/01/1970	**DECLARE hoje NUMERICO** **hoje <- obtenha_data()**
obtenha_dia(data)	**Parâmetro:** • **data:** data obtida pela sub-rotina **obtenha_data()** **Retorno:** • o dia da **data** fornecida	**DECLARE hoje, dia NUMERICO** **hoje <- obtenha_data()** **dia <- obtenha_dia(hoje)**
obtenha_hora(horario)	**Parâmetro:** • **horario:** horário obtido pela sub-rotina **obtenha_horario()** **Retorno:** • as horas do **horario** fornecido	**DECLARE agora, hora NUMERICO** **agora <- obtenha_horario()** **hora <- obtenha_hora(agora)**
obtenha_horario()	**Retorno:** • a diferença, medida em milissegundos, entre a data e hora atuais e 01/01/1970 00:00	**DECLARE agora NUMERICO** **agora <- obtenha_horario()**
obtenha_mes(data)	**Parâmetro:** • **data:** data obtida pela sub-rotina **obtenha_data()** **Retorno:** • o mês da **data** fornecida	**DECLARE hoje, mes NUMERICO** **hoje <- obtenha_data()** **mes <- obtenha_mes(hoje)**
obtenha_minuto(horario)	**Parâmetro:** • **horario:** horário obtido pela sub-rotina **obtenha_horario()** **Retorno:** • os minutos do **horario** fornecido	**DECLARE agora, minuto NUMERICO** **agora <- obtenha_horario()** **minuto <- obtenha_minuto(agora)**
obtenha_segundo(horario)	**Parâmetro:** • **horario:** horário obtido pela sub-rotina **obtenha_horario()** **Retorno:** • os segundos do **horario** fornecido	**DECLARE agora, segundo NUMERICO** **agora <- obtenha_horario()** **segundo <- obtenha_segundo(agora)**

(autoria própria)

A linguagem Portugol não permite a manipulação de cadeias de caracteres nem o manuseio de arquivos. Ascencio e Campos (2007) em nenhum momento definem operações como, por exemplo, concatenação de cadeias de caracteres ou escrita em arquivo para a linguagem Portugol.

4.2 ESTRUTURAS CONDICIONAIS

A **estrutura condicional** permite estabelecer que um comando ou bloco de comandos deve ser executado somente se determinada condição for verdadeira. Opcionalmente, pode-se especificar também um comando ou bloco de comandos para ser executado se a condição for falsa. O primeiro tipo de estrutura condicional é chamado de **estrutura condicional simples**, e o segundo, **estrutura condicional composta** (ASCENCIO; CAMPOS, 2007, p. 50).

4.2.1 Estrutura condicional simples

A estrutura condicional simples em Portugol apresenta a seguinte estrutura básica:

Quadro 22 - Estrutura condicional simples em Portugol

```
SE condicao ENTAO
     comando
```

(ASCENCIO; CAMPOS, 2007, p. 50)

comando é um comando que deve ser executado apenas se **condicao**, que pode ser uma variável ou expressão do tipo lógico, apresentar o valor **VERDADEIRO**. No lugar de apenas um comando, é possível especificar um bloco de comandos, delimitando-o com as palavras reservadas **INICIO** e **FIM** (ASCENCIO; CAMPOS, 2007, p. 50):

Quadro 23 - Estrutura condicional simples utilizando blocos de comandos em Portugol

```
SE condicao ENTAO
INICIO
     comando_1
     comando_2
     …
     comando_m
FIM
```

(ASCENCIO; CAMPOS, 2007, p. 50)

Nesse caso, os comandos contidos no bloco de comandos serão executados em sequência caso **condicao** apresente o valor **VERDADEIRO**.

Um exemplo de programa que utiliza a estrutura condicional simples é apresentado no Quadro 24.

Quadro 24 - Exemplo de uso de estrutura condicional simples em Portugol

```
// Faça um programa que receba dois números e mostre o maior.

ALGORITMO
DECLARE num1, num2 NUMERICO
ESCREVA "Digite o primeiro número:"
LEIA num1
ESCREVA "Digite o segundo número:"
LEIA num2
SE num1 > num2 ENTAO
     ESCREVA "O maior número é: ", num1
SE num2 > num1 ENTAO
     ESCREVA "O maior número é: ", num2
SE num1 = num2 ENTAO
     ESCREVA "Os números são iguais"
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 60)

4.2.2 Estrutura condicional composta

A estrutura condicional composta em Portugol apresenta a seguinte estrutura básica:

Quadro 25 - Estrutura condicional composta em Portugol

```
SE condicao ENTAO
     comando_1
SENAO
     comando_2
```

(ASCENCIO; CAMPOS, 2007, p. 50)

comando_1 é um comando que deve ser executado apenas se **condicao**, que pode ser uma variável ou expressão do tipo lógico, apresentar o valor **VERDADEIRO**, e **comando_2** é um comando que deve ser executado caso contrário. No lugar de apenas um comando, é possível especificar um bloco de comandos, de maneira análoga à estrutura condicional simples, como é mostrado no Quadro 26.

Um exemplo de programa que utiliza a estrutura condicional composta é apresentado no Quadro 27.

Quadro 26 - Estrutura condicional composta utilizando blocos de comandos em Portugol

```
SE condicao ENTAO
INICIO
     comando_1
     comando_2
     …
     comando_m
FIM
SENAO
INICIO
     comando_1
     comando_2
     …
     comando_m
FIM
```

(ASCENCIO; CAMPOS, 2007, p. 50)

Quadro 27 - Estrutura condicional composta utilizando blocos de comandos em Portugol

```
// Faça um programa que receba um número inteiro e verifique se é par ou
// ímpar.

ALGORITMO
DECLARE num, r NUMERICO
ESCREVA "Digite um número:"
LEIA num
r <- resto(num, 2)
SE r = 0 ENTAO
     ESCREVA "O número é par"
SENAO
     ESCREVA "O número é ímpar"
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 62)

4.3 ESTRUTURAS DE REPETIÇÃO

A **estrutura de repetição** permite estabelecer que um comando ou bloco de comandos deve ser executado repetidas vezes. O número de repetições pode ser fixado ou vinculado à verificação de uma condição. Para possibilitar que a repetição seja definida da forma mais conveniente a quem desenvolve o programa, a linguagem Portugol possui três tipos de estruturas de repetição (ASCENCIO; CAMPOS, 2007, p. 93), apresentadas a seguir.

4.3.1 Estrutura de repetição PARA

A estrutura de repetição **PARA** é utilizada quando se sabe o número de vezes que a repetição deve ocorrer. Ela apresenta o seguinte formato:

Quadro 28 - Estrutura de repetição PARA em Portugol

```
PARA indice <- valor_inicial ATE valor_final FACA [PASSO n]
     comando
```

(ASCENCIO; CAMPOS, 2007, p. 93)

comando é um comando que deve ser executado para **indice** variando de **valor_inicial** até **valor_final**. Opcionalmente, pode-se definir o valor em que a variável **indice** é incrementada (ou decrementada), fornecendo um valor inteiro **n** após a palavra reservada **PASSO**, inserida após a palavra reservada **FACA**. Por padrão, se **n** não for fornecido, o valor da variável **indice** será incrementado de 1 unidade após cada repetição. No lugar de apenas um comando, é possível especificar um bloco de comandos a serem repetidos:

Quadro 29 - Estrutura de repetição PARA com vários comandos em Portugol

```
PARA indice <- valor_inicial ATE valor_final FACA [PASSO n]
INICIO
     comando_1
     comando_2
     …
     comando_m
FIM
```

(ASCENCIO; CAMPOS, 2007, p. 93)

Um exemplo de programa que utiliza a estrutura de repetição **PARA** é apresentado no Quadro 30.

4.3.2 Estrutura de repetição ENQUANTO

A estrutura de repetição **ENQUANTO** é utilizada quando não se sabe o número de vezes que um comando (ou um bloco de comandos) deve ser executado e possivelmente ele não será executado, por isso uma condição deve ser testada antes que cada iteração da repetição seja executada (ASCENCIO; CAMPOS, 2007, p. 94).

Quadro 30 - Exemplo de uso de estrutura de repetição PARA em Portugol

```
// Faça um programa que leia um valor N inteiro e positivo, calcule e
// mostre o valor de E, conforme a fórmula a seguir:
//
// E = 1 + 1/1! + 1/2! + 1/3! + ... + 1/N!

ALGORITMO
DECLARE n, euler, i, j, fat NUMERICO
LEIA n
euler <- 1
PARA i <- 1 ATE n FACA
INICIO
     fat <- 1
     PARA j <- 1 ATE i FACA
     INICIO
          fat <- fat * j
     FIM
     euler <- euler + 1/fat
FIM
ESCREVA euler
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 109)

A estrutura de repetição **ENQUANTO** apresenta o seguinte formato:

Quadro 31 - Estrutura de repetição ENQUANTO em Portugol

```
ENQUANTO condicao FACA
     comando
```

(ASCENCIO; CAMPOS, 2007, p. 94)

comando é um comando que deve ser executado enquanto **condicao**, que pode ser uma variável ou expressão do tipo lógico, apresentar o valor **VERDADEIRO**. No lugar de apenas um comando, é possível especificar um bloco de comandos a serem repetidos:

Quadro 32 - Estrutura de repetição ENQUANTO com vários comandos em Portugol

```
ENQUANTO condicao FACA
INICIO
     comando_1
     comando_2
     …
     comando_m
FIM
```

(ASCENCIO; CAMPOS, 2007, p. 94)

Um exemplo de programa que utiliza a estrutura de repetição **ENQUANTO** é apresentado no Quadro 33.

Quadro 33 - Exemplo de uso de estrutura de repetição ENQUANTO em Portugol

```
// Faça um programa que leia um conjunto não determinado de valores, um de
// cada vez, e escreva uma tabela com cabeçalho, que deve ser repetido a
// cada vinte linhas. A tabela deverá conter o valor lido, seu quadrado,
// seu cubo e sua raiz quadrada. Finalize a entrada de dados com um valor
// negativo ou zero.

ALGORITMO
DECLARE linhas, num, quad, cubo, raiz NUMERICO
LEIA num
ESCREVA "Valor - Quadrado - Cubo - Raiz"
linhas <- 1
ENQUANTO num > 0 FACA
INICIO
     quad <- num * num
     cubo <- num * num * num
     raiz <- raiz_quadrada(num)
     SE linhas < 20 ENTAO
     INICIO
          linhas <- linhas + 1
          ESCREVA num, " - ", quad, " - ", cubo, " - ", raiz
     FIM
     SENAO
     INICIO
          limpar_tela()
          linhas <- 1
          ESCREVA "Valor - Quadrado - Cubo - Raiz"
          linhas <- linhas + 1
          ESCREVA num, " - ", quad, " - ", cubo, " - ", raiz
     FIM
     LEIA num
FIM
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 129)

4.3.3 Estrutura de repetição REPITA

A estrutura de repetição **REPITA** é utilizada quando não se sabe o número de vezes que um comando (ou um bloco de comandos) deve ser executado, mas a execução deve ocorrer pelo menos uma vez, por isso uma condição deve ser testada após cada iteração para verificar se a repetição deve ocorrer uma vez mais (ASCENCIO; CAMPOS, 2007, p. 95).

A estrutura de repetição **REPITA** apresenta o seguinte formato:

Quadro 34 - Estrutura de repetição REPITA em Portugol

```
REPITA
     comando
ATE condicao
```

(ASCENCIO; CAMPOS, 2007, p. 95)

comando é um comando que deve ser executado até que **condicao**, que pode ser uma variável ou expressão do tipo lógico, apresente o valor **VERDADEIRO**.

No lugar de apenas um comando, é possível especificar um conjunto de comandos a serem repetidos:

Quadro 35 - Estrutura de repetição REPITA com vários comandos em Portugol

```
REPITA
     comando_1
     comando_2
     …
     comando_m
ATE condicao
```

(ASCENCIO; CAMPOS, 2007, p. 95)

Um exemplo de programa que utiliza a estrutura de repetição **REPITA** é apresentado no Quadro 36.

Quadro 36 - Exemplo de uso de estrutura de repetição REPITA em Portugol

```
// Faça um programa que conte regressivamente de 10 até 1.

ALGORITMO
DECLARE contador NUMERICO
contador <- 10
REPITA
      ESCREVA contador
      contador <- contador - 1
ATE contador = 0
ESCREVA "Fim!"
FIM_ALGORITMO.
```

(autoria própria)

4.4 VETORES

Um **vetor**, também conhecido por **variável composta homogênea unidimensional**, é um conjunto de posições de memória que armazenam o mesmo tipo de dado. Elas possuem o mesmo identificador, são armazenadas sequencialmente na memória e se distinguem por um índice, que indica sua localização dentro da estrutura (ASCENCIO; CAMPOS, 2007, p. 145).

4.4.1 Declaração de vetores

Os vetores são declarados após a palavra reservada **`DECLARE`**, no mesmo lugar destinado à declaração de variáveis, sendo uma linha para cada tipo de dado (ASCENCIO; CAMPOS, 2007, p. 145), como é mostrado o Quadro 37.

Quadro 37 - Declaração de vetores em Portugol

```
DECLARE nome[tamanho] tipo
```

(ASCENCIO; CAMPOS, 2007, p. 145)

`nome` é o identificador do vetor, **`tamanho`** é a quantidade de varíáveis que devem compor o vetor e **`tipo`** é o tipo dos dados que serão armazenados no vetor (numérico, lógico ou literal, como visto na seção 4.1.1, ou registro, como visto na seção 4.7).

Seja, por exemplo, a seguinte declaração de vetor:

Quadro 38 - Exemplo de declaração de vetor em Portugol

```
DECLARE x[5] NUMERICO
```

(ASCENCIO; CAMPOS, 2007, p. 145)

Ela indica que 5 posições de memória devem ser reservadas em sequência para o armazenamento de 5 números.

4.4.2 Uso de vetores

O acesso a uma posição de memória em um vetor exige que sua localização no vetor seja indicada após o identificador do vetor (ASCENCIO; CAMPOS, 2007, p. 145). Esse acesso pode ser feito de qualquer lugar onde é possível acessar uma variável.

Considerando a declaração de vetor exemplificada no Quadro 38, o comando de atribuição a seguir, por exemplo, indica que o valor **45** deve ser armazenado na posição **1** do vetor **x**.

Quadro 39 - Uso de vetores em Portugol

```
x[1] <- 45
```

(ASCENCIO; CAMPOS, 2007, p. 145)

Para preencher um vetor, uma estrutura de repetição **PARA** pode ser utilizada em conjunto com o comando de entrada de dados, como no exemplo a seguir.

Quadro 40 - Exemplo de preenchimento de vetor em Portugol

```
PARA i <- 1 ATE 5 FACA
INICIO
     ESCREVA "Digite o ", i, "º número"
     LEIA x[i]
FIM
```

(ASCENCIO; CAMPOS, 2007, p. 145-146)

De maneira semelhante, para exibir todos os valores armazenados em um vetor, uma estrutura de repetição **PARA** pode ser utilizada em conjunto com o comando de saída de dados, como no exemplo a seguir.

Quadro 41 - Exemplo de exibição de vetor em Portugol

```
PARA i <- 1 ATE 5 FACA
INICIO
     ESCREVA "Este é o ", i, "º número"
     ESCREVA x[i]
FIM
```

(ASCENCIO; CAMPOS, 2007, p. 146)

Um exemplo de programa que utiliza vetores é apresentado no Quadro 42.

Quadro 42 - Exemplo de uso de vetores em Portugol

```
// Faça um programa que preencha um vetor com nove números inteiros,
// calcule e mostre os números primos e suas respectivas posições.

ALGORITMO
DECLARE
     num[9] NUMERICO
     i, j, cont NUMERICO
PARA i <- 1 ATE 9 FACA
INICIO
     LEIA num[i]
FIM
PARA i <- 1 ATE 9 FACA
INICIO
     cont <- 0
     PARA j <- 1 ATE num[i] FACA
     INICIO
          SE resto(num[i], j) = 0 ENTAO
               cont <- cont + 1
     FIM
     SE cont <= 2 ENTAO
     INICIO
          ESCREVA num[i], " - ", i
     FIM
FIM
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 151)

4.5 MATRIZES

Uma **matriz**, também conhecida por **variável composta homogênea multidimensional**, é um conjunto de posições de memória que armazenam o mesmo tipo de dado. Elas possuem o mesmo identificador, são armazenadas sequencialmente na memória e se distinguem por índices, que indicam sua localização dentro da estrutura, sendo um índice para cada uma das dimensões da matriz (ASCENCIO; CAMPOS, 2007, p. 187).

4.5.1 Declaração de matrizes

As matrizes são declaradas após a palavra reservada **`DECLARE`**, no mesmo lugar destinado à declaração de variáveis e vetores, sendo uma linha para cada tipo de dado (ASCENCIO; CAMPOS, 2007, p. 187), como é mostrado no Quadro 43.

Quadro 43 - Declaração de matrizes em Portugol

```
DECLARE nome[dimensao_1, dimensao_2, …, dimensao_n] tipo
```

(ASCENCIO; CAMPOS, 2007, p. 187)

nome é o identificador da matriz, **dimensao_1**, **dimensao_2**, …, **dimensao_n** indicam os tamanhos das dimensões 1, 2, …, n da matriz, respectivamente, e **tipo** é o tipo dos dados que serão armazenados na matriz (numérico, lógico ou literal, como visto na seção 4.1.1, ou registro, como visto na seção 4.7).

Seja, por exemplo, a declaração de matriz apresentada no Quadro 44.

Quadro 44 - Exemplo de declaração de matriz em Portugol

```
DECLARE x[3,5] NUMERICO
```

(ASCENCIO; CAMPOS, 2007, p. 187)

Essa declaração define uma matriz bidimensional (análoga a uma tabela) em que o tamanho da primeira dimensão (linha) é 3 e o da segunda dimensão (coluna) é 5.

4.5.2 Uso de matrizes

O acesso a uma posição de memória em uma matriz exige que sua localização na matriz seja indicada após o identificador da matriz (ASCENCIO; CAMPOS, 2007, p. 188). Esse acesso pode ser feito de qualquer lugar onde é possível acessar uma variável ou vetor.

Considerando a declaração de matriz exemplificada no Quadro 44, o comando de atribuição a seguir, por exemplo, indica que o valor **13** deve ser armazenado na linha **3**, coluna **1** da matriz bidimensional **x**.

Quadro 45 - Uso de matrizes em Portugol

```
x[3,1] <- 13
```

(ASCENCIO; CAMPOS, 2007, p. 188)

Para preencher uma matriz, estruturas de repetição **PARA** (uma para cada dimensão) podem ser utilizadas em conjunto com o comando de entrada de dados, como no exemplo a seguir.

Quadro 46 - Exemplo de preenchimento de matriz em Portugol

```
PARA i <- 1 ATE 3 FACA
     PARA j <- 1 ATE 5 FACA
     INICIO
          ESCREVA "Digite o número da linha ", i, " e coluna ", j
          LEIA x[i,j]
     FIM
```

(ASCENCIO; CAMPOS, 2007, p. 188)

De maneira semelhante, para exibir todos os valores armazenados em uma matriz, estruturas de repetição **PARA** podem ser utilizadas em conjunto com o comando de saída de dados, como no exemplo a seguir.

Quadro 47 - Exemplo de exibição de matriz em Portugol

```
PARA i <- 1 ATE 3 FACA
     PARA j <- 1 ATE 5 FACA
     INICIO
          ESCREVA "Este é o número da linha ", i, " e coluna ", j
          ESCREVA x[i,j]
     FIM
```

(ASCENCIO; CAMPOS, 2007, p. 190)

Um exemplo de programa que utiliza matrizes é apresentado no Quadro 48.

4.6 SUB-ROTINAS

Sub-rotinas, também chamadas de subprogramas, são blocos de instruções que realizam tarefas específicas. Elas permitem subdividir o problema a ser resolvido pelo programa em problemas menores. As sub-rotinas podem ou não receber argumentos, assim como podem ou não retornar um valor (ASCENCIO; CAMPOS, 2007, p. 230).

Dentro das sub-rotinas pode ocorrer declaração de variáveis, chamadas **variáveis locais** porque podem ser utilizadas apenas dentro da sub-rotina e perdem seu conteúdo quando a execução da sub-rotina termina. Variáveis declaradas no algoritmo são chamadas **variáveis globais** porque podem ser acessadas em qualquer ponto do programa e são destruídas apenas quando a execução deste termina (ASCENCIO; CAMPOS, 2007, p. 231).

Quadro 48 - Exemplo de uso de matrizes em Portugol

```
// Faça um programa que preencha uma matriz M(2x2), calcule e mostre a
// matriz R, resultante da multiplicação dos elementos de M pelo seu maior
// elemento.

ALGORITMO
DECLARE mat[2,2], resultado[2,2], i, j, maior NUMERICO
PARA i <- 1 ATE 2 FACA
INICIO
     PARA j <- 1 ATE 2 FACA
     INICIO
          LEIA mat[i,j]
     FIM
FIM
maior <- mat[1,1]
PARA i <- 1 ATE 2 FACA
INICIO
     PARA j <- 1 ATE 2 FACA
     INICIO
          SE mat[i,j] > maior ENTAO
               maior <- mat[i,j]
     FIM
FIM
PARA i <- 1 ATE 2 FACA
INICIO
     PARA j <- 1 ATE 2 FACA
     INICIO
          resultado[i,j] <- maior * mat[i,j]
     FIM
FIM
PARA i <- 1 ATE 2 FACA
INICIO
     PARA j <- 1 ate 2 FACA
     INICIO
          ESCREVA resultado[i,j]
     FIM
FIM
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 198)

4.6.1 Passagem de parâmetros

A passagem de parâmetros ocorre por **referência**, ou seja, os parâmetros passados para a sub-rotina correspondem aos endereços de memória representados pelas variáveis. Assim, é possível alterar o valor do parâmetro durante a execução da sub-rotina e perceber essa alteração na volta para a execução do algoritmo principal (ASCENCIO; CAMPOS, 2007, p. 239). Um exemplo de programa que utiliza passagem de parâmetros por referência é apresentado no Quadro 49.

Quadro 49 - Exemplo de passagem de parâmetros por referência em Portugol

```
// Crie um programa que carregue uma matriz 3x4 com números reais. Utilize
// uma função para copiar todos os valores da matriz para um vetor de doze
// posições. Este vetor deverá ser mostrado no programa principal.

ALGORITMO
DECLARE mat[3,4], vet[12], i, j NUMERICO
PARA i <- 1 ATE 3 FACA
INICIO
     PARA j <- 1 ATE 4 FACA
     INICIO
          LEIA mat[i,j]
     FIM
FIM
gera_vetor(mat, vet)
PARA i <- 1 ATE 12 FACA
INICIO
     ESCREVA vet[i]
FIM
FIM_ALGORITMO

SUB-ROTINA gera_vetor(m[3,4], v[12] NUMERICO)
DECLARE i, j, k NUMERICO
k <- 1
PARA i <- 1 ATE 3 FACA
INICIO
     PARA j <- 1 ATE 4 FACA
     INICIO
          v[k] <- m[i,j]
          k <- k + 1
     FIM
FIM
FIM_SUB_ROTINA gera_vetor
```

(ASCENCIO; CAMPOS, 2007, p. 265)

4.6.2 Retorno

Para encerrar a execução de uma sub-rotina e retornar à execução do algoritmo principal deve-se utilizar o comando **RETORNE**, fornecendo-lhe o valor a ser retornado (ASCENCIO; CAMPOS, 2007, p. 230).

Um exemplo de programa que utiliza uma sub-rotina que retorna valor é apresentado no Quadro 50.

Quadro 50 - Exemplo de uso de sub-rotina que retorna valor em Portugol

```
// Faça um programa contendo uma sub-rotina que retorne 1 se o número
// digitado for positivo ou 0 se for negativo.

ALGORITMO
DECLARE num, x NUMERICO
LEIA num
x <- verifica(num)
SE x = 1 ENTAO
     ESCREVA "Número positivo"
SENAO
     ESCREVA "Número negativo"
FIM_ALGORITMO

SUB-ROTINA verifica(num NUMERICO)
DECLARE res NUMERICO
SE num >= 0 ENTAO
     res <- 1
SENAO
     res <- 0
RETORNE res
FIM_SUB_ROTINA verifica
```

(ASCENCIO; CAMPOS, 2007, p. 246)

4.7 REGISTROS

Um **registro**, também conhecido por **variável composta heterogênea**, é uma estrutura de dados capaz de agregar informações. Cada informação contida em um registro é chamada de **campo**. Os campos de um registro podem ser de diferentes tipos primitivos, vetores, matrizes, ou até mesmo registros (ASCENCIO; CAMPOS, 2007, p. 303).

4.7.1 Declaração de registros

Os registros são declarados após a palavra reservada **`DECLARE`**, no mesmo lugar destinado à declaração de variáveis, vetores e matrizes. Para cada registro, deve-se definir nomes e tipos de dados para seus campos (ASCENCIO; CAMPOS, 2007, p. 303):

Quadro 51 - Declaração de registros em Portugol

```
DECLARE nome REGISTRO (nome_do_campo_1 tipo_do_campo_1, nome_do_campo_2
tipo_do_campo_2, …, nome_do_campo_n tipo_do_campo_n)
```

(ASCENCIO; CAMPOS, 2007, p. 303)

O Quadro 52 mostra um exemplo de declaração de registro.

Também é possível declarar vetores ou matrizes de registros, como no exemplo do Quadro 53. Nesse caso, em cada posição de memória do vetor ou matriz será armazenado um registro com os campos definidos.

Quadro 52 - Exemplo de declaração de registro em Portugol

```
DECLARE conta registro (num, saldo NUMERICO nome LITERAL)
```

(ASCENCIO; CAMPOS, 2007, p. 303)

Quadro 53 - Exemplo de declaração de registro em Portugol

```
DECLARE conta[3] registro (num, saldo NUMERICO nome LITERAL)
```

(ASCENCIO; CAMPOS, 2007, p. 303)

4.7.2 Uso de registros

Para acessar um campo de um registro, é necessário indicar o nome da variável e o nome do campo desejado, separados por um ponto (ASCENCIO; CAMPOS, 2007, p. 303).

Considerando a declaração de registro exemplificada no Quadro 52, o comando de atribuição a seguir, por exemplo, indica que o valor **12** deve ser armazenado no campo **num** do registro **conta**.

Quadro 54 - Uso de registros em Portugol

```
conta.num <- 12
```

(ASCENCIO; CAMPOS, 2007, p. 304)

Considerando a declaração de vetor de registro exemplificada no Quadro 53, o comando de saída a seguir, por exemplo, indica que o valor armazenado no campo **num** do registro na posição **i** do vetor **conta** deve ser exibido para o usuário.

Quadro 55 - Uso de vetor de registros em Portugol

```
ESCREVA conta[i].num
```

(autoria própria)

Um exemplo de programa que utiliza um vetor de registros é mostrado no Quadro 56.

Quadro 56 - Exemplo de uso de vetor de registros em Portugol

```
// Faça um programa que realize o cadastro de contas bancárias com as
// seguintes informações: número da conta, nome do cliente e saldo. O
// banco permitirá o cadastramento de apenas quinze contas e não poderá
// haver mais que uma conta com o mesmo número. Crie o menu de opções a
// seguir.
//
// Menu de opções:
// 1. Cadastrar contas.
// 2. Visualizar todas as contas de determinado cliente.
// 3. Excluir a conta com menor saldo (supondo a não-existência de saldos
// iguais).
// 4. Sair.

ALGORITMO
DECLARE
     conta[15] REGISTRO (num, saldo NUMERICO nome LITERAL)
     i, op, posi, achou, num_conta, menor_saldo NUMERICO
     nome_cliente LITERAL
PARA i <- 1 ATE 15 FACA
INICIO
     conta[i].num <- 0
     conta[i].nome <- ""
     conta[i].saldo <- 0
FIM
posi <- 1
REPITA
     ESCREVA "Menu de Opções"
     ESCREVA "1. Cadastrar contas"
     ESCREVA "2. Visualizar todas as contas de determinado cliente"
     ESCREVA "3. Excluir a conta com menor saldo"
     ESCREVA "4. Sair"
     ESCREVA "Digite sua opção:"
     LEIA op
     SE op < 1 OU op > 4 ENTAO
          ESCREVA "Opção inválida!"
     SE op = 1 ENTAO
     INICIO
          SE posi > 15 ENTAO
               ESCREVA "Todas as contas já foram cadastradas!"
          SENAO
          INICIO
               achou <- 0
               ESCREVA "Digite o número da conta a ser incluída:"
               LEIA num_conta
               PARA i <- 1 ATE posi - 1 FACA
               INICIO
                    SE num_conta = conta[i].num ENTAO
                         achou <- 1
               FIM
               SE achou = 1 ENTAO
                    ESCREVA "Já existe conta cadastrada com esse número!"
               SENAO
               INICIO
```

```
                    conta[posi].num <- num_conta
                    ESCREVA "Digite o nome do cliente:"
                    LEIA conta[posi].nome
                    ESCREVA "Digite o saldo do cliente:"
                    LEIA conta[posi].saldo
                    ESCREVA "Conta cadastrada com sucesso:"
                    posi <- posi + 1
               FIM
          FIM
     FIM
     SE op = 2 ENTAO
     INICIO
          ESCREVA "Digite o nome do cliente a ser consultado:"
          LEIA nome_cliente
          achou <- 0
          PARA i <- 1 ATE posi - 1 FACA
          INICIO
               SE conta[i].nome = nome_cliente ENTAO
               INICIO
                    ESCREVA conta[i].num, " - ", conta[i].saldo
                    achou <- 1
               FIM
          FIM
          SE achou = 0 ENTAO
               ESCREVA "Não existe conta cadastrada para este cliente!"
     FIM
     SE op = 3 ENTAO
     INICIO
          SE posi = 1 ENTAO
               ESCREVA "Nenhuma conta foi cadastrada!"
          SENAO
          INICIO
               menor_saldo <- conta[1].saldo
               achou <- 1
               i <- 2
               ENQUANTO i < posi FACA
               INICIO
                    SE conta[i].saldo < menor_saldo ENTAO
                    INICIO
                         menor_saldo <- conta[i].saldo
                         achou <- i
                    FIM
                    i <- i + 1
               FIM
               SE achou <> posi - 1 ENTAO
                    PARA i <- achou + 1 ATE posi - 1 FACA
                    INICIO
                         conta[i - 1].num <- conta[i].num
                         conta[i - 1].nome <- conta[i].nome
                         conta[i - 1].saldo <- conta[i].saldo
                    FIM
               ESCREVA "Conta excluída com sucesso!"
               posi <- posi - 1
          FIM
     FIM
ATE op = 4
FIM_ALGORITMO.
```

(ASCENCIO; CAMPOS, 2007, p. 313-314)

Esta página foi intencionalmente deixada em branco.

5 IMPLEMENTAÇÃO

Para o desenvolvimento da aplicação foi utilizada a linguagem de programação Java em conjunto com a plataforma Java Standard Edition, abreviadamente Java SE (ORACLE, 2015c), respeitando padrões de codificação e de uso de tecnologias (ORACLE, 1999).

A linguagem Java permite utilizar o paradigma da programação orientada a objetos para criar aplicações *desktop* multiplataforma, que podem ser executadas em diferentes sistemas operacionais, dispensando aos desenvolvedores dessas aplicações a tarefa de implementar o suporte a cada um desses sistemas. Java possibilita o desenvolvimento de aplicações robustas, apesar de ser mais simples que outras linguagens, como C ou C++, o que confere maior agilidade e eficiência ao desenvolvimento (GOSLING; MCGILTON, 1996).

No caso específico do desenvolvimento de compiladores e interpretadores, segundo Appel e Palsberg (2002), a linguagem Java é vantajosa principalmente por: (i) ser segura, no sentido de não permitir que programas burlem o sistema de tipos e violem abstrações; e (ii) possuir coleta de lixo, o que simplifica o gerenciamento de memória alocada dinamicamente.

O *framework* SableCC para desenvolvimento de compiladores na linguagem Java (GAGNON, 1998; SABLECC, 2015) foi utilizado para implementar o interpretador. Esse *framework* simplifica o desenvolvimento de interpretadores de duas maneiras: (i) ele utiliza técnicas da orientação a objetos e do sistema de tipos seguro da linguagem Java para construir uma árvore sintática abstrata que é baseada na gramática da linguagem compilada; e (ii) ele utiliza uma versão modificada do padrão de projeto *visitor* para gerar classes de passeio pela árvore sintática abstrata, o que permite a implementação de ações sobre seus nós.

Os analisadores léxico e sintático e as classes que correspondem aos *tokens* e às produções foram gerados pelo *framework* SableCC com base no arquivo que especifica a gramática da linguagem (ver Apêndice A). O analisador sintático e o analisador léxico trabalham em conjunto, de modo que o primeiro sempre solicita ao segundo o próximo *token*, e este o obtém, se houver, do código-fonte fornecido como entrada. O processo segue até que não haja mais *tokens*, caso em que é retornada a árvore sintática abstrata correspondente ao código-fonte, ou até que um erro léxico ou sintático seja encontrado, caso em que é lançada uma exceção, que é tratada pela aplicação com uma mensagem indicando o erro ao usuário.

A classe que implementa o passeio pela árvore também é gerada pelo *framework* SableCC. Por herança, é possível personalizar as ações executadas ao visitar cada nó. Dessa maneira foram implementados o analisador semântico e o executor. A estrutura de árvore gerada pelo analisador sintático foi mantida como representação intermediária, o que permitiu

a utilização das facilidades oferecidas pelo SableCC em todo o processo de interpretação. Erros semânticos e de execução também originam exceções, que também são tratadas com mensagens de erro indicativas.

Utilizou-se o componente RSyntaxTextArea (FIFESOFT, 2015b) para a implementação do editor de código, que apresenta, além das funcionalidades comuns a editores de texto (recortar, copiar, colar, desfazer, refazer, localizar, substituir), funcionalidades que auxiliam o programador: (i) numeração de linhas; (ii) realce de sintaxe; (iii) compleção de código, feita com o auxílio do componente AutoComplete (FIFESOFT, 2015a); (iv) endentação; (v) casamento de parênteses (útil para verificar se há algum parêntese faltando, especialmente em expressões em que são utilizados em quantidade); e (vi) realce da linha atual.

Para implementar o restante da interface gráfica do ambiente de desenvolvimento utilizou-se a biblioteca Swing (ORACLE, 2015d), que se mostrou vantajosa por: (i) ser parte da plataforma Java SE, assim não precisa ser embutida na aplicação, cujo tamanho é reduzido; e (ii) conferir à aplicação o mesmo visual em todas as plataformas em que é executada.

Para o controle de versão dos arquivos do projeto foi utilizado o sistema Git (GIT, 2015). Um sistema de controle de versão registra toda a evolução do projeto, cada alteração sobre cada arquivo (DIAS, 2011).

Para auxiliar o desenvolvimento, foi utilizado o ambiente de desenvolvimento integrado Eclipse IDE for Java Developers (THE ECLIPSE FOUNDATION, 2015), que suporta a construção de aplicações utilizando a linguagem, a plataforma e o sistema de controle de versão escolhidos.

De forma geral, optou-se por utilizar ferramentas de domínio público, largamente utilizadas pelo mercado e pela academia, com vasta documentação disponível, com os recursos necessários ao projeto, e de fácil aprendizado.

O desenvolvimento da ferramenta ocorreu de forma iterativa e incremental. As entregas corresponderam a cada um dos componentes do interpretador: analisador léxico, analisador sintático, analisador semântico e executor. Cada um desses componentes foi testado com vários exemplos disponíveis em Ascencio e Campos (2007), para verificar sua conformidade com as regras da linguagem Portugol. Após a entrega desses, seguiu a entrega da interface gráfica do ambiente de desenvolvimento e da implantação na Internet.

6 RESULTADOS

Para fins de demonstração da aplicação, considere-se como exemplo o algoritmo para mostrar o resultado da multiplicação de dois números, apresentado no Quadro 21.

Na primeira entrega, o interpretador contava apenas com o analisador léxico. Para verificar seu funcionamento, ele foi implementado de maneira a imprimir a lista de *tokens* identificados no código-fonte fornecido como entrada. A lista de *tokens* retornada para o exemplo considerado é apresentada no Quadro 57. Para facilitar a leitura, os *tokens* foram agrupados de acordo com a linha em que se encontravam no código do exemplo em Portugol.

Quadro 57 - Saída produzida pelo analisador léxico

```
TPalavraReservadaAlgoritmo TEspacoEmBranco

TPalavraReservadaDeclare TEspacoEmBranco TIdentificador TVirgula
TEspacoEmBranco TIdentificador TVirgula TEspacoEmBranco TIdentificador
TEspacoEmBranco TPalavraReservadaNumerico TEspacoEmBranco

TPalavraReservadaEscreva TEspacoEmBranco TCadeiaDeCaracteres
TEspacoEmBranco

TPalavraReservadaLeia TEspacoEmBranco TIdentificador TVirgula
TEspacoEmBranco TIdentificador TEspacoEmBranco

TIdentificador TEspacoEmBranco TOperadorAtribuicao TEspacoEmBranco
TIdentificador TEspacoEmBranco TOperadorVezes TEspacoEmBranco
TIdentificador TEspacoEmBranco

TPalavraReservadaEscreva TEspacoEmBranco TCadeiaDeCaracteres TVirgula
TEspacoEmBranco TIdentificador TEspacoEmBranco

TPalavraReservadaFimAlgoritmo TPonto
```

(autoria própria)

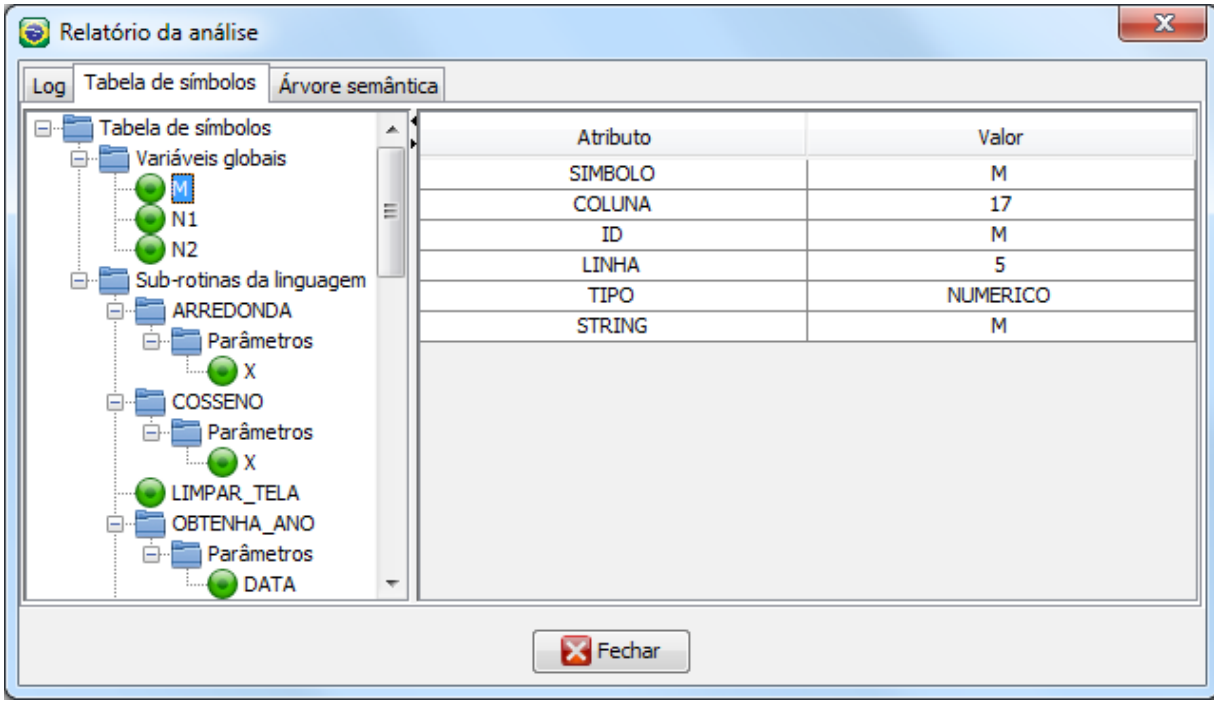
Nas entregas seguintes, foram acrescentados ao interpretador o analisador sintático e o analisador semântico. Para verificar o funcionamento do interpretador, foi desenvolvida uma tela que exibia o resultado da análise, dividido em três partes: (i) um *log* contendo mensagens informando qual nó da árvore sintática abstrata havia acabado de ser visitado pelo analisador semântico (ver Quadro 58); (ii) a tabela de símbolos, representada na verdade por uma árvore, para facilitar a localização dos símbolos, e pela tabela contendo os atributos do símbolo selecionado (ver Figura 9); e (iii) uma representação gráfica da árvore sintática abstrata, com a discriminação em tabela dos atributos computados para o nó selecionado (ver Figura 10).

Quadro 58 - Saída produzida pelo analisador semântico

```
01: Variável N1 do tipo NUMERICO declarada na linha 5, coluna 9
02: Variável N2 do tipo NUMERICO declarada na linha 5, coluna 13
03: Variável M do tipo NUMERICO declarada na linha 5, coluna 17
04: A expressão "Digite dois números:" é do tipo LITERAL
05: Comando de saída analisado
06: A posição de memória N1 é do tipo NUMERICO
07: A posição de memória N2 é do tipo NUMERICO
08: Comando de entrada analisado
09: A posição de memória M é do tipo NUMERICO
10: A posição de memória N1 é do tipo NUMERICO
11: A expressão N1 é do tipo NUMERICO
12: A posição de memória N2 é do tipo NUMERICO
13: A expressão N2 é do tipo NUMERICO
14: A expressão (N1 * N2) é do tipo NUMERICO
15: Comando de atribuição analisado, os tipos da posição de memória M e da
expressão (N1 * N2) são compatíveis
16: A expressão "Multiplicação = " é do tipo LITERAL
17: A posição de memória M é do tipo NUMERICO
18: A expressão M é do tipo NUMERICO
19: Comando de saída analisado
20: Algoritmo analisado
21: Tudo OK
```

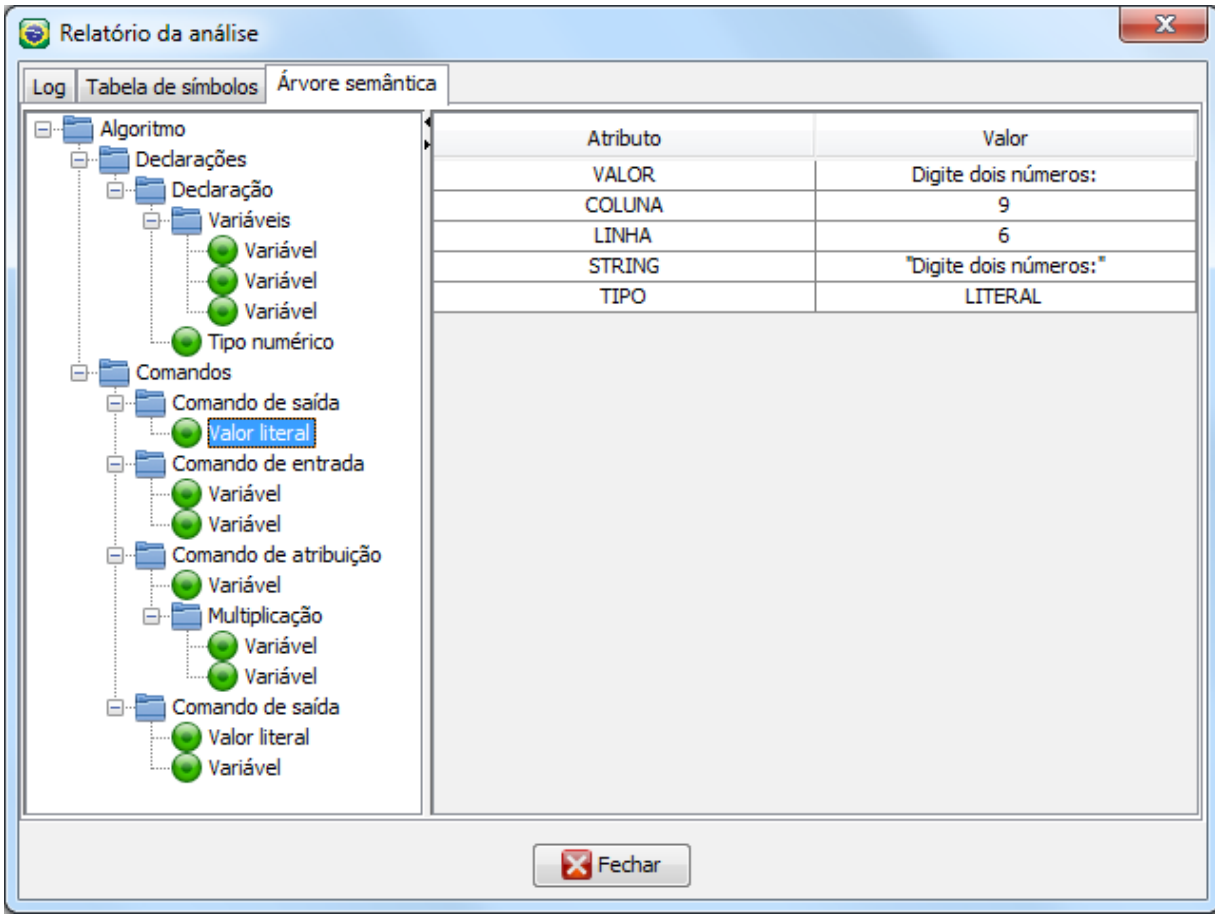
(autoria própria)

Figura 9 - Tabela de símbolos



(autoria própria)

Figura 10 - Representação gráfica da árvore sintática abstrata



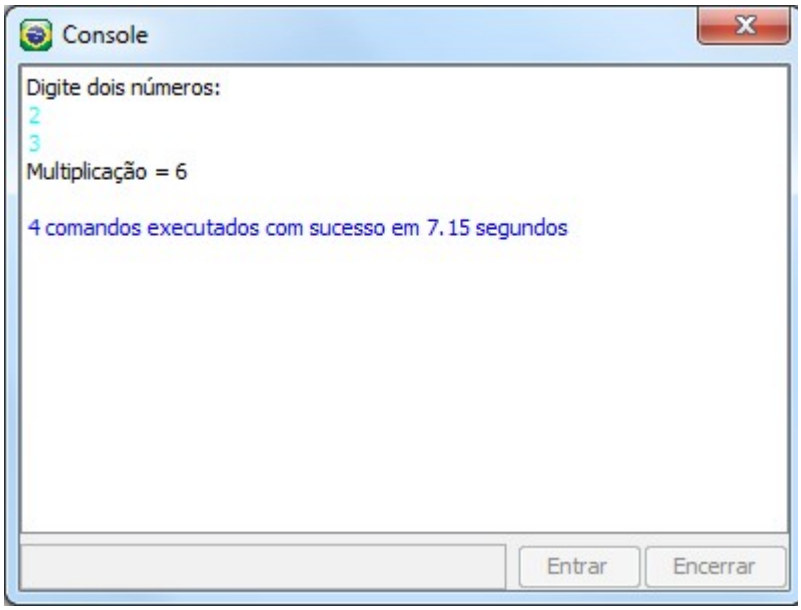
(autoria própria)

Essa tela foi mantida na aplicação final, de modo a permitir que os programadores iniciantes compreendam como o computador interpreta o código-fonte fornecido.

Na sequência, foi entregue o executor. Foi implementado um console, com o objetivo de simular o console do computador, pelo qual o usuário pode interagir com o programa por ele fornecido. A Figura 11 mostra o resultado da execução do programa considerado como exemplo (Quadro 21) para dois valores fornecidos.

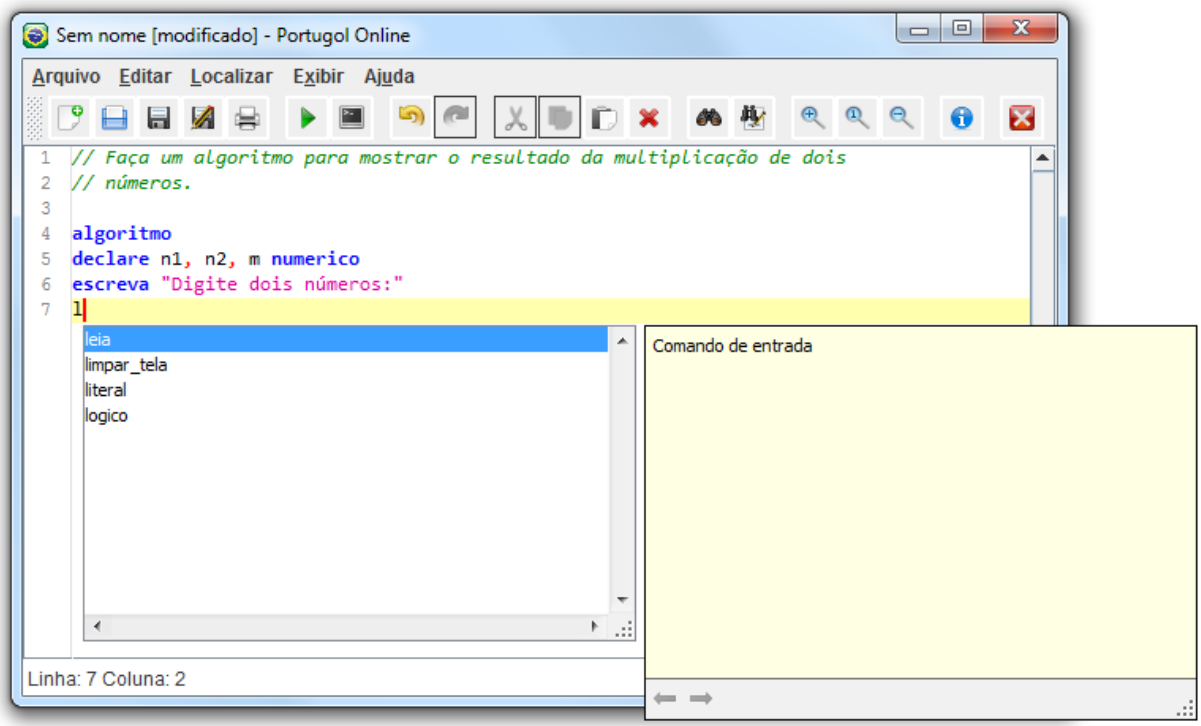
A penúltima entrega correspondeu ao ambiente de desenvolvimento integrado com o interpretador, funcionando como aplicação *desktop*. A Figura 12 mostra a tela principal do ambiente de desenvolvimento, com destaque para o realce de sintaxe e a compleção de código com ajuda.

Figura 11 - Execução do programa



(autoria própria)

Figura 12 - Ambiente integrado de desenvolvimento como aplicação *desktop*

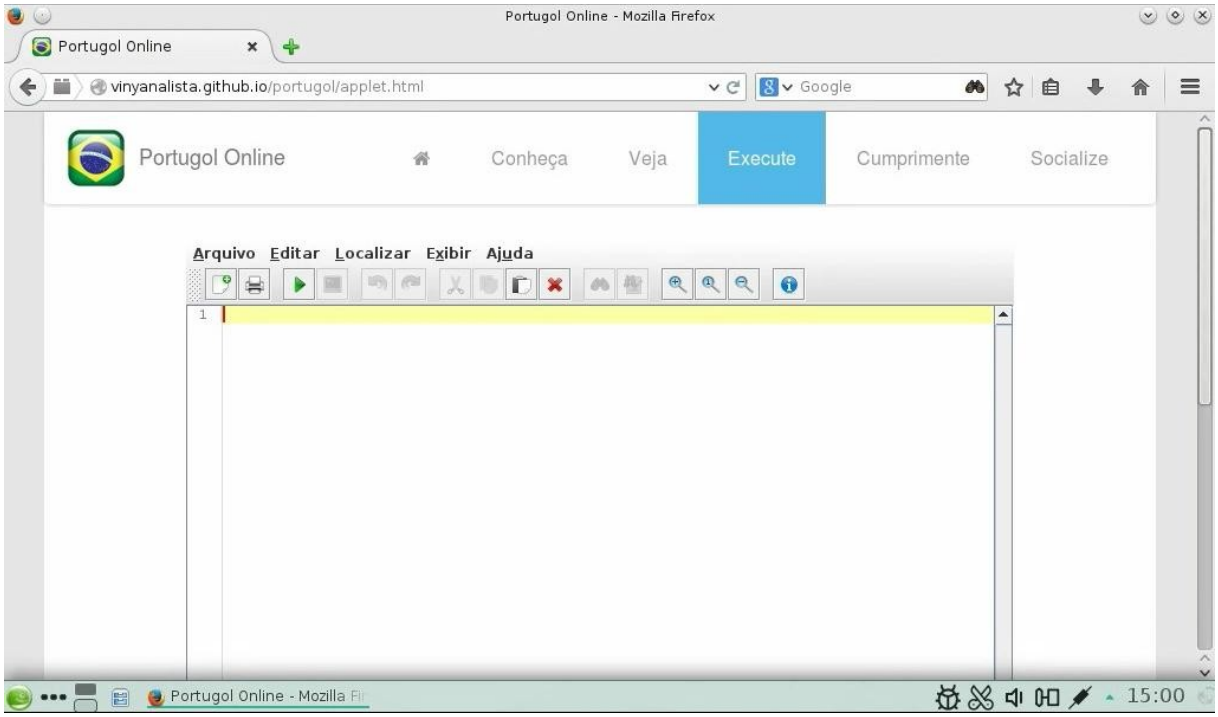


(autoria própria)

A última entrega correspondeu à implantação da aplicação na Internet. Para isso, foi desenvolvido um *site* (<http://vinyanalista.github.io/portugol>) com base em um modelo já pronto (SHAPEBOOTSTRAP, 2015). Além da aplicação *desktop* para execução *off-line*, o ambiente de desenvolvimento integrado passou a contar com mais duas possibilidades de execução: via Java Web Start (ORACLE, 2015a) ou como *applet* (ORACLE, 2015b), ambas suportadas pela plataforma Java SE.

A Figura 13 mostra o ambiente de desenvolvimento integrado sendo executado como *applet* no navegador.

Figura 13 - Ambiente de desenvolvimento integrado como *applet*



(autoria própria)

Com o intuito de verificar do funcionamento da ferramenta, o *link* para a página foi divulgado e foi solicitado aos visitantes deixar um comentário ao final, relatando suas experiências com a ferramenta, que recebeu avaliações positivas. Um usuário reportou um defeito (*bug*), que foi corrigido.

A divulgação ocorreu por meio do compartilhamento do *link* para a página nas redes sociais e da publicação em uma comunidade de usuários de *software* livre de um tutorial que apresenta a ferramenta e instrui sobre como utilizá-la (MEDEIROS, 2015).

Esta página foi intencionalmente deixada em branco.

7 TRABALHOS RELACIONADOS

Na literatura, não há um consenso sobre as regras da linguagem Portugol. Este trabalho não é o primeiro a propor uma gramática ou uma ferramenta para suportá-la.

Variações da linguagem Portugol são definidas em vários livros e trabalhos acadêmicos, a exemplo de: Araújo (2009); Ascencio e Campos (2007); Bergsten (2012); Farrer et al (1999); Forbellone e Eberspächer (2005); Guimarães e Lages (1995); Instituto Politécnico de Tomar (2015); Instituto Politécnico de Viana do Castelo (2015); Manzano e Oliveira (2014); Menezes (2003); Miranda (2004); Santiago e Dazzi (2004); Vargas e Martins (2005); e Vilarim (2004). Vale mencionar também o trabalho de Evaristo e Crespo (2010), que define a Linguagem Algorítmica Executável, semelhante à linguagem Portugol.

Dentre esses livros e trabalhos, merece destaque o livro de Ascencio e Campos (2007), indicado como uma das referências no curso técnico de Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia de Sergipe (IFS) e na graduação em Ciência da Computação da Universidade Federal de Sergipe (UFS).

Uma pesquisa na Internet por ferramentas, disponíveis para *download*, que suportam variações da linguagem Portugol revelou a existência das seguintes:

a) Bipide (Ambiente de Desenvolvimento Integrado para a Arquitetura dos Processadores BIP) (BIPIDE, 2015; VIEIRA; RAABE; ZEFERINO; 2010): visa apoiar o ensino não apenas de algoritmos, mas também da arquitetura e organização de computadores. Traduz o algoritmo fornecido pelo usuário em Portugol para código *assembly* que pode ser executado por um processador BIP. Em seguida, simula a execução desse código pelo processador, exibindo seu funcionamento por meio de animação. Requer o sistema operacional Windows;
b) Portugol IDE (INSTITUTO POLITÉCNICO DE TOMAR, 2015; MANSO; OLIVEIRA; MARQUES, 2009): *software* livre que permite o desenvolvimento, a execução e a depuração de algoritmos nas linguagens algorítmica e fluxográfica. Pode ser executado nos sistemas operacionais Windows, Linux e Mac OS X;
c) G-Portugol (ARAÚJO, 2009; G-PORTUGOL, 2015): projeto de *software* livre que define uma variação de Portugol homônima e fornece um compilador, capaz de traduzir para C, *assembly* ou código executável, um interpretador e um ambiente de desenvolvimento integrado. Pode ser executado nos sistemas operacionais Windows e Linux, mas é distribuído na forma de código-fonte, de modo que deve ser primeiro compilado para que então possa ser utilizado, o que dificulta sua

adoção por programadores iniciantes;

d) Portugol Studio (NOSCHANG et al, 2014; PORTUGOL, 2015): ambiente de desenvolvimento que integra editor de código-fonte, compilador, depurador e ajuda. Utiliza uma variação de Portugol mais próxima de linguagens como C, Java e PHP do que outras notações encontradas nos livros. Objetiva, dessa maneira, reduzir o impacto na transição do Portugol para linguagens profissionais após a fase inicial de aprendizagem. É um projeto de *software* livre e está disponível para os sistemas operacionais Windows, Linux e Mac OS X;
e) Portugol Viana (INSTITUTO POLITÉCNICO DE VIANA DO CASTELO, 2015; JESUS, 2011): projeto de *software* livre desenvolvido a partir do Portugol IDE que define a linguagem Portugol Viana e fornece uma ferramenta multiplataforma que possibilita a edição, verificação, execução e depuração de algoritmos codificados nessa linguagem;
f) UFMA-CP (MACP, 2015; NETO et al, 2008): ambiente de desenvolvimento integrado livre e multiplataforma que suporta a variação de Portugol proposta por Vilarim (2004). Permite executar, depurar e traduzir o algoritmo desenvolvido em Portugol para C++ ou linguagem de máquina, com a geração de código executável;
g) VisuAlg (APOIO INFORMÁTICA, 2015; SOUZA, 2009): é uma ferramenta semelhante a um ambiente de desenvolvimento formal, como Delphi ou Visual Basic, mas voltada para o meio acadêmico. Utiliza uma variação de Portugol inspirada na linguagem Pascal. Apesar de ser gratuito, é um *software* proprietário disponível apenas para o sistema operacional Windows; e
h) WebPortugol (HOSTINS; RAABE, 2007; UNIVALI, 2015): ferramenta *web*, *software* livre, que conecta um STI (sistema tutor inteligente) a um ambiente de desenvolvimento na forma de um *applet* do Java. Esse ambiente fornece ao STI informações sobre o desempenho do aluno ao resolver atividades propostas pelo professor. Essas informações auxiliam o STI a identificar o perfil de aprendizagem do aluno e fornecer-lhe as atividades mais adequadas. A ferramenta permite ao professor disponibilizar casos de teste junto às questões, de modo que o aluno pode, de maneira autônoma, verificar a corretude das suas soluções.

Verifica-se nas ferramentas encontradas uma tendência à utilização *off-line* (apenas o WebPortugol é uma ferramenta *web*), ao suporte a múltiplas plataformas e à disponibilização do código-fonte (apenas o Bipide e o VisuAlg não têm seu código-fonte disponível publicamente e são restritos ao sistema operacional Windows). Vale observar que ao acessar o

site do WebPortugol (UNIVALI, 2015), vê-se uma recomendação para usar o Portugol Studio.

Apesar de não suportarem uma linguagem denominada Portugol, também estão disponíveis para *download* na Internet as ferramentas ILA (Interpretador de Linguagem Algoritmica), que suporta a Linguagem Algorítmica Executável (CRESPO, 2015; EVARISTO; CRESPO, 2010), e AMBAP (Ambiente para o Aprendizado de Programação), inspirado no ILA (ALMEIDA et al, 2002; PROJETO, 2015).

Outros trabalhos acadêmicos descrevem ferramentas que aparentemente não estão disponíveis para *download*.

Alguns desses trabalhos apresentam compiladores para a linguagem Portugol, a exemplo: do Portugol/Plus (ESMIN, 1998), que gera códigos-fonte em Pascal a partir de algoritmos em Portugol; do Happy Portugol (FISCHER, 2006), que gera códigos executáveis; e do compilador de Portugol para *assembly* apresentado por Bergsten (2012).

Há outros trabalhos que sugerem ferramentas que realizam interpretação, como: o ComPort – Compilador Portugol (MENEZES, 2003); o CIFluxProg (Construtor e Interpretador de Fluxograma para Programação) (SANTIAGO; DAZZI, 2004) e sua evolução, o CIFluxProgII (MIRANDA, 2004), que permitem representar algoritmos em Portugol ou em fluxogramas; e o ambiente de desenvolvimento apresentado por Vargas e Martins (2005), que dispõe tanto de interpretador quanto de compilador.

Há ainda um trabalho que sugere a existência de uma ferramenta *online*, o WEB-UNERJOL (Ferrandin e Stephani, 2005).

Este trabalho também não é o primeiro a propor uma ferramenta para facilitar o ensino de programação. Trabalhos semelhantes nesse sentido tendem a propor a utilização de uma pseudolinguagem (HOSTINS; RAABE, 2007) e uma ferramenta que a suporte, mas há trabalhos que propõem outros recursos, como animações (GUIMARÃES, 1995 apud MIRANDA, 2004), fluxogramas (MENDES; GOMES, 2000), jogos (RAPKIEWICZ et al, 2006), micromundos (LEWIS, 2010; MARQUES, 2013), sistemas tutores inteligentes (RAABE; GIRAFFA, 2006; RAABE; SILVA, 2005) e ambientes virtuais de aprendizagem (MOREIRA; FAVERO, 2009; ROCHA et al, 2010; SANTIAGO; PEIXOTO; SILVA, 2011).

O presente trabalho apresentou uma linguagem formal o mais próxima possível da variação de Portugol apresentada em Ascencio e Campos (2007) e uma ferramenta que a suporta. Batizada de Portugol Online, essa ferramenta é um ambiente de desenvolvimento integrado que possibilita a edição, verificação e execução de algoritmos nessa linguagem. Ele foi disponibilizado para *download* ou execução *online* em seu *site* (<http://vinyanalista.github.io/portugol>), onde também é possível obter seu código-fonte.

Nenhum dos trabalhos aqui apresentados, nem mesmo este, apresenta uma solução definitiva, única e completa para todas as dificuldades de aprendizagem apontadas na seção 1. No entanto, todos apresentam contribuições significativas para sanar ou, pelo menos, amenizar a interferência de uma ou mais daquelas dificuldades listadas.

8 CONCLUSÃO

Durante o tempo dedicado a esse trabalho, o graduando realizou revisão bibliográfica sobre linguagens de programação e tradutores, pesquisou sobre as metodologias ágeis e as ferramentas necessárias ao desenvolvimento do projeto, formalizou a linguagem Portugol e desenvolveu uma ferramenta para suportá-la, que foi batizada de Portugol Online. Ela pode ser executada a partir do seu *site* (<http://vinyanalista.github.io/portugol>) via Java Web Start ou como um *applet*, ou baixada para uso *off-line*.

Espera-se que a ferramenta desenvolvida nesse trabalho possa contribuir para uma melhoria no processo de ensino-aprendizagem de programação, auxiliando alunos e professores dos cursos brasileiros de computação.

Esse trabalho foi bastante proveitoso para o graduando, pois possibilitou revisar, aplicar e consolidar conhecimentos adquiridos durante todo o curso, mas em especial nas disciplinas de programação (imperativa, orientada a objetos e para a *web*), estruturas de dados, paradigmas de programação, projeto e análise de algoritmos, desenvolvimento de *software*, interface humano-computador, linguagens formais e compiladores.

Como os autores não esperam ser os únicos detentores do conhecimento adquirido com a realização desse trabalho, todo o material relacionado a ele – esse texto, a aplicação e seu código-fonte – foi disponibilizado no *site* do Portugol Online sob licença GPL (NOVELL, 2015). A licença GPL (*General Public License*, Licença Pública Geral) garante que qualquer pessoa tenha acesso livre e irrestrito aos materiais disponibilizados nos seus termos, o que tende a atrair não apenas usuários, mas também colaboradores para o projeto.

Sugestões para trabalhos futuros incluem:

a) Realizar um estudo de caso, com a utilização da ferramenta em uma disciplina de introdução à programação;
b) Adicionar à linguagem suporte à manipulação de cadeias de caracteres e ao manuseio de arquivos;
c) Adicionar à linguagem suporte à programação orientada a objetos;
d) Desenvolver e adicionar à ferramenta um depurador (*debugger*); e
e) Desenvolver uma versão da ferramenta para dispositivos móveis.

Esta página foi intencionalmente deixada em branco.

REFERÊNCIAS

AHO, A. V. et al. **Compilers**: Principles, Techniques, and Tools. 2nd ed. Boston: Pearson Addison Wesley, 2006. ISBN: 978-0-321-48681-3.

ALMEIDA, E. S. et al. AMBAP: Um Ambiente de Apoio ao Aprendizado de Programação. In: Workshop sobre Educação em Computação, 10., 2002, Florianópolis. **Anais do XXII Congresso da Sociedade Brasileira de Computação**, Porto Alegre: Sociedade Brasileira de Computação, 2002.

APPEL, A. A.; PALSBERG, J. **Modern compiler implementation in Java**. 2nd ed. Cambridge: Cambridge University Press, 2002. ISBN: 978-0-521-82060-8.

APOIO INFORMÁTICA. **VisuAlg**. 2015. Disponível em: <http://www.apoioinformatica.inf.br/produtos/visualg>. Acesso em: 29 mar. 2015.

ARAÚJO, A. B. **Investigação e extensão de uma ferramenta para auxílio ao ensino de algoritmos em ambientes GNU/Linux – G-Portugol**. 2009. 62 f. Trabalho de Conclusão de Curso (Graduação) – Curso de Ciência da Computação, Universidade do Estado do Rio Grande do Norte, Natal, 2009.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos da programação de computadores**. 2. ed. São Paulo: Pearson Prentice Hall, 2007. ISBN: 978-85-7605-148-0.

BERGSTEN, N. A. L. **Um compilador Portugol-assembly para microcontrolador**. 2012. Trabalho de Conclusão de Curso (Graduação) – Curso de Engenharia de Computação, Universidade Estadual de Feira de Santana, Feira de Santana, 2012.

BINI, E. M.; KOSCIANSKI, A. O ensino de programação de computadores em um ambiente criativo e motivador. In: Encontro Nacional de Pesquisa em Educação em Ciências, 7., 2009, Florianópolis. **Atas do VII Enpec – Encontro Nacional de Pesquisa em Educação em Ciências**, Florianópolis: Associação Brasileira de Pesquisa em Educação em Ciências, 2009. ISSN: 2176-6940.

BIPIDE – Ambiente de Desenvolvimento Integrado para os processadores BIP. 2015. Disponível em: <http://bipide.com.br/>. Acesso em: 29 mar. 2015.

CRESPO, S. **ILA – Interpretador de Linguagem Algoritmica**. 2015. Disponível em:

<http://professor.unisinos.br/wp/crespo/ila/>. Acesso em: 09 abr. 2015.

DIAS, A. F. **Conceitos Básicos de Controle de Versão de Software — Centralizado e Distribuído**. 2011. Disponível em: <http://pronus.eng.br/artigos_tutoriais/gerencia_configuracao/conceitos_basicos_controle_versao_centralizado_e_distribuido.php>. Acesso em: 09 mar. 2015.

ESMIN, A. A. A. Portugol/Plus: uma ferramenta de apoio ao ensino de lógica de programação baseado no Portugol. In: Congresso da Rede Iberoamericana de Informática Educativa, 4., 1998, Brasília. **Actas do IV Congresso da Rede Iberoamericana de Informática Educativa**, Brasília, 1998.

EVARISTO, J.; CRESPO, S. **Aprendendo a Programar Programando numa linguagem algorítmica executável (ILA)**. 2. ed. Maceió, 2010.

FARRER, H. et al. **Algoritmos Estruturados**. 3. ed. Rio de Janeiro: LTC, 1999. ISBN: 978-85-216-1180-6.

FERRANDIN, M.; STEPHANI, S. L. Ferramenta para o ensino de Programação via Internet. In: Congresso Sul Brasileiro de Computação, 1., 2005, Criciúma. **Anais do I Congresso Sul Brasileiro de Computação**, Criciúma: Universidade do Extremo Sul Catarinense, 2005. ISSN: 2359-2656.

FIFESOFT. **AutoComplete**. 2015a. Disponível em: <http://fifesoft.com/autocomplete/>. Acesso em: 10 mar. 2015.

_____. **RSyntaxTextArea**. 2015b. Disponível em: <http://fifesoft.com/rsyntaxtextarea/>. Acesso em: 10 mar. 2015.

FISCHER, M. R. **Gerador de código objeto para o Happy Portugol**. 2006. Trabalho de Conclusão de Curso (Graduação) – Curso de Ciência da Computação, Universidade do Vale do Itajaí, Itajaí, 2006.

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Pearson Prentice Hall, 2005. ISBN: 978-85-7605-024-7.

G-PORTUGOL. 2015. Disponível em: <http://sourceforge.net/projects/gpt.berlios/>. Acesso em: 09 abr. 2015.

GAGNON, E. **SableCC, an Object-Oriented Compiler Framework**. 1998. 107 f. Tese (Mestrado em Ciência da Computação) – McGill University, Montreal, 1998.

GIT. 2015. Disponível em: <http://git-scm.com/>. Acesso em: 09 mar. 2015.

GOSLING, J.; MCGILTON, H. **The Java Language Environment**. 1996. Disponível em: <http://www.oracle.com/technetwork/java/index-136113.html>. Acesso em: 09 mar. 2015.

GUIMARÃES, A. M.; LAGES, N. A. C. **Algoritmos e estruturas de dados**. Rio de Janeiro: LTC, 1995. ISBN: 978-85-216-0378-8.

GUIMARÃES, M. A. M. **Um ambiente para ensino de algoritmos introdutórios**. 1995. 233 f. Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1995.

HOSTINS, H.; RAABE, A. Auxiliando a aprendizagem de algoritmos com a ferramenta WebPortugol. In: Workshop sobre Educação em Computação, 15., 2007, Rio de Janeiro. **Anais do XXVII Congresso da Sociedade Brasileira de Computação**, Porto Alegre: Sociedade Brasileira de Computação, 2007, p. 96-105.

INSTITUTO POLITÉCNICO DE TOMAR. **Portugol IDE | Aprendizagem de algoritmia**. 2015. Disponível em: <http://www.dei.estt.ipt.pt/portugol/>. Acesso em: 09 abr. 2015.

INSTITUTO POLITÉCNICO DE VIANA DO CASTELO. **Melhoria do Processo de Ensino-Aprendizagem nas disciplinas de Programação e Algoritmos**. 2015. Disponível em: <http://portugolviana.estg.ipvc.pt/>. Acesso em: 09 abr. 2015.

JESUS, E. Teaching computer programming with structured programming language and flowcharts. In: Workshop on Open Source and Design of Communication, 2011, Lisboa. **Proceedings of the 2011 Workshop on Open Source and Design of Communication**, New York: Association for Computing Machinery, 2011, p. 45-48. ISBN: 978-1-4503-0873-1.

KOLIVER, C.; DORNELES, R. V.; CASA, M. E. Das (Muitas) Dúvidas e (Poucas) Certezas do Ensino de Algoritmos. In: Workshop sobre Educação em Informática, 12., 2004, Salvador.

Anais do XXIV Congresso da Sociedade Brasileira de Computação, Porto Alegre: Sociedade Brasileira de Computação, 2004, p. 949-960.

LEWIS, C. M. How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch. In: ACM technical symposium on Computer science education, 41st, 2010, Milwaukee. **Proceedings of the 41st ACM technical symposium on Computer science education**, New York: Association for Computing Machinery, 2010, p. 346-350. ISBN: 978-1-4503-0006-3.

LOUDEN, K. C. **Compiler Construction**: Principles and Practice. Boston: Cengage Learning, 1997. ISBN: 978-0-534-93972-4.

MACP - Compilador Portugol. 2015. Disponível em: <http://portugol.sourceforge.net/>. Acesso em: 09 abr. 2015.

MAK, R. **Writing Compilers and Interpreters**: A Software Engineering Approach. 3rd ed. Indianapolis: Wiley Plublishing, 2009. ISBN: 978-0-470-17707-5.

MANSO, A.; OLIVEIRA, L.; MARQUES, C. G. Ambiente de Aprendizagem de Algoritmos – Portugol IDE. In: Conferência Internacional de TIC na Educação, 6., 2009, Braga. **Actas da VI Conferência Internacional de TIC na Educação – Challenges 2009**, Braga: Centro de Competência da Universidade do Minho, 2009, p. 969-983. ISBN: 978-972-98456-6-6.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. **Algoritmos**: Lógica para Desenvolvimento de Programação de Computadores. 27. ed. São Paulo: Érica, 2014. ISBN: 978-85-365-0221-2.

MARQUES, M. C. P. O. **O ensino da programação no desenvolvimento de jogos através do ambiente Scratch**. 2013. Dissertação (Mestrado em Ensino de Informática) – Universidade do Minho, Braga, 2013.

MEDEIROS, A. V. M. **Portugol Online - Software livre para facilitar o estudo de algoritmos**. 2015. Disponível em: <http://www.vivaolinux.com.br/artigo/Portugol-Online-Software-livre-para-facilitar-o-estudo-de-algoritmos/>. Acesso em: 19 mar. 2015.

MENDES, A. J. N.; GOMES, A. J. Suporte à aprendizagem da programação com o ambiente SICAS. In: Congreso Iberoamericano de Informática Educativa, 5., 2000, Viña del Mar. **Actas del V Congreso Iberoamericano de Informática Educativa**, Viña del Mar, 2000.

MENEZES, N. N. C. **COMPORT – Compilador Portugol**. 2003. 125 p. Trabalho de Conclusão de Curso (Graduação) – Curso de Processamento de Dados, Universidade do Amazonas, Manaus, 2003.

MICHAELIS. **computar: Significado de "computar" no Dicionário Português Online: Moderno Dicionário da Língua Portuguesa - Michaelis - UOL**. 2015. Disponível em: <http://michaelis.uol.com.br/moderno/portugues/index.php?lingua=portugues-portugues&palavra=computar>. Acesso em: 23 out. 2014.

MICROSOFT. **Precedence and Order of Evaluation**. 2015. Disponível em: <https://msdn.microsoft.com/en-us/library/2bxt6kc4.aspx>. Acesso em: 05 mar. 2015.

MIRANDA, E. M. **Uma ferramenta de apoio ao processo de aprendizagem de algoritmos**. 2004. Tese (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis, 2004.

MONTEIRO, M. A. **Introdução à Organização de Computadores**. 5a. ed. Rio de Janeiro: LTC, 2007. ISBN: 978-85-216-1543-9.

MOREIRA, M. P.; FAVERO, E. L. Um Ambiente para Ensino de Programação com Feedback Automático de Exercícios. In: Workshop sobre Educação em Informática, 17., 2009, Bento Gonçalves. **Anais do XXIX Congresso da Sociedade Brasileira de Computação**, Porto Alegre: Sociedade Brasileira de Computação, 2009, p. 429-438.

NETO, C. S. S. et al. Compilador UFMA-CP: uma ferramenta de apoio para o ensino básico de lógica de programação. In: Jornada de Informática do Maranhão, 2., 2008, São Luís. **Anais da II Jornada de Informática do Maranhão**, São Luís: Universidade Federal do Maranhão, 2008. ISSN: 2358-8861.

NOSCHANG, L. F. et al. Portugol Studio: Uma IDE para Iniciantes em Programação. In: Workshop sobre Educação em Computação, 22., 2014, Brasília. **Anais do XXXIV Congresso da Sociedade Brasileira de Computação**, Porto Alegre: Sociedade Brasileira de Computação, 2014, p. 1287-1296. ISSN: 2175-2761.

NOVELL. **Licença Pública Geral GNU**. 2015. Disponível em: <https://pt.opensuse.org/Licença_Pública_Geral_GNU>. Acesso em: 13 mar. 2015.

ORACLE. **Code Conventions for the Java Programming Language**. 1999. Disponível em: <http://www.oracle.com/technetwork/java/index-135089.html>. Acesso em: 09 mar. 2015.

_____. **Java SE Desktop Technologies - Java Web Start Technology**. 2015a. Disponível em: <http://www.oracle.com/technetwork/java/javase/javawebstart/>. Acesso em: 10 mar. 2015.

_____. **Lesson: Java Applets**. 2015b. Disponível em: <http://docs.oracle.com/javase/tutorial/deployment/applet/>. Acesso em: 09 mar. 2015.

_____. **Oracle Technology Network for Java Developers**. 2015c. Disponível em: <http://www.oracle.com/technetwork/java>. Acesso em: 09 mar. 2015.

_____. **Swing APIs and Developer Guides**. 2015d. Disponível em: <http://docs.oracle.com/javase/8/docs/technotes/guides/swing/>. Acesso em: 10 mar. 2015.

PORTUGOL Studio. 2015. Disponível em: <http://sourceforge.net/projects/portugolstudio/>. Acesso em: 10 abr. 2015.

PROJETO Ambap - Laboratório de Desenvolvimento de Software. 2015. Disponível em: <https://sites.google.com/site/ldsicufal/softwares/projeto-ambap>. Acesso em: 09 abr. 2015.

RAABE, A. L. A.; GIRAFFA, L. M. M. Uma Arquitetura de Tutor para Promover Experiências de Aprendizagem Mediadas. In: Simpósio Brasileiro de Informática na Educação, 17., 2006, Brasília. **Anais do XVII Simpósio Brasileiro de Informática na Educação**, 2006, p. 278-287.

RAABE, A. L. A.; SILVA, J. M. C. Um Ambiente para Atendimento as Dificuldades de Aprendizagem de Algoritmos. In: Workshop sobre Educação em Informática, 13., 2005, São Leopoldo. **Anais do XXV Congresso da Sociedade Brasileira de Computação**, Porto Alegre: Sociedade Brasileira de Computação, 2005, p. 2326-2337.

RAPKIEWICZ, C. E. et al. Estratégias pedagógicas no ensino de algoritmos e programação associadas ao uso de jogos educacionais. **Revista Novas Tecnologias na Educação**, Porto Alegre, v. 4, n. 2, p. 1-11, dez. 2006. ISSN: 1679-1916.

ROCHA, P. S. et al. Ensino e Aprendizagem de Programação: Análise da Aplicação de Proposta Metodológica Baseada no Sistema Personalizado de Ensino. **Revista Novas Tecnologias na Educação**, Porto Alegre, v. 8, n. 3, dez. 2010. ISSN: 1679-1916.

RUSSI, D. F.; CHARÃO, A. S. Ambientes de Desenvolvimento Integrado no Apoio ao Ensino da Linguagem de Programação Haskell. **Revista Novas Tecnologias na Educação**, Porto Alegre, v. 9, n. 2, dez. 2011. ISSN: 1679-1916.

SABLECC. 2015. Disponível em: <http://sablecc.org/>. Acesso em: 09 mar. 2015.

SANTIAGO, E. D.; PEIXOTO, F. S.; SILVA, A. M. O Ambiente Virtual de Aprendizagem Moodle Atuando como Ferramenta de Apoio ao Ensino Presencial na Disciplina de Algoritimos. **Revista Semente**, Maceió, v. 6, n. 6, p. 2-10, 2011. ISSN: 1980-8607.

SANTIAGO, R.; DAZZI, R. L. S. Interpretador de Portugol. In: Congresso Brasileiro de Computação, 4., 2004, Itajaí. **Anais do IV Congresso Brasileiro de Computação**, Itajaí: UNIVALI, 2004, p. 26-29. ISSN: 1677-2822.

SHAPEBOOTSTRAP. **Xeon – Best Onepage Site Template**. 2015. Disponível em: <http://shapebootstrap.net/item/xeon-best-onepage-site-template/>. Acesso em: 10 mar. 2015.

SHAW, M. What Makes Good Research in Software Engineering. **International Journal on Software Tools for Technology Transfer**, New York, v. 4, n. 1, p. 1-7, oct. 2002. ISSN: 1433-2787.

SIPSER, M. **Introduction to the Theory of Computation**. 2nd ed. Boston: Cengage Learning, 2005. ISBN: 978-0-534-95097-2.

SOARES, M. S. Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software. **Revista Eletrônica de Sistemas de Informação**, Campo Largo, v. 3, n. 1, 2004. ISSN: 1677-3071.

SOUZA, C. M. VisuAlg - Ferramenta de Apoio ao Ensino de Programação. **Revista TECCEN**, Vassouras, v. 2, n. 2, p. 1-9, set. 2009. ISSN: 1984-0993.

THE ECLIPSE FOUNDATION. **Eclipse - The Eclipse Foundation open source community website**. 2015. Disponível em: <http://www.eclipse.org>. Acesso em: 09 mar. 2015.

UNIVALI. **Acesso ao WebPortugol**. 2015. Disponível em: <http://www.univali.br/webportugol>. Acesso em: 09 abr. 2015.

VARGAS, K. S.; MARTINS, J. Ferramenta para Apoio ao Ensino de Introdução à Programação. In: Seminário de Computação, 14., 2005, Blumenau. **Anais do XIV SEMINCO**, Blumenau: Universidade Regional de Blumenau, 2005, p. 79-90.

VIEIRA, P. V.; RAABE, A. L. A.; ZEFERINO, C. A. Bipide – Ambiente de Desenvolvimento Integrado para a Arquitetura dos Processadores BIP. **Revista Brasileira de Informática na Educação**, Porto Alegre, v. 18, n. 1, p. 32-43, 2010. ISSN: 2317-6121.

VILARIM, G. **Algoritmos**: Programação Para Iniciantes. 1. ed. Rio de Janeiro: Ciência Moderna, 2004. ISBN: 978-85-7393-316-1.

WU, F.; NARANG, H.; CABRAL, M. Design and Implementation of an Interpreter Using Software Engineering Concepts. **International Journal of Advanced Computer Science and Applications**, New York, v. 5, n. 7, p. 170-177, 2014. ISSN: 2156-5570.

ZANINI, A. S.; RAABE, A. L. A. Análise dos enunciados utilizados nos problemas de programação introdutória em cursos de Ciência da Computação no Brasil. In: Workshop sobre Educação em Computação, 20., 2012, Curitiba. **Anais do XXXII Congresso da Sociedade Brasileira de Computação**, Porto Alegre: Sociedade Brasileira de Computação, 2012. ISSN: 2175-2761.

APÊNDICE A – Gramática da linguagem Portugol

As expressões regulares e as produções da gramática que definem a linguagem Portugol são apresentadas a seguir, no formato como foram fornecidas ao *framework* SableCC para a geração das classes necessárias à implementação do interpretador.

Quadro 59 - Gramática da linguagem Portugol

```
/*
Disciplina : TCC
Professor  : Michel Soares
Aluno      : Antônio Vinícius Menezes Medeiros
Descrição  : arquivo de definição da linguagem Portugol
*/

Package br.com.vinyanalista.portugol.base;

Helpers
    // Linguagem insensível à capitalização
    a = 'A' | 'a' ;
    b = 'B' | 'b' ;
    c = 'C' | 'c' ;
    d = 'D' | 'd' ;
    e = 'E' | 'e' ;
    f = 'F' | 'f' ;
    g = 'G' | 'g' ;
    h = 'H' | 'h' ;
    i = 'I' | 'i' ;
    j = 'J' | 'j' ;
    k = 'K' | 'k' ;
    l = 'L' | 'l' ;
    m = 'M' | 'm' ;
```

```
    n = 'N' | 'n' ;
    o = 'O' | 'o' ;
    p = 'P' | 'p' ;
    q = 'Q' | 'q' ;
    r = 'R' | 'r' ;
    s = 'S' | 's' ;
    t = 'T' | 't' ;
    u = 'U' | 'u' ;
    v = 'V' | 'v' ;
    w = 'W' | 'w' ;
    x = 'X' | 'x' ;
    y = 'Y' | 'y' ;
    z = 'Z' | 'z' ;

    // Dígitos e letras
    digito               = ['0' .. '9'] ;
    letra                = [['a' .. 'z'] + ['A' .. 'Z']] ;
    qualquer_caractere   = [0 .. 0xffff];

    // Caracteres permitidos nas cadeias, contém caracteres especiais e acentuação
    // Referência: http://unicode-table.com
    caracteres_permitidos_nas_cadeias = [32 .. 33] | [35 .. 126] | [0x00a0 .. 0xffff];

    // Símbolos
    abre_colchete      = '[' ;
    abre_parentese     = '(' ;
    aspa_dupla         = '"' ;
    asterisco          = '*' ;
    barra              = '/' ;
    circunflexo        = '^' ;
    fecha_colchete     = ']' ;
    fecha_parentese    = ')' ;
    hifen              = '-' ;
    igual              = '=' ;
    mais               = '+' ;
    menos              = '-' ;
    ponto              = '.' ;
```

```
    sublinhado         = '_' ;
    virgula            = ',' ;

    // Caracteres não imprimíveis
    cr  = 13 ; // carriage return (\r)
    lf  = 10 ; // line feed (\n)
    fim_de_linha
        = lf // UNIX
        | (cr lf) // DOS
        | cr  // MAC
        ;
    // Referência: http://blog.glauco.mp/os-formatos-de-arquivos-unix-dos-e-mac-e-como-converte-los/
    espaco       = ' ' ;
    tabulacao    = 9 ;

Tokens
    ////////////////////////
    // Palavras reservadas //
    ////////////////////////

    // 3.1 Estrutura sequencial em algoritmos
    palavra_reservada_algoritmo        = a l g o r i t m o ;
    palavra_reservada_declare          = d e c l a r e ;
    palavra_reservada_fim_algoritmo    = f i m sublinhado a l g o r i t m o ;

    // 3.1.1 Declaração de variáveis em algoritmos
    palavra_reservada_numerico  = n u m e r i c o ;
    palavra_reservada_literal   = l i t e r a l ;
    palavra_reservada_logico    = l o g i c o ;

    // 3.1.3 Comando de entrada em algoritmos
    palavra_reservada_leia = l e i a ;

    // 3.1.4 Comando de saída em algoritmos
    palavra_reservada_escreva = e s c r e v a ;

    // 4.1.1 Estrutura condicional simples
```

```
palavra_reservada_se        = s e ;
palavra_reservada_entao     = e n t a o ;
palavra_reservada_inicio    = i n i c i o ;
palavra_reservada_fim       = f i m ;

// 4.1.2 Estrutura condicional composta
palavra_reservada_senao = s e n a o ;

// 5.1.1 Estrutura de repetição para número definido de repetições (estrutura PARA)
palavra_reservada_para     = p a r a ;
palavra_reservada_ate      = a t e ;
palavra_reservada_faca     = f a c a ;
palavra_reservada_passo    = p a s s o ;

// 5.1.2 Estrutura de repetição para número indefinido de repetições e teste no início (estrutura ENQUANTO)
palavra_reservada_enquanto = e n q u a n t o ;

// 5.1.3 Estrutura de repetição para número indefinido de repetições e teste no final (estrutura REPITA)
palavra_reservada_repita = r e p i t a ;

// 8.1 Sub-rotinas (programação modularizada)
palavra_reservada_sub_rotina        = s u b hifen r o t i n a ;
palavra_reservada_retorne           = r e t o r n e ;
palavra_reservada_fim_sub_rotina    = f i m sublinhado s u b sublinhado r o t i n a ;

// 10.2 Declaração de registros em algoritmos
palavra_reservada_registro = r e g i s t r o ;

//////////////
// Valores //
//////////////

// 1.6.1 Numérico
numero_inteiro    = (mais | menos |) (digito)+ ;
numero_real       = (mais | menos |) (digito)+ ponto (digito)+ ;

// 1.6.2 Lógico
```

```
valor_logico = v e r d a d e i r o | f a l s o ;

// 1.6.3 Literal ou caractere
cadeia_de_caracteres = aspa_dupla caracteres_permitidos_nas_cadeias* aspa_dupla ;

////////////////
// Operadores //
////////////////

// 3.1.2 Comando de atribuição em algoritmos
operador_atribuicao = '<-' ;

// Aritméticos
operador_mais            = mais ;
operador_menos           = menos ;
operador_vezes           = asterisco ;
operador_dividido_por    = barra ;

// Lógicos
operador_e                 = e ;
operador_igual             = igual ;
operador_diferente         = '<>' ;
operador_maior_ou_igual    = '>=' ;
operador_maior             = '>' ;
operador_menor_ou_igual    = '<=' ;
operador_menor             = '<' ;
operador_nao               = n a o ;
operador_ou                = o u ;

//////////////////////
// Identificadores //
//////////////////////

// 1.7 Formação de identificadores
identificador = (letra | sublinhado)(digito | letra | sublinhado)* ;

////////////////////////////
```

```
    // Separadores de símbolos //
    ////////////////////////////

    abre_colchete      = abre_colchete ;
    abre_parentese     = abre_parentese ;
    fecha_colchete     = fecha_colchete ;
    fecha_parentese    = fecha_parentese ;
    ponto              = ponto ;
    virgula            = virgula ;

    ///////////////////////
    // Espaços em branco //
    ///////////////////////

    espaco_em_branco    = (espaco | tabulacao | fim_de_linha)* ;
    comentario          = barra barra [qualquer_caractere - [cr + lf]]*;

Ignored Tokens
    espaco_em_branco, comentario;

Productions
    // 3.1 Estrutura sequencial em algoritmos
    algoritmo // Lado esquerdo da produção
    {-> algoritmo} // Equivalente na árvore sintática abstrata

        = palavra_reservada_algoritmo lista_de_declaracoes? comando* palavra_reservada_fim_algoritmo ponto? sub_rotina* // Lado
direito da produção
        {-> New algoritmo([lista_de_declaracoes.declaracao], [comando], [sub_rotina])} // Equivalente na árvore sintática abstrata
        ;

    // 3.1.1 Declaração de variáveis em algoritmos
    lista_de_declaracoes
    {-> declaracao*}

        = palavra_reservada_declare declaracao+
        {-> [declaracao]}
        ;
```

```
declaracao
{-> declaracao}

    = lista_de_variaveis tipo
    {-> New declaracao([lista_de_variaveis.variavel], tipo)}
    ;

lista_de_variaveis
{-> variavel*}

    = variavel continuacao_da_lista_de_variaveis*
    {-> [variavel, continuacao_da_lista_de_variaveis.variavel]}
    ;

continuacao_da_lista_de_variaveis
{-> variavel}

    = virgula variavel
    {-> variavel}
    ;

variavel
{-> variavel}

    = {simples} identificador
    {-> New variavel.simples(identificador)}

    | {vetor_ou_matriz} identificador abre_colchete lista_de_expressoes fecha_colchete
    {-> New variavel.vetor_ou_matriz(identificador, [lista_de_expressoes.expressao])}
    ;

// 1.6 Tipos de dados
tipo
{-> tipo}

    = {numerico} palavra_reservada_numerico
```

```
        {-> New tipo.numerico()}

        | {literal} palavra_reservada_literal
        {-> New tipo.literal()}

        | {logico} palavra_reservada_logico
        {-> New tipo.logico()}

        // 10.2 Declaração de registros em algoritmos
        | {registro} palavra_reservada_registro abre_parentese lista_de_campos fecha_parentese
        {-> New tipo.registro([lista_de_campos.declaracao])}
        ;

    lista_de_campos
    {-> declaracao*}

        = declaracao+
        {-> [declaracao]}
        ;

    comando
    {-> comando}

        = {aberto} comando_aberto
        {-> comando_aberto.comando}

        | {fechado} comando_fechado
        {-> comando_fechado.comando}
        ;

    comando_aberto
    {-> comando}

        // 4.1.1 Estrutura condicional simples
        = {condicional_simples} palavra_reservada_se expressao palavra_reservada_entao comando
        {-> New comando.condicional(expressao, comando, Null)}
```

```
        // 4.1.2 Estrutura condicional composta
        | {condicional_composta} palavra_reservada_se expressao palavra_reservada_entao comando_fechado palavra_reservada_senao
comando_aberto
        {-> New comando.condicional(expressao, comando_fechado.comando, comando_aberto.comando)}
        ;

    comando_fechado
    {-> comando}

        // 3.1.2 Comando de atribuição em algoritmos
        = {atribuicao} posicao_de_memoria operador_atribuicao expressao
        {-> New comando.atribuicao(posicao_de_memoria, expressao)}

        // 3.1.3 Comando de entrada em algoritmos
        | {entrada} palavra_reservada_leia lista_de_posicoes_de_memoria
        {-> New comando.entrada([lista_de_posicoes_de_memoria.posicao_de_memoria])}

        // 3.1.4 Comando de saída em algoritmos
        | {saida} palavra_reservada_escreva lista_de_expressoes
        {-> New comando.saida([lista_de_expressoes.expressao])}

        // 4.1.1 Estrutura condicional simples
        | {bloco} palavra_reservada_inicio comando* palavra_reservada_fim
        {-> New comando.bloco([comando])}

        // 4.1.2 Estrutura condicional composta
        | {condicional_composta} palavra_reservada_se expressao palavra_reservada_entao [entao]:comando_fechado
palavra_reservada_senao [senao]:comando_fechado
        {-> New comando.condicional(expressao, entao.comando, senao.comando)}

        // 5.1.1 Estrutura de repetição para número definido de repetições (estrutura para)
        | {repeticao_para} palavra_reservada_para variavel operador_atribuicao [inicio]:expressao palavra_reservada_ate
[fim]:expressao palavra_reservada_faca comando_fechado
        {-> New comando.repeticao_para(variavel, inicio, fim, Null, comando_fechado.comando)}

        | {repeticao_para_com_passo} palavra_reservada_para variavel operador_atribuicao [inicio]:expressao palavra_reservada_ate
[fim]:expressao palavra_reservada_faca palavra_reservada_passo [passo]:numero_inteiro comando_fechado
```

```
        {-> New comando.repeticao_para(variavel, inicio, fim, passo, comando_fechado.comando)}

        // 5.1.2 Estrutura de repetição para número indefinido de repetições e teste no início (estrutura enquanto)
        | {repeticao_enquanto} palavra_reservada_enquanto expressao palavra_reservada_faca comando_fechado
        {-> New comando.repeticao_enquanto(expressao, comando_fechado.comando)}

        // 5.1.3 Estrutura de repetição para número indefinido de repetições e teste no final (estrutura repita)
        | {repeticao_repita} palavra_reservada_repita comando* palavra_reservada_ate expressao
        {-> New comando.repeticao_repita([comando], expressao)}

        // 8.1 Sub-rotinas (programação modularizada)
        | {chamada_a_sub_rotina} chamada_a_sub_rotina
        {-> New comando.chamada_a_sub_rotina(chamada_a_sub_rotina)}

        | {retorne} palavra_reservada_retorne expressao
        {-> New comando.retorne(expressao)}
        ;

    posicao_de_memoria
    {-> posicao_de_memoria}

        = {variavel} variavel
        {-> New posicao_de_memoria.variavel(variavel)}

        // 10.2.1 Acesso aos campos de um registro em algoritmo
        | {campo} [registro]:variavel ponto [campo]:variavel
        {-> New posicao_de_memoria.campo(registro, campo)}
        ;

    // Gramática de expressões
    // Precedência das operações baseada na precedência adotada na linguagem C
    // Referência: https://msdn.microsoft.com/en-us/library/2bxt6kc4.aspx
    expressao
    {-> expressao}

        = expressao_a
        {-> expressao_a.expressao}
```

```
    ;

expressao_a
{-> expressao}

    = {disjuncao} expressao_a operador_ou expressao_b
    {-> New expressao.disjuncao(expressao_a.expressao, expressao_b.expressao)}

    | {outra} expressao_b
    {-> expressao_b.expressao}
    ;

expressao_b
{-> expressao}

    = {conjuncao} expressao_b operador_e expressao_c
    {-> New expressao.conjuncao(expressao_b.expressao, expressao_c.expressao)}

    | {outra} expressao_c
    {-> expressao_c.expressao}
    ;

expressao_c
{-> expressao}

    = {igualdade} expressao_c operador_igual expressao_d
    {-> New expressao.igualdade(expressao_c.expressao, expressao_d.expressao)}

    | {diferenca} expressao_c operador_diferente expressao_d
    {-> New expressao.diferenca(expressao_c.expressao, expressao_d.expressao)}

    | {outra} expressao_d
    {-> expressao_d.expressao}
    ;

expressao_d
{-> expressao}
```

```
    = {menor} expressao_d operador_menor expressao_e
    {-> New expressao.menor(expressao_d.expressao, expressao_e.expressao)}

    | {menor_ou_igual} expressao_d operador_menor_ou_igual expressao_e
    {-> New expressao.menor_ou_igual(expressao_d.expressao, expressao_e.expressao)}

    | {maior} expressao_d operador_maior expressao_e
    {-> New expressao.maior(expressao_d.expressao, expressao_e.expressao)}

    | {maior_ou_igual} expressao_d operador_maior_ou_igual expressao_e
    {-> New expressao.maior_ou_igual(expressao_d.expressao, expressao_e.expressao)}

    | {outra} expressao_e
    {-> expressao_e.expressao}
    ;

expressao_e
{-> expressao}

    = {soma} expressao_e operador_mais expressao_f
    {-> New expressao.soma(expressao_e.expressao, expressao_f.expressao)}

    | {subtracao} expressao_e operador_menos expressao_f
    {-> New expressao.subtracao(expressao_e.expressao, expressao_f.expressao)}

    | {outra} expressao_f
    {-> expressao_f.expressao}
    ;

expressao_f
{-> expressao}
    = {multiplicacao} expressao_f operador_vezes expressao_g
    {-> New expressao.multiplicacao(expressao_f.expressao, expressao_g.expressao)}

    | {divisao} expressao_f operador_dividido_por expressao_g
    {-> New expressao.divisao(expressao_f.expressao, expressao_g.expressao)}
```

```
    | {outra} expressao_g
    {-> expressao_g.expressao}
    ;

expressao_g
{-> expressao}

    = {positivo} operador_mais expressao_g
    {-> New expressao.positivo(expressao_g.expressao)}

    | {negativo} operador_menos expressao_g
    {-> New expressao.negativo(expressao_g.expressao)}

    | {negacao} operador_nao expressao_g
    {-> New expressao.negacao(expressao_g.expressao)}

    | {outra} expressao_h
    {-> expressao_h.expressao}
    ;

expressao_h
{-> expressao}

    = {chamada_a_sub_rotina} chamada_a_sub_rotina
    {-> New expressao.chamada_a_sub_rotina(chamada_a_sub_rotina)}

    | {expressao} abre_parentese expressao fecha_parentese
    {-> expressao}

    | {posicao_de_memoria} posicao_de_memoria
    {-> New expressao.posicao_de_memoria(posicao_de_memoria)}

    | {valor} valor
    {-> New expressao.valor(valor)}
    ;
```

```
    chamada_a_sub_rotina
    {-> chamada_a_sub_rotina}

        = identificador abre_parentese lista_de_expressoes? fecha_parentese
        {-> New chamada_a_sub_rotina(identificador, [lista_de_expressoes.expressao])}
        ;

    lista_de_expressoes
    {-> expressao*}

        = expressao continuacao_da_lista_de_expressoes*
        {-> [expressao, continuacao_da_lista_de_expressoes.expressao]}
        ;

    continuacao_da_lista_de_expressoes
    {-> expressao}

        = virgula expressao
        {-> expressao}
        ;

    // 1.6 Tipos de dados
    valor
    {-> valor}

        // 1.6.1 Numérico
        = {inteiro} numero_inteiro
        {-> New valor.inteiro(numero_inteiro)}

        | {real}    numero_real
        {-> New valor.real(numero_real)}

        // 1.6.2 Lógico
        | {logico}  valor_logico
        {-> New valor.logico(valor_logico)}

        // 1.6.3 Literal ou caractere
```

```
        | {literal} cadeia_de_caracteres
        {-> New valor.literal(cadeia_de_caracteres)}
        ;

    lista_de_posicoes_de_memoria
    {-> posicao_de_memoria*}

        = posicao_de_memoria continuacao_da_lista_de_posicoes_de_memoria*
        {-> [posicao_de_memoria, continuacao_da_lista_de_posicoes_de_memoria.posicao_de_memoria]}
        ;

    continuacao_da_lista_de_posicoes_de_memoria
    {-> posicao_de_memoria}

        = virgula posicao_de_memoria
        {-> posicao_de_memoria}
        ;

    // 8.1 Sub-rotinas (programação modularizada)
    sub_rotina
    {-> sub_rotina}

        = palavra_reservada_sub_rotina identificador abre_parentese declaracao* fecha_parentese lista_de_declaracoes? comando*
palavra_reservada_fim_sub_rotina [identificador_repetido]:identificador
        {-> New sub_rotina(identificador, [declaracao], [lista_de_declaracoes.declaracao], [comando], identificador_repetido)}
        ;

Abstract Syntax Tree
    algoritmo
        = declaracao* comando* sub_rotina*
        ;

    declaracao
        = variavel+ tipo
        ;

    variavel
```

```
        = {simples}            identificador
        | {vetor_ou_matriz}    identificador expressao+
        ;

    tipo
        = {numerico}
        | {literal}
        | {logico}
        | {registro}    declaracao+
        ;

    comando
        = {atribuicao}              posicao_de_memoria expressao
        | {entrada}                 posicao_de_memoria+
        | {saida}                   expressao*
        | {bloco}                   comando*
        | {condicional}             expressao [entao]:comando [senao]:comando?
        | {repeticao_para}          variavel [inicio]:expressao [fim]:expressao [passo]:numero_inteiro? comando
        | {repeticao_enquanto}      expressao comando
        | {repeticao_repita}        comando* expressao
        | {chamada_a_sub_rotina}    chamada_a_sub_rotina
        | {retorne}                 expressao
        ;

    posicao_de_memoria
        = {variavel}    variavel
        | {campo}       [registro]:variavel [campo]:variavel
        ;

    expressao
        = {disjuncao}               [esquerda]:expressao [direita]:expressao
        | {conjuncao}               [esquerda]:expressao [direita]:expressao
        | {igualdade}               [esquerda]:expressao [direita]:expressao
        | {diferenca}               [esquerda]:expressao [direita]:expressao
        | {menor}                   [esquerda]:expressao [direita]:expressao
        | {menor_ou_igual}          [esquerda]:expressao [direita]:expressao
        | {maior}                   [esquerda]:expressao [direita]:expressao
```

```
        | {maior_ou_igual}          [esquerda]:expressao [direita]:expressao
        | {soma}                    [esquerda]:expressao [direita]:expressao
        | {subtracao}               [esquerda]:expressao [direita]:expressao
        | {multiplicacao}           [esquerda]:expressao [direita]:expressao
        | {divisao}                 [esquerda]:expressao [direita]:expressao
        | {positivo}                expressao
        | {negativo}                expressao
        | {negacao}                 expressao
        | {posicao_de_memoria}      posicao_de_memoria
        | {chamada_a_sub_rotina}    chamada_a_sub_rotina
        | {valor}                   valor
        ;

    chamada_a_sub_rotina
        = identificador expressao*
        ;

    valor
        = {inteiro}    numero_inteiro
        | {real}       numero_real
        | {logico}     valor_logico
        | {literal}    cadeia_de_caracteres
        ;

    sub_rotina
        = identificador [parametros]:declaracao* [variaveis_locais]:declaracao* comando* [identificador_repetido]:identificador
        ;
```

(autoria própria)