UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE CIÊNCIAS EXATAS E DA TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL DOS SANTOS WEBER

Desenvolvimento de um Compilador de Português Estruturado para a JVM no Portal de Algoritmos da UCS

Ricardo Vargas Dorneles
Orientador

Alexandre Erasmo Krohn Nascimento
Coorientador

Caxias do Sul, Julho de 2016

Desenvolvimento de um Compilador de Português Estruturado para a JVM no Portal de Algoritmos da UCS

por

Gabriel dos Santos Weber

Projeto de Diplomação submetido ao curso de Bacharelado em Ciência da Computação do Centro de Ciências Exatas e da Tecnologia da Universidade de Caxias do Sul, como requisito obrigatório para graduação.

Projeto de Diplomação

Orientador: Ricardo Vargas Dorneles
Coorientador: Alexandre Erasmo Krohn Nascimento
Banca examinadora:

André Luis Martinotto
CCET/UCS
Alexandre Erasmo Krohn Nascimento
CCET/UCS

Projeto de Diplomação apresentado em
24 de Junho de 2016

Simone Cristine Mendes Paiva
Coordenadora

SUMÁRIO

LISTA DE ACRÔNIMOS

ACM	*Association for Computing Machinery*
ACM - ICPC	*ACM International Collegiate Programming Contest*
API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
AVA	Ambiente Virtual de Aprendizagem
CSS	*Cascading Style Sheet*
EAD	Ensino à Distância
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
IDEs	*Integrated Development Environments*
ISA	*Instruction Set Architecture*
JavaCC	*the Java Compiler Compiler*
JPDA	*Java Platform Debugger Architecture*
JDI	*Java Debug Interface*
JVM	*Java Virtual Machine*
LLVM	*Low Level Virtual Machine*
Moodle	*Modular Object-Oriented Dynamic Learning Environment*
NPAPI	*Netscape Plugin Application Programming Interface*
REST	*Representational State Transfer*

SSA	*Static Single Assignment form*
UCS	Universidade de Caxias do Sul
URLs	*Uniform Resource Locators*
W3C	*World Wide Web Consortium*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TRECHOS DE CÓDIGO

RESUMO

O Portal de Algoritmos da UCS é uma plataforma *web* composta de uma interface de edição e interpretação de código, em Português Estruturado, uma área com um conjunto de problemas algorítmicos, uma ferramenta para validação de soluções algorítmicas e uma interface para os professores, executando no navegador do usuário. Porém, a tecnologia utilizada no desenvolvimento do Portal está se tornando obsoleta e não encontra mais suporte nos navegadores mais utilizados atualmente. Outro aspecto do Portal é que seu interpretador foi desenvolvido de forma a não ter uma divisão entre as etapas, como por exemplo, análises léxica e sintática, geração de código intermediário e execução, bem definidas. Sendo assim, este trabalho tem como um dos objetivos o desenvolvimento de uma representação intermediária para o Portal de Algoritmos, tornando as etapas do processo de interpretação do código separadas e bem definidas, o que permitirá, assim, que o Portal possa ser expandido para outras linguagens de programação e também facilitará a inserção de mais etapas no processo de interpretação, como por exemplo a otimização de código.

Palavras-chave: Portal de Aprendizagem, Compiladores, Linguagem de Programação, Ensino, Algoritmos.

Development of an Intermediate Representation for the Algorithms Portal of UCS

ABSTRACT

The Algorithms Portal of UCS is a web platform composed of a code editor interface and an interpreter, for the Structured Portuguese language, an area containing a set of algorithmic problems, an solution assessment tool and an interface for the professors, through the client's web browser. However, the technology utilized in the development of the Portal is becoming obsolete and it's not being supported anymore by the most utilized browsers. Another aspect of the Portal is that its interpreter was designed as to not have well defined its phases, like scanning and parsing, intermediate representation generation, and execution. Therefore one of this paper goal is to develop an intermediate representation for the Algorithms Portal, making the source code interpretation phases well defined and separated, allowing the expansion of the languages interpreted by the Portal and the insertion of more phases in the process of interpreting, like code optimization.

Keywords: Learning Management System, Compilers, Programming Language, Teaching, Algorithms.

1 INTRODUÇÃO

A disciplina de algoritmos é matéria introdutória em muitos cursos de nível superior de computação no Brasil, sendo assim, muitos dos estudantes que cursam esta matéria são novatos à computação ou à programação. Tendo em vista esse perfil iniciante, faz-se necessário dispor de uma ferramenta que seja fácil de usar no auxílio do aprendizado das técnicas de programação e na qual os alunos não tenham que se preocupar com a configuração de todo um ambiente de desenvolvimento e evitar problemas com mensagens de erros de compilação (DORNELES; JUNIOR; ADAMI, 2010).

Tendo em vista a procura por uma ferramenta que seja de fácil acesso aos alunos e relativamente simples de ser utilizada, foi desenvolvido na *UCS* (Universidade de Caxias do Sul) o Portal de Algoritmos da *UCS*, ferramenta *web* que proporciona um ambiente de desenvolvimento para algoritmos utilizando a linguagem do português estruturado (DORNELES; JUNIOR; ADAMI, 2010).

Portais deste tipo existem, hoje em dia, em grande quantidade pela internet. Para citar alguns dos mais amplamente conhecidos temos o *CodeAcademy* que é um site onde é possível aprender diversas linguagens voltadas para o desenvolvimento *web*, como por exemplo *HTML* (*HyperText Markup Language*), *CSS* (*Cascading Style Sheet*), *JavaScript* entre outras. Temos como exemplo também o *Kahn Academy*, uma plataforma de ensino pela *web*, que possui uma área exclusiva sobre computação e programação. Todos esses portais possuem em comum a facilidade de uso e a interface intuitiva e amigável ao usuário, bem como lições e exercícios acerca dos assuntos de programação sendo estudados.

A linguagem escolhida para o Portal de Algoritmos da *UCS* é a do português estruturado, por se tratar de uma versão mais reduzida de uma linguagem de programação e por ser fácil a memorização dos comandos da linguagem pelos alunos (DORNELES; JUNIOR; ADAMI, 2010). Nessa linguagem, muito próxima ao da língua portuguesa, há poucos tipos de dados representados, o que facilita o aprendizado da lógica de programação e da análise e resolução de problemas de maneira estruturada e na forma de algoritmo, objetivos essenciais em um curso de algoritmos

(KAMIYA; BRANDÃO, 2009).

O Portal de Algoritmos da *UCS* é uma ferramenta utilizada pelos professores nas aulas de algoritmos para facilitar o ensino da matéria de algoritmos. É uma ferramenta baseada na internet, executada em um navegador e permite a composição, edição e execução de algoritmos desenvolvidos pelos alunos.

O Portal permite que os alunos selecionem quais problemas desejam resolver e que possam postar suas soluções para avaliação do professor. O portal apresenta também um *ranking* contendo o nome dos usuários que mais contribuíram com postagem de soluções para os problemas. A ferramenta possui também algumas características encontradas em alguns ambientes de desenvolvimento, como por exemplo, visualização das variáveis e quais valores elas possuem em dado momento de execução. Permite também executar o algoritmo linha a linha e permite incluir pontos de parada (*breakpoints*). O usuário pode também validar seu algoritmo com alguns valores de entrada padrão, fornecidos pelo problema que está resolvendo (DORNELES; JUNIOR; ADAMI, 2010).

O portal é feito atualmente em um Java *applet*, que é um programa Java especial, fornecido aos navegadores pelo servidor e executa no lado cliente. É normalmente um programa incorporado em uma página *web*, executada no contexto do navegador (ORACLE, 2015), sendo assim, toda a análise léxica, análise sintática e a geração do código intermediário são feitos no lado do cliente, no navegador do usuário, bem como a interpretação desse código, ou seja, a produção dos resultados da execução do código fonte de entrada (COOPER; TORCZON, 2014; AHO et al., 2008).

Porém, um dos problemas encontrados no portal atualmente, está no fato de que alguns navegadores a partir de 2014[1], como o *Chrome*, da Google, não executam mais aplicativos desenvolvidos em java *applet*. Aplicativos escritos utilizando Java para a *web*, como por exemplo os *applets*, são baseados na arquitetura multiplataforma *NPAPI* (*Netscape Plugin Application Programming Interface*) a qual não possui mais suporte em versões recentes de alguns navegadores como por exemplo o *Chrome*[2,3], da Google, e o *Edge* [4], da Microsoft. Já há planos de o Mozilla Firefox suspender o suporte a esses aplicativos no final de 2016[5]. Dessa forma, para que o portal de algoritmos possa continuar em uso, é preciso que configurações extras sejam feitas em alguns navegadores, sem a garantia de que possa executar de forma satisfatória.

[1] http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html
[2] https://java.com/en/download/faq/chrome.xml
[3] http://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html
[4] `https://www.java.com/pt_BR/download/faq/win10_faq.xml`
[5] https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/

1.1 Objetivos

1.1.1 Objetivo geral

Este trabalho tem por objetivo desenvolver uma representação intermediária que traduza os comandos, estruturas de dados e de controle da linguagem utilizada. Esta linguagem será, em um primeiro momento, a do português estruturado, de forma a auxiliar os alunos no aprendizado da matéria de algoritmos com uma ferramenta *web* de fácil acesso e que proporcione um ambiente de desenvolvimento intuitivo e fácil de usar.

1.2 Estrutura do trabalho

No Capítulo 2 são apresentados conceitos básicos sobre portais de aprendizagem, sobre o atual Portal de Algoritmos da *UCS*, sobre compiladores, representação intermediária, sobre a linguagem do Português Estruturado e sobre máquinas virtuais.

No Capítulo 3 são apresentadas as características da arquitetura do novo interpretador do Portal de Algoritmos da *UCS*, como a organização do *front-end*, a etapa de tradução para a representação intermediária e a implementação da máquina virtual do *back-end* que irá interpretar o código intermediário.

No Capítulo 4 é descrito como cada um dos comandos, operadores e estruturas existentes no Português Estruturado do Portal de Algoritmos da *UCS* é mapeado para as instruções da representação intermediária.

No Capítulo 5 são apresentados detalhes do projeto do serviço *web* e da estrutura da parte cliente do editor de códigos, bem como do serviço *REST* que irá fazer a comunicação entre o *front-end* executando no navegador do usuário com o interpretador executando no servidor.

2 REFERENCIAL TEÓRICO

Antes de entrar na descrição das estruturas desenvolvidas neste trabalho, faz-se importante a definição de alguns conceitos e termos que estarão presentes ao longo deste texto. Neste capítulo serão abordados assuntos referentes a portais *online* de aprendizado. Serão tratados conceitos, também, sobre compiladores e interpretadores, os quais permeiam o desenvolvimento deste trabalho. Será conceituada também a linguagem utilizada no ensino de algoritmos, o Português Estruturado e também serão abordados aqui conceitos de representação intermediária de uma linguagem.

2.1 Portais de Aprendizado Online

Os Portais de Aprendizado Online têm se tornado ferramentas de grande utilidade em cursos acadêmicos no Brasil e no mundo. Muitas universidades brasileiras, desde o início dos anos 2000, vem investindo e utilizando portais *online*, tanto em seus cursos *EAD* (Ensino à Distância) quanto no apoio aos seus cursos presenciais (VARELLA et al., 2002; FRANCO; CORDEIRO; CASTILLO, 2003). Atualmente muitas universidades, incluindo a *UCS*, possuem seus próprios *AVA* (Ambiente Virtual de Aprendizagem) onde alunos podem encontrar materiais das disciplinas que estão atualmente cursando, bem como postarem trabalhos para avaliações e contarem também, em alguns casos, com um fórum e um sistema de troca de mensagens para comunicação entre alunos e professores. Os professores também encontram nesses ambientes a possibilidade de poderem compartilhar com os alunos materiais de ensino e trabalhos a serem feitos no período letivo, bem como ferramentas de apoio ao ensino *online* (VARELLA et al., 2002; FRANCO; CORDEIRO; CASTILLO, 2003; CHAVES et al., 2014).

Como exemplo mais conhecido de um *AVA*, existe o *Moodle* (*Modular Object-Oriented Dynamic Learning Environment*). O *Moodle* consiste em uma ferramenta de aprendizagem *online* utilizada no mundo todo, de código aberto e uma boa documentação, permitindo que novas funcionalidades possam ser agregadas por desenvolvedores (KUMAR; GANKOTIYA; DUTTA, 2011). O *Moodle* conta com diversas

ferramentas que auxiliam o professor de acordo com seus objetivos de ensino (ALVES; BRITO, 2005). Por essa característica de *software* de código aberto e por ter uma grande base de usuários internacionais é que o *Moodle* vem sendo utilizado por diversas instituições do mundo, inclusive instituições de ensino brasileiras, tanto federais quanto particulares (ALVES; BRITO, 2005).

2.2 Portais de Aprendizado Online de Programação

Os portais *online* de aprendizagem de programação seguem uma linha diferente dos portais convencionais, como por exemplo dos *AVAs*, pelo fato de terem características e necessidades diferentes. Em um portal de aprendizado de programação, além das ferramentas comuns encontradas em um *AVA*, existe ainda a necessidade de uma forma automatizada de correção das submissões dos usuários, que nesses portais são os códigos fontes escritos pelos usuários. Existem três principais componentes envolvidos na correção de um exercício de programação (CHEANG et al., 2003):

- Correção: para qualquer dado de entrada válido, um programa é considerado correto se este produz os dados de saída desejados;
- Eficiência: um programa é eficiente se este executa suas tarefas sem consumir muito tempo de processamento e espaço em memória, definidos ou na descrição do problema ou por limitações de máquina;
- Manutenibilidade: um programa é manutenível se o seu código é de fácil entendimento, possui nomes de variáveis descritivas, comentários, identação adequada, etc...

Com essas definições, pode-se observar que o trabalho de correção de tarefas de programação torna-se difícil para os professores, tendo em vista que fatores como verificar se o programa está correto ou se o mesmo é eficiente deve ser feito para cada código fonte entregue pelos alunos. Para verificar se um programa é correto, é necessário, em muitos casos, abrir o código fonte um por um, verificando linha a linha onde os possíveis problemas podem ocorrer para poder fazer as recomendações de correção. É necessário também que sejam executados testes com conjuntos de dados a fim de verificar se a saída de um programa está de acordo com o esperado no exercício. A verificação da eficiência também se torna dispendiosa, pois é preciso, também, verificar em cada código fonte a estrutura de dados utilizada, qual algoritmo foi empregado na resolução do problema, resultando assim em uma verificação linha a linha do código (CHEANG et al., 2003).

De forma a solucionar parte deste problema da correção de programas de computador é que são utilizados sistemas de *juízes online*. Um sistema de *juiz online*

é, geralmente, um servidor que contém uma coleção de problemas variados bem como coleções de dados para validar um programa submetido a esse servidor. Um programa é considerado correto caso, para determinados dados de entrada, produza dados de saída esperados. Esses dados de entrada e saída estão armazenados no servidor do sistema de *juiz online*. Para verificar a eficiência do programa, normalmente é disponibilizado um certo limite de tempo e memória para o programa executar e produzir a saída esperada (CHEANG et al., 2003; REVILLA; MANZOOR; LIU, 2008; ZHIGANG et al., 2012). Um dos sistemas mais conhecidos é o sistema de *juiz online* desenvolvido pela Universidade de Valladolid, o *Juiz Online UVa*[1] (REVILLA; MANZOOR; LIU, 2008). Esse sistema de *juiz online* é utilizado pela *ACM* (*Association for Computing Machinery*) para validar códigos desenvolvidos por participantes da *ACM - ICPC* (*ACM International Collegiate Programming Contest*). Porém, não apenas participantes de competições da *ACM* podem submeter seus programas para validação, mas qualquer pessoa ao redor do mundo (REVILLA; MANZOOR; LIU, 2008).

Um exemplo de portal *online* de aprendizado de programação, que também possui um sistema de *juiz online* para validação de código, é o *Code Academy*[2]. Este portal tem como proposta o ensino de linguagens de programação e tecnologias voltadas ao desenvolvimento para a *web*. Uma vez selecionada pelo usuário a linguagem de programação que este deseja aprender, são disponibilizados exercícios e pequenos projetos com o intuito de desenvolver o conhecimento na tecnologia selecionada. É possível também, selecionar pequenos projetos que o site disponibiliza, com o objetivo de ensinar os conceitos aprendidos nas trilhas das tecnologias, em um projeto maior e com um grau de complexidade que exige um pouco mais do usuário.

Outro exemplo é o *Hacker Rank*[3]. Que é um site com um sistema de *juiz online*, porém mais voltado a competições de programação. Nesse site, há uma grande variedade de problemas das mais diversas áreas da ciência da computação, bem como problemas de programação envolvendo matemática. Os usuários podem resolver os problemas propostos utilizando mais de 30 linguagens de programação suportadas pelo site. O sistema de *juiz online* do *Hacker Rank* avalia o código fonte submetido pelo usuário, executando sobre eles uma bateria de testes com dados de entrada pré estabelecidos e verifica se as saídas dos programas são equivalentes às saídas esperadas. Há também um tempo limite para que o programa execute e produza a saída. Nesse site há também um sistema de classificação dos usuários, onde cada submissão correta e eficiente vale pontuação no sistema de classificação.

[1] https://uva.*online*judge.org/
[2] www.codecademy.com
[3] https://www.hackerrank.com/

2.3 Experiências com Portais de Ensino

A utilização de portais *online* de ensino exige de seus usuários, neste caso mais especificamente dos alunos, que possuam desejavelmente um elevado grau de independência, autonomia e proatividade. Porém, essa autonomia e proatividade são desejáveis não só apenas com relação ao conteúdo a ser aprendido, mas também em saber definir o que e qual conteúdo aprender ou que caminhos utilizar para aprendê-los (RICCIO, 2010).

Alunos que utilizam ambientes *online* de ensino contam com a oportunidade de estarem em um ambiente onde possam desenvolver ainda mais as práticas colaborativas, interativas e contam com um número muito maior de opções de mídias de conteúdo (ALVES; BRITO, 2005). Essa interatividade em um nível maior do que poderiam encontrar em uma sala de aula tradicional, possibilita que os alunos participem com mais intensidade, tornando-se não apenas receptores do conteúdo como também produtores (LÈVY, 1994).

Porém, ainda há uma quantidade de dificuldades encontradas no uso de ferramentas *online*, dificuldades essas por exigirem mudanças de entendimento na relação entre alunos e professores no que diz respeito ao ensino. Alunos precisam dispor de mais autonomia, autoria e um grau maior de colaboração, competências as quais nem sempre lhes são cobradas ou ensinadas em fases anteriores de sua vida escolar (ALVES; BRITO, 2005). Com relação aos professores, é preciso que modifiquem sua postura quanto às suas práticas presenciais, pois não basta apenas que mantenham sua postura instrucional. Outro fator também que pode trazer complicações é a dificuldade de interação com a tecnologia em geral que alguns professores possuem ou até mesmo a pressão das instituições no uso das ferramentas *online* (ALVES; BRITO, 2005).

2.4 O Portal de Algoritmos da UCS

A *UCS* possui o seu próprio ambiente de aprendizado *online* para o auxilio no ensino de algoritmos. A ferramenta, chamada *AlgoWeb*, é uma plataforma *web* composta de uma interface de edição e interpretação de código, em Português Estruturado, uma área com um conjunto de problemas algorítmicos, uma ferramenta para validação de soluções algorítmicas e uma interface para os professores (DORNELES; JUNIOR; ADAMI, 2010).

Uma gama de ambientes de desenvolvimentos já existiam à época em que o *AlgoWeb* foi desenvolvido. Esses ambientes possibilitavam a composição, edição e execução de código fonte feito utilizando a linguagem do Português Estruturado. Para citar alguns, o *Visualg* (SOUZA, 2009), o *webPortugol* (HOSTINS; RAABE, 2007) e o *Portugol IDE* (MANSO; OLIVEIRA; MARQUES, 2009). Apesar de úteis,

esses ambientes não disponibilizavam recursos para os professores disponibilizarem aos alunos um conjunto de problemas para serem resolvidos, problemas esses que contivessem um conjunto de dados de entrada e um conjunto de dados esperados na saída, de forma a validar a execução correta do algoritmo desenvolvido pelos alunos (DORNELES; JUNIOR; ADAMI, 2010).

De forma, então, a atender esta necessidade, foi desenvolvido o *AlgoWeb*, que vem sendo utilizado por diversos estudantes da *UCS* desde 2009 (DORNELES; JUNIOR; ADAMI, 2010).

O editor de código fonte do *AlgoWeb* possui funcionalidades e ferramentas de depuração existentes em outras *IDEs* (*Integrated Development Environments*) encontradas no mercado, como por exemplo, destaque do texto através de cores, mensagens de erro de compilação, possibilidade de executar o algoritmo linha a linha, adicionar *breakpoints* ou executar o código diretamente sem interrupções (DORNELES; JUNIOR; ADAMI, 2010).

O conjunto de problemas disponibilizados pelo *AlgoWeb*, ou o *"Portfólio de Problemas"*, contém uma gama variada de problemas que cobrem na totalidade a ementa de um curso de algoritmos. Os problemas são divididos em grupos de acordo com o tipo de estrutura de controle ou os dados que precisam para serem resolvidos. Uma vez selecionado o problema a ser resolvido, o ambiente exibe na tela uma descrição do problema, um exemplo de dados de entrada, quando necessário, e os respectivos dados de saída (DORNELES; JUNIOR; ADAMI, 2010).

O verificador de soluções é a parte do *AlgoWeb* no qual o usuário, ao finalizar de escrever seu código fonte, submete o mesmo como uma solução ao problema trabalhado, o qual é, então, analisado pelo interpretador do *AlgoWeb*. Essa fase de verificação é onde o interpretador verifica se o algoritmo produzido pelo usuário executa de forma correta e compara também se a execução está correta. Uma execução é considerada correta quando o código submetido pelo usuário à análise passa em uma bateria de casos de testes executada pelo *AlgoWeb*. Caso a solução proposta pelo usuário passe em todas as verificações, ela é considerada como uma solução válida (DORNELES; JUNIOR; ADAMI, 2010).

Por fim, o *AlgoWeb* conta com uma interface para o professor. Essa interface fornece uma ferramenta para que os professores possam cadastrar novos problemas, verificar as soluções propostas pelos alunos, ter um histórico de submissões por alunos e permite que possa editar problemas existentes (DORNELES; JUNIOR; ADAMI, 2010).

2.5 Estrutura Atual do Portal de Algoritmos

Este capítulo descreve a estrutura atual do Portal de Algoritmos. Neste capítulo será descrito tanto a estrutura do editor de códigos, a parte cliente, como o interpretador e a representação intermediária existente atualmente.

O Portal de Algoritmos atualmente é uma aplicação Java desenvolvida em *Java Applet*, com a finalidade de executar diretamente nos navegadores dos usuários. A implementação do editor de código está interligada com o *front-end* do interpretador, bem como o *front-end* está acoplado ao *back-end* do Portal de Algoritmos.

Existe uma representação intermediária extremamente simples desenvolvida atualmente no Portal. A mesma representação intermediária é responsável também por executar cada uma das instruções representadas. Ela também notifica o analisador sintático para modificar alguns estados internos do analisador, fazendo uma troca de mensagens entre as duas etapas do interpretador. Na Imagem 2.1 esta representado um esboço da estrutura atual do Portal de Algoritmos.

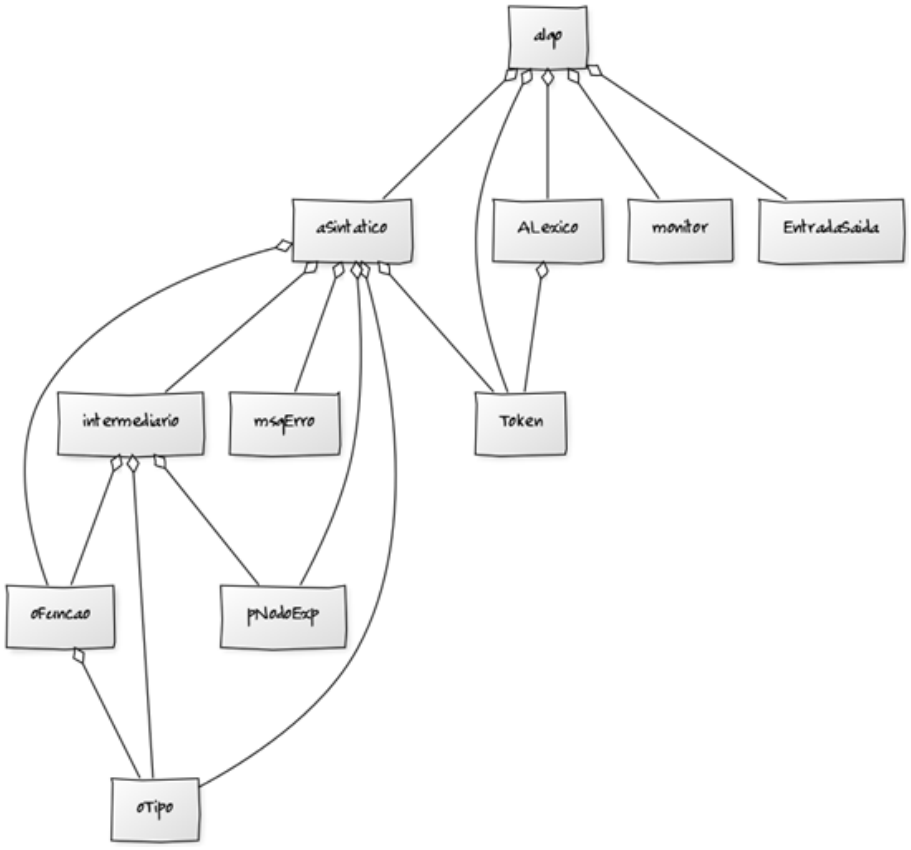

Figura 2.1: Esboço da estrutura atual do Portal de Algoritmos
Do próprio autor

2.5.1 Interface

A estrutura segue um padrão monolítico de *software*. Todas as fases e etapas da interpretação do código estão implementadas dentro de um único módulo. A classe principal do Portal é a classe *Algo.java*. Esta classe, que estende a classe *Applet*, define o objeto de interface da aplicação. Nela estão definidos os controles do editor de texto, as funcionalidades de depuração, a visualização das informações do problema que está sendo resolvido, informações dos valores armazenados nas variáveis em determinando momento da execução, etc. É através do objeto definido por essa classe que o usuário interage com o Portal de Algoritmos e desenvolve seu código fonte.

A execução do código fonte é iniciada pela classe *aSintatico.java*. O método *executar*, contido na classe *Algo.java* instancia um objeto do tipo *aSintatico* e passa por parâmetro, entre outras coisas (inclusive uma referência do próprio objeto *Algo*), o código fonte desenvolvido pelo usuário. Instanciado o objeto do tipo *aSintatico*, é chamado o método *algoritmo* deste objeto, que retorna um valor do tipo booleano indicando se a interpretação do código ocorreu até o final ou não.

2.5.2 Analisador léxico e sintático

Dentro desse método *algoritmo*, da classe *aSintatico*, é que ocorrem todas as etapas de reconhecimento das regras léxicas, sintáticas, semânticas e a tradução para o código intermediário interno do Portal. Nesse método, também, é interpretado o código e os resultados da execução são gerados. A classe *aSintatico* é quem, através de uma propriedade interna sua, um objeto do tipo *ALexico*, reconhece cada um dos *tokens* contidos na sequência de caracteres do código fonte. Cada *token* reconhecido é representado por um objeto do tipo *token.java*. Este objeto armazena informações, como por exemplo a linha dentro do código fonte onde o *token* está, o código deste *token*, seu lexema e sua posição dentro da sequência de caracteres. Após o reconhecimento de *tokens* válidos, são verificadas as regras sintáticas da linguagem e, ao mesmo tempo, executadas as produções referentes à tradução, da regra sintática reconhecida, em código intermediário.

2.5.3 Representação Intermediária

A representação intermediária atual do Portal de Algoritmos é definida pela classe *intermediario.java*. Essa representação intermediária é armazenada em memória, em um vetor interno da classe *aSintatico*. O objeto do tipo *intermediario* executa nove tipos de instruções diretamente, que são as instruções de atribuição, leitura, escrita, chamada de função ou procedimento, desvio condicional, desvio incondicional, retorno de procedimento, retorno de função e de fim de algoritmo. Na execução destas instruções, pode ser necessário acessar o valor de alguma variável,

obter o resultado de alguma operação (aritmética, lógica ou relacional) ou fazer uma chamada de função. Para tais, o objeto do tipo *intermediario* possui um objeto do tipo *pNodoExp.java*. Esse objeto, por sua vez, possui uma propriedade do tipo *nodoExp.java* a qual executa as operações (aritméticas, lógicas ou relacionais), faz a chamada de funções ou retorna valores de variáveis. Esta classe define, também, as funções pré definidas na linguagem, como por exemplo, função de seno, cosseno, tamanho de um literal, etc. A classe *nodoExp* também contém toda a lógica de tratamento de vetores e matrizes, como por exemplo o acesso a valores contidos no vetor ou matriz, inserção de valores no vetor ou matriz, e etc.

Há também o tratamento de erros encontrados, tanto na fase de análise léxica e sintática, bem como na fase de interpretação do código. As mensagens de erro geradas durante a análise léxica e sintática são enviadas diretamente para a interface do usuário, através do método *adicionaAviso*, da classe *Algo*. As mensagens de erros geradas pelo objeto do tipo *intermediario* são encapsuladas em um objeto do tipo *msgErro.java*. Essa classe possui apenas uma propriedade do tipo *String*, contendo a mensagem do erro encontrado, e uma propriedade do tipo booleano, cujo valor é usado para verificar se houve um erro ou não, o qual é usado para interromper a execução do código.

2.6 Compiladores e Interpretadores

Os programas de computadores, atualmente, são utilizados em larga escala por diversas áreas onde há atividade humana. Em diversos setores da economia, por exemplo, podemos encontrar ao menos um programa desenvolvido a fim de otimizar e automatizar alguma atividade, possibilitando que um trabalho desenvolvido por muitas pessoas possa ser facilmente exercido por apenas uma pessoa. Esses programas são desenvolvidos por humanos, utilizando uma linguagem de mais alto nível, diferente do tipo de linguagem que uma máquina consegue entender.

> Linguagens de programação são notações para descrever computações para pessoas e para máquinas. O mundo conforme o conhecemos depende de linguagens de programação, pois todo o *software* executado em todos os computadores foi escrito em alguma linguagem de programação. Porém, antes de ser executado, um programa primeiro precisa ser traduzido para um formato que lhe permita ser executado por um computador (AHO et al., 2008).

Essa tradução é feita normalmente por um outro programa de computador: ou por um compilador ou por um interpretador.

Um compilador é um programa que traduz um código escrito em uma linguagem de mais alto nível para uma de mais baixo nível, na forma de instruções ou coman-

dos mais específicos ao *hardware* que irá executar essas instruções. O compilador tem o objetivo de converter um programa fonte, escrito em uma linguagem mais aproximada à linguagem humana, em um programa alvo que contém instruções de máquina (COOPER; TORCZON, 2014; MAK, 1996), semelhante ao que ocorre na Figura 2.2.

Figura 2.2: Compilador
(COOPER; TORCZON, 2014)

O interpretador, por outro lado, é um programa que recebe como entrada um código escrito em linguagem de alto nível e traduz esse programa fonte em representações internas, chamadas de *código intermediário* e executa esse código interno, produzindo assim os resultados da execução deste código (MAK, 1996), semelhante ao que ocorre na Figura 2.3.

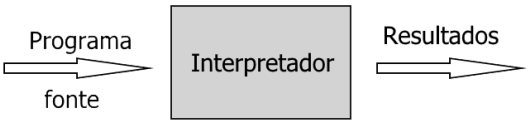

Figura 2.3: Interpretador
(COOPER; TORCZON, 2014)

A principal diferença entre um compilador e um interpretador está no fato de que o primeiro não executa o código fonte, ele apenas o traduz para um código alvo, o qual será posteriormente executado pelo computador, ao passo que um interpretador executa o programa fonte e produz resultados para essa execução (COOPER; TORCZON, 2014; MAK, 1996). O produto de um compilador é um programa alvo, que pode ser a linguagem de máquina específica de algum computador ou em linguagem de montagem. No caso do produto da compilação ser um programa em linguagem de montagem, este precisa passar ainda por um processo de montagem, a fim de ser traduzido para linguagem de máquina. Terminados esses passos de tradução executados por um compilador, um programa, normalmente chamado de *linker*, precisa ainda combinar esses arquivos em código alvo em um programa alvo que poderá, enfim, ser executado em um computador (MAK, 1996).

2.7 Português Estruturado

A escolha da linguagem de programação para um curso introdutório de algoritmos é de extrema importância para o sucesso do ensino da matéria. Como o principal objetivo nessas matérias introdutórias é o ensino do pensar estruturado e algorítmico, a linguagem não deve compor uma barreira extra à já complicada tarefa dos alunos. Por isso, uma linguagem que se assemelhe mais à linguagem falada pelos alunos brasileiros, o português, torna mais fácil a tarefa de aprender a construir algoritmos. O Português Estruturado, ou ainda o *Portugol*, nasceu com esses objetivos em mente. Deveria ser uma linguagem de fácil aprendizagem, que contivesse as estruturas necessárias para a construção de um raciocínio algorítmico e que provesse uma transição mais tranquila para outras linguagens de alto nível, como por exemplo o C, Java, C#, etc... (MANSO; OLIVEIRA; MARQUES, 2009).

A sintaxe do Português Estruturado assemelha-se à sintaxe do Pascal, porém sem o ponto-e-vírgula ao final dos comandos. A linguagem possui outras características semelhantes a linguagens de alto nível, como a declaração de variáveis quando preciso, definição de tipos, comandos de repetição e de desvio, entre outros. (SOUZA, 2009; MANSO; OLIVEIRA; MARQUES, 2009).

O Trecho de Código 2.1 contém um exemplo de código fonte escrito na linguagem do Português Estruturado.

Trecho de Código 2.1: Exemplo de pseudocódigo usando a sintaxe do Português Estruturado

```
 1 algoritmo "numeros_amigos"
 2 var  a,b,somaa, somab,i:inteiro
 3 inicio
 4     para a de 1 ate 10 faca
 5         somaa<-0
 6         para i de 1 ate a\2 faca
 7             se a%i=0 entao
 8                 somaa<-somaa+i
 9             fimse
10         fimpara
11         b<-somaa
12         somab<-0
13         para i de 1 ate b\2 faca
14             se b%i=0 entao
15                 somab<-somab+i
16             fimse
17         fimpara
18         se a=somab entao
19             escreval(a," e ",b," sao amigos")
20         fimse
21     fimpara
22 fimalgoritmo
```

2.8 Representação Intermediária

Os compiladores e interpretadores são softwares compostos de diversos módulos que atuam sobre um código fonte de entrada, conforme mostra a Figura 2.4. Para que cada módulo de um compilador ou de um interpretador possa operar eficientemente sobre o código inicial, é preciso que o compilador, por exemplo, retire algum conhecimento sobre o código fonte e armazene esse conhecimento em estruturas de dados internas ao compilador. Essa estrutura interna dos compiladores é chamada de representação intermediária (COOPER; TORCZON, 2014).

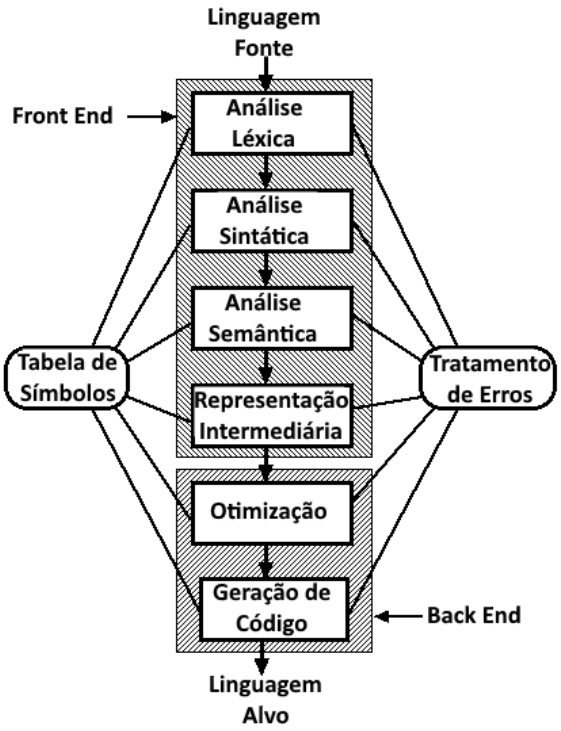

Figura 2.4: Etapas do *front-end* e do *back-end* de um compilador
Adaptada de: `http://blog.bbcoimbra.com/2011/09/18/introducao-aos-compiladores.html`

Como normalmente a maioria dos compiladores são compostos por uma parte de *front-end*, que analisa o código fonte, e por uma parte de *back-end*, que gera o código alvo, faz-se necessária a implementação de uma representação intermediária a fim de que não fique o compilador de uma linguagem atrelado à geração de código de máquina específica.

> Com uma representação intermediária definida adequadamente, um compilador para a linguagem i e a máquina j pode então ser construído, combinando-se o *front-end* para a linguagem i com o *back-end* para a máquina j. Essa abordagem para a criação de um conjunto de compiladores pode economizar muito esforço: m X n compiladores podem ser construídos escrevendo-se apenas m

front-ends e *n back-ends* (AHO et al., 2008).

Algumas das formas existentes de representação intermediária são (AHO et al., 2008):

- Representações intermediárias gráficas: algoritmos são representados internamente na forma de nós, arestas, árvores, etc... Por exemplo: árvores sintáticas, árvores sintáticas abstratas, grafos de fluxo de controle, grafos de dependência, grafo de chamada;
- Representações intermediárias lineares: o compilador representa internamente o código fonte na forma de um pseudocódigo de alguma máquina abstrata. Por exemplo: código de máquina de pilha, códigos de três endereços;
- Representações intermediárias híbridas: combinam elementos das representações gráficas e lineares a fim de se aproveitar dos pontos fortes de ambas as representações e minimizar os pontos fracos das mesmas.

2.9 Máquinas Virtuais

Os sistemas computacionais, apesar de sua complexidade, existem e continuam evoluindo devido ao fato de serem projetados com hierarquias contendo camadas bem definidas, que separam níveis diferentes de abstração (SMITH; NAIR, 2005).

Porém, mesmo sistemas com hierarquia bem definida também tem suas limitações. Subsistemas e dispositivos projetados para uma arquitetura específica podem não funcionar em outra arquitetura. Outro exemplo são programas distribuídos através de seus arquivos binários, ficando estes limitados a uma *ISA* (*Instruction Set Architecture*) específica e dependentes de uma interface de um sistema operacional (SMITH; NAIR, 2005).

A virtualização provê uma forma de contornar tais restrições:

> Virtualizar um sistema ou componente - como o processador, memória ou um dispositivo de E/S - em um certo nível de abstração mapeia sua interface a recursos visíveis na interface ou recursos de um sistema subjacente real, possivelmente diferente. (SMITH; NAIR, 2005).

As máquinas virtuais mais conhecidas vêm em duas categorias (SMITH; NAIR, 2005):

- Máquina Virtual de Sistema: são máquinas virtuais que simulam uma plataforma de sistema completo, podendo ser executado nessas máquinas sistemas operacionais inteiros. Usualmente, essas máquinas simulam uma arquitetura existente;
- Máquina Virtual de Processo: são máquinas virtuais projetadas para executa-

rem apenas um processo (programa). São geralmente relacionadas diretamente com alguma linguagem de programação, sendo desenvolvidas para proverem portabilidade e independência de hardware ou sistema operacional.

As máquinas virtuais inerentes ao escopo deste trabalho são as Máquinas Virtuais de Processo.

2.9.1 LLVM

O *LLVM* (*Low Level Virtual Machine*) é um projeto de infraestrutura de compilador projetado para ser um conjunto de bibliotecas reutilizáveis e com interfaces bem definidas. É um projeto que iniciou-se no ano de 2000, na Universidade de Illinois, Estados Unidos. O acrônimo de *LLVM*, que em português significa *máquina virtual de baixo nível*, remonta aos objetivos iniciais do projeto: de fornecer uma estratégia de compilação moderna, baseada em compilação no formato *SSA* (*Static Single Assignment form*), capaz de fornecer suporte tanto para compilação estática quanto dinâmica de linguagens arbitrárias de programação (LATTNER et al., 2010).

A representação intermediária da *LLVM* é a parte central desta máquina virtual. É uma representação de baixo nível, bem próximo à linguagem de montagem. A representação intermediária da *LLVM* possui instruções fortemente tipadas e que abstraem detalhes da máquina alvo. O Trecho de Código 2.2 demonstra um exemplo de uma função traduzida para a representação intermediária da *LLVM*.

Trecho de Código 2.2: Exemplo de código intermediário da *LLVM*

```
 1 ready> def foo(a b) a*a + 2*a*b + b*b;
 2 Read function definition:
 3 define double @foo(double %a, double %b) {
 4 entry:
 5 %multmp = fmul double %a, %a
 6 %multmp1 = fmul double 2.000000e+00, %a
 7 %multmp2 = fmul double %multmp1, %b
 8 %addtmp = fadd double %multmp, %multmp2
 9 %multmp3 = fmul double %b, %b
10 %addtmp4 = fadd double %addtmp, %multmp3
11 ret double %addtmp4
12 }
```

Retirado de: `llvm.org/docs/tutorial/LangImpl3.html`

A máquina virtual *LLVM* é baseada em registradores, porém, não fica limitada a um número fixo de registradores. A representação intermediária da *LLVM* utiliza um conjunto infinito de registradores através do uso do `%` em frente ao nome de um registrador interno ou nome de uma váriavel.

A *LLVM* possui uma vasta documentação, facilmente encontrada no site do

projeto[4], de suas instruções, ferramentas de geração de código, ferramentas de otimização, etc. A *LLVM* vem sendo bastante utilizada para o desenvolvimento de *front-ends* de linguagens de programação como o C, C++, para utilização em projetos de *software* livre, devido ao fato de que os compiladores para C, por exemplo, de código aberto estão estagnados há anos, não possuindo recursos de otimização de código, *retargeting* para código de máquina, entre outros[5]. Sendo assim, como a *LLVM* é uma coleção de bibliotecas de compilação, com uma representação intermediária, a *LLVM IR*, bem definida e projetada, está em grande expansão no uso em diferentes meios, não apenas no meio do *software* livre, mas também em meios acadêmicos, de pesquisas e também em alguns compiladores comerciais.

2.9.2 JVM

A *JVM* (*Java Virtual Machine*) é a peça fundamental da plataforma *Java*. É o componente de tecnologia responsável por tornar esta plataforma independente tanto de hardware quanto de sistema operacional, pelo tamanho reduzido do seu código compilado e por proteger os usuários de programas maliciosos (LINDHOLM et al., 2015).

A *JVM* não está associada diretamente à linguagem de programação Java. Não se deve confundir as duas, por não serem sinônimos. Ambas tem em comum o nome e um formato binário, o formato de arquivo *class*. É no arquivo *class* que estão contidas as instruções que operam na *JVM*, instruções essas conhecidas também como *bytecode*, e a tabela de símbolos, bem como outras informações auxiliares.

Por motivos de segurança, são impostos rígidos controles sintáticos e estruturais no código que é escrito em um arquivo *class*. Porém, qualquer linguagem de programação que possa ser representada nos termos válidos do arquivo *class* pode ser executada por uma *JVM* (LINDHOLM et al., 2015).

O conjunto de instruções da *JVM*, ou *bytecodes*, consiste de um *opcode* de um byte especificando a operação a ser executada, seguido de um ou mais operandos que proveem argumentos ou dados para a operação executada. Muitos dos *opcodes* definidos na *JVM* codificam informações sobre o tipo de dado na operação (LINDHOLM et al., 2015). O Trecho de Código 2.3 demonstra um pseudocódigo escrito em *bytecode* da *JVM*. O exemplo é a tradução de uma função que recebe por parâmetro dois valores do tipo inteiro, executa a soma dos mesmos e retorna o resultado da soma.

Trecho de Código 2.3: Exemplo de pseudocódigo intermediário da *JVM*

```
1 Method int addTwo(int,int)
2 0   iload_1        // Empilha o valor da variável local 1
```

[4] http://llvm.org/docs/

[5] http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf

```
3 1   iload_2         // Empilha o valor da variável locale 2
4 2   iadd            // Soma; empilhando o resultado na pilha de operandos
5 3   ireturn         // retorna o resultado do tipo inteiro
```

Retirado de: `http`:
`//docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html#jvms-3.1`

3 ARQUITETURA DO INTERPRETADOR

Este capítulo descreve a arquitetura do interpretador desenvolvido neste trabalho. Será abordado o módulo de *front-end*, composto pelo analisador léxico, analisador sintático e pelo gerador do código intermediário. Será tratado também sobre o *back-end*, que contém a máquina virtual que interpretará o código. Este capítulo trata apenas da organização do *front-end* e do *back-end* do interpretador do Portal de Algoritmos da *UCS*. A estrutura do *front-end* do lado cliente, isso é, do editor de código fonte, será tratada com mais detalhes no Capítulo 5.

O interpretador do Portal de Algoritmos *UCS* foi desenvolvido utilizando a linguagem de programação *Java* e estruturado de uma maneira a seguir o paradigma de programação orientado a objetos, diferentemente da grande maioria dos compiladores e interpretadores existentes, que são desenvolvidos baseados no paradigma procedural (CAMPBELL; IYER; AKBAL-DELIBAS, 2012). O interpretador é executado em um servidor, onde recebe, através do serviço *REST*, o código fonte escrito pelo usuário no editor de código, executando em um navegador. Esta comunicação entre o interpretador do Portal, que executa no servidor, com a interface do editor de códigos, executando no navegador do usuário, é detalhado mais adiante, no Capítulo 5.

Como convenção, fica definido que as classes criadas para os objetos da *AST* (*Abstract Syntax Tree*), o nome do arquivo de definição das regras sintáticas e léxicas e as demais classes auxiliares que venham a ser desenvolvidas, terão na sua nomenclatura a sigla *SP* no início. Essa sigla são as iniciais de Português Estruturado em inglês (*Structured Portuguese*).

3.1 Estrutura do *Front-End*

Esta seção trata do desenvolvimento do analisador léxico, sintático e semântico, bem como da tradução do código fonte de entrada na representação intermediária. É responsabilidade da primeira fase do interpretador, o *front-end*, realizar o entendimento do programa fonte e traduzir essa análise inicial em representação in-

termediária, que será utilizada nas fases seguintes do processo de interpretação do código fonte (COOPER; TORCZON, 2014).

O analisador léxico e sintático é a porta de entrada do *front-end* do interpretador. Ele recebe como entrada uma sequência de caracteres representando o código fonte escrito em linguagem do Português Estruturado e deve fazer a primeira análise sobre essa sequência de caracteres a fim de verificar se, nesta sequência de entrada de caracteres, as regras definidas na gramática da linguagem do Português Estruturado estão sendo atendidas ou não (COOPER; TORCZON, 2014; AHO et al., 2008).

O desenvolvimento desta primeira etapa dentro do *front-end* foi feito com o auxílio do software *JavaCC* (*the Java Compiler Compiler*). *JavaCC* é um software que gera analisadores léxicos com base em expressões regulares e analisadores sintáticos utilizando uma gramática para a linguagem pretendida (NORVELL, 2004; CAMPBELL; IYER; AKBAL-DELIBAS, 2012).

O *JavaCC* utiliza, na criação do analisador léxico e sintático, um arquivo contendo a definição de todos os *tokens* que devem ser identificados e também toda a estrutura sintática da gramática da linguagem utilizada, que neste trabalho será o Português Estruturado. Esta definição será especificada no arquivo *PortAlgParser.jj*. Neste arquivo estão contidos todos os *tokens* que deverão ser reconhecidos pelo analisador léxico, como por exemplo palavras reservadas do Português Estruturado, identificadores, valores numéricos e literais, *tokens* que devem ser descartados, como por exemplo, espaços em branco, comentários no código fonte, caracteres especiais de formatação de texto, etc. Junto neste arquivo, existem também as definições das regras sintáticas da linguagem.

A estrutura da declaração de um não-terminal, nessa especificação utilizada no *JavaCC*, é muito semelhante à declaração de um método feito na linguagem *Java*. Ela possui um nome, pode ter um tipo de retorno, pode ter uma lista de parâmetros e possui um bloco de código que contém as regras sintáticas e quaisquer outras ações que se queira executar em conjunto com a regra sintática.

Na declaração da especificação de um não-terminal é que estão incluídas as ações semânticas que geram a *AST*. Cada regra sintática, declarada dentro de um não-terminal, produz um nodo na *AST* correspondente ao comando ou expressão analisada. Essa *AST* gerada pelo analisador sintático é responsável, em seguida, pela análise semântica do código fonte de entrada, como por exemplo a verificação de tipos e a criação e atualização da tabela de símbolos com mais informações provenientes da análise da *AST*. Em seguida, a próxima tarefa da *AST* será a tradução para a representação intermediária escolhida para este trabalho, o *bytecode* da *JVM*.

Após toda a definição dos *tokens*, regras sintáticas e semânticas no arquivo de definição da linguagem, o *JavaCC*, então, cria diversos arquivos .java. Dentre esses arquivos gerados estão o *TokenManager.java*, que contém a especificação do anali-

sador léxico. Outro arquivo gerado é o *Parser.java*, arquivo esse responsável pela execução das ações sintáticas e semânticas, com o auxílio do *TokenManager.java*, provendo os *tokens* necessários para a execução. Na Tabela 3.1 estão listadas as classes geradas pelo *JavaCC* com base no arquivo de especificação.

Tabela 3.1: Classes geradas pelo *JavaCC*

Classe gerada	Definição
SimpleCharStream.java	Representação da sequência de caracteres
Token.java	Objeto contendo as propriedades de um *token*
SPTokenManager.java	Responsável por ler a sequência de caracteres e retirar os *tokens* definidos na especificação
SPParserConstants.java	Constantes representando cada um dos *tokens* definidos na especificação
TokenMgrError.java	Retorna uma mensagem detalhada de erro na análise léxica
ParseException.java	Exceção que é lançada quando de erro na análise sintática. Pode ser estendida e customizada
SPParser.java	Programa principal que executa a análise sintática e quaisquer outras ações definidas na especificação

3.1.1 Representação Intermediária

Durante o desenvolvimento deste trabalho, uma pesquisa e um estudo foram realizados sobre as diferentes formas de representações intermediárias e estruturas de dados utilizadas nas representações. Num primeiro momento foi cogitado o desenvolvimento de uma representação intermediária própria. Porém, após algumas avaliações, ficou constatado que este tipo de abordagem traria algumas desvantagens, como por exemplo:

- A representação intermediária desenvolvida poderia ficar muito dependente da linguagem (Português Estruturado);
- As instruções desenvolvidas poderiam não abranger uma gama maior de estruturas e operações que poderiam ser futuramente implementadas na linguagem compilada/interpretada;
- Tornaria complicada a tarefa de incrementar o Portal, caso necessitasse incluir mais uma linguagem de programação para ser interpretada.

Com base nessas limitações encontradas partiu-se para uma abordagem diferente: a pesquisa de representações intermediárias já existentes, de código (ou especificação) livre e que pudesse ser desenvolvido de forma a permitir que uma variedade de linguagens de programação pudessem ser compiladas e representadas através de suas instruções. Após pesquisas realizadas, duas máquinas virtuais se apresentaram

como fortes candidatas a serem utilizadas para o desenvolvimento neste trabalho.

3.1.1.1 Escolha para esse trabalho

Com base nas máquinas virtuais candidatas, anteriormente comentadas (*LLVM* e *JVM*), a definição de qual das duas máquinas virtuais e seus conjuntos de instruções será escolhido para o desenvolvimento do interpretador neste trabalho foi feita com base em alguns pontos, como:

- Conjunto de instruções que possibilite a representação de todos os comandos e operações do Português Estruturado;
- Facilidade na tradução do código fonte para a representação intermediária;
- Existência de documentação e especificação técnica suficientes para a implementação da máquina virtual;
- Instruções que possibilitem, num futuro, a expansão das capacidades de reconhecimento de linguagens de programação pelo Portal de Algoritmos.

Foi utilizado, também, para auxiliar a tomada de decisão, a Tabela comparativa 3.2 (adaptada de `http://docs.oracle.com/javase/specs/jvms/se8/html/index.html` e `http://llvm.org/docs/`), mostrando alguns aspectos levados em consideração na escolha da máquina virtual e, consequentemente, de suas instruções.

Tabela 3.2: Tabela comparativa entre *JVM* e *LLVM*

	JVM	LLVM
Documentação das instruções	Documentação abundante e bem escrita para suas instruções e bastantes exemplos de desenvolvimento tanto da máquina virtual quanto da representação.	Documentação bem detalhada de suas instruções, porém não tão abundante e com poucos exemplos de desenvolvimento de sua representação intermediária.
Conjunto de Instruções	Bastante amplo para representar os comandos do Português Estruturado.	Comandos e instruções de baixo nível, com foco maior em prover estruturas a facilitar a otimização do código. Representa bem os comandos do Português Estruturado, porém para algumas operações pode ser necessário mais código na tradução.
Tamanho do código gerado	Produz código intermediário de tamanho pequeno, devido à suas instruções serem representadas por um byte	Gera código reduzido após a tradução. Cada instrução ocupa palavras de 32 bits.
Modelo de máquina	Pilha	Registradores
Gerenciamento de Memória	Automático	Manual
Quantidade de Instruções	255 *opcodes* (incluído faixa de instruções não utilizadas, para futura expansão).	Somente 31 instruções. Isso é devido à maioria dos *opcodes* ser sobrecarregado e não haver repetições.

Com base nas comparações realizadas (demonstradas na Tabela 3.2), principalmente no que tange à quantidade e, acima de tudo, qualidade de documentação, decidiu-se por implementar o conjunto de instruções e a máquina virtual, para execução do código, da *JVM*. A *JVM* possui um manual técnico muito bem elaborado, tanto das instruções (*opcodes*) quanto de aspectos de implementação da própria máquina virtual que executará as instruções.

Um outro motivo para a escolha da *JVM* é que, ao compilar o código do Português Estruturado para *bytecode* no arquivo *class*, este arquivo pode ser executado por qualquer implementação da *JVM*. Por exemplo, o arquivo *class* pode ser executado diretamente pela *JVM* desenvolvida pela Oracle, empresa que desenvolve e mantém a linguagem de programação Java. Assim, as possibilidades são muitas para o código escrito em Português Estruturado, ficando o mesmo não limitado a executar

apenas na implementação do Portal, mas podendo ser portado para qualquer outra máquina virtual, sem dependência de hardware ou sistema operacional.

Outro fator motivador da escolha é que as instruções definidas para a *JVM* são simples e conseguem representar muito bem o conjunto de comandos, operadores e estruturas existentes no Português Estruturado. Como as instruções são de tamanho pequeno, ocupando um byte por instrução, o código gerado para a representação do código fonte tende a ser compacto, ocupando menos espaço de memória. Outro fator é que, por a representação intermediária da *JVM* ser rigorosamente definida e padronizada, o código gerado no arquivo *class*, produto da compilação do código em Português Estruturado, pode ser utilizado em outras disciplinas do curso de computação, como por exemplo na disciplina de compiladores, com o intuito do estudo das representações intermediárias, otimização de código e outros assuntos inerentes à disciplina.

3.1.2 Tradução para a Representação Intermediária

A tradução para a representação intermediária é feita através da *AST*. Cada objeto nodo da árvore é responsável por traduzir o seu comando, expressão ou estrutura em seu *bytecode* correspondente. O produto desta tradução é um arquivo no formato *class*, que contém o código compilado que é, então, executado pela máquina virtual do *back-end*. Para auxiliar na tradução da *AST* para a representação intermediária e não sobrecarregar de responsabilidade os objetos dos nodos da *AST*, foi desenvolvida uma classe *SPCLEmitter*.

Essa classe *SPCLEmitter* conhece a estrutura do arquivo do tipo *class* e, através de parâmetros recebidos dos objetos nodos da *AST*, gera os *bytecodes* correspondentes. Essa classe grava, também, em disco, o arquivo que ela tem armazenado em memória, para o uso do *back-end*. Para cada uma das estruturas que compõem o arquivo *class*, há um método, ou um conjunto de métodos, responsáveis por armazenar os dados e informações, provenientes da *AST*, no arquivo *class*.

3.1.2.1 Estrutura do arquivo class

Essa subseção descreve a estrutura de um arquivo do tipo *class*, produto da tradução de uma *AST* em *bytecodes*. Toda a definição completa da estrutura do arquivo *class* pode ser encontrada na especificação da *Java SE 8*[1] com grande riqueza de detalhes. Serão apresentadas aqui somente as partes e estruturas mais importantes para o entendimento do arquivo do tipo *class*.

Um arquivo *class* é uma sequência de *bytes*. Instruções que necessitam de quantidades de 16 bits, 32 bits e 64 bits são construídas lendo 2, 4 e 8 *bytes* consecutivos. Se fosse descrito na linguagem C, o arquivo *class* teria a estrutura semelhante ao

[1] `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html`

que é mostrado no Trecho de Código 3.1, conforme (LINDHOLM et al., 2015).

Trecho de Código 3.1: Estrutura do arquivo *class*

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Os tipos *u2* e *u4* representam quantidades de dois e quatro *bytes*, respectivamente. A Tabela 3.3 descreve cada um dos atributos contidos na estrutura do arquivo *class*.

Tabela 3.3: Items contidos na estrutura do arquivo *class*
(CAMPBELL; IYER; AKBAL-DELIBAS, 2012)

Atributo	Descrição
magic	Um número (0xCAFEBABE) identificando o formato de arquivo *class*.
`minor_version`, `major_version`	Juntos, o número de versão maior e menor determinam a versão do arquivo *class*.
`constant_pool_count`	Número de entradas na tabela `constant_pool` mais um.
`constant_pool[]`	Uma tabela de estruturas que representam várias constantes *string*, nomes de classes e interfaces, nomes de propriedades e outras constantes que são referenciadas na estrutura do arquivo *class* e suas sub-estruturas.
`access_flags`	Máscara de sinalizadores usados para denotar permissões de acesso e propriedades desta classe ou interface.
`this_class`	Deve ser um índice válido na tabela `constant_pool`. A entrada neste índice deve ser uma estrutura representando a classe ou interface definidas por este arquivo *class*.
`super_class`	Deve ser um índice válido na tabela `constant_pool`. A entrada neste índice deve ser uma estrutura representando a super classe direta da classe ou interface definidas por este arquivo *class*.
`interfaces_count`	O número de super interfaces diretas da classe ou interface definida pelo arquivo *class*.
interfaces[]	Cada valor na tabela deve ser um índice válido na tabela `constant_pool`. A entrada em cada índice deve ser uma estrutura representando uma interface que é uma super interface direta da classe ou interface definida por este arquivo *class*.
`fields_count`	Número de entradas na tabela de propriedades.
fields[]	Cada valor na tabela deve ser uma estrutura `field_info` contendo a descriçao completa de um campo na classe ou interface definida neste arquivo *class*.
`methods_count`	Número de entradas na tabela de métodos.
methods[]	Cada valor na tabela deve ser uma estrutura `method_info` contendo a descrição completa do método na classe ou interface definida neste arquivo *class*.
`attributes_count`	Número de entradas na tabela de atributos.
attributes[]	Deve ser uma tabela dos atributos do arquivo *class*

3.2 Estrutura do *Back-End*

O *back-end* do interpretador será composto de uma ferramenta de depuração de código, para a execução do código gerado e armazenado em um arquivo *.class*, traduzido pelo *front-end*. A ferramenta de depuração desenvolvida neste trabalho segue os padrões e definições da *JDI* (*Java Debug Interface*)[2].

A escolha de desenvolver uma ferramenta de depuração utilizando o *framework* da *JDI*, em detrimento de usar tantas outras já existentes, se deve ao fato do portal necessitar da possibilidade de ferramentas de *debbuging*. Como a execução do código fonte desenvolvido exige que seja possível a adição de pontos de parada no meio do código (*breakpoints*), execução passo-a-passo das linhas de código e visualização, em tempo de execução, dos valores armazenados nas variáveis existentes no momento da execução, faz-se necessário um controle muito maior da execução do código do que simplesmente relegar a execução a uma *JVM* já pronta e apenas verificar os resultados produzidos pela execução do código, a fim de verificar a produção correta de dados de saída.

[2] `http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html`

4 MAPEAMENTO DAS INSTRUÇÕES

Neste capítulo será apresentado o mapeamento dos comandos e instruções do Português Estruturado para a representação intermediária de *bytecode*. Em cada seção que tiver um *opcode* específico para a instrução em Português Estruturado, será apresentada uma tabela contendo um mnemônico para o *opcode* representado e o código do *opcode* em decimal e hexadecimal (entre parênteses).

Cada seção de comandos apresenta, também, uma tabela contendo uma comparação entre um código escrito em Português Estruturado e o seu equivalente traduzido para *bytecode*. A coluna do código correspondente em *bytecode* contém um índice da posição do *opcode* dentro do vetor que contém os bytes do código *bytecode* para esse método. Esse índice pode ser visto alternativamente como o deslocamento em bytes do início do método até o índice do *opcode*.

4.1 Tipos de Dados

Nessa seção serão demonstradas as traduções dos tipos suportados pelo Português Estruturado para *bytecode*. No Português Estruturado são suportados os seguintes tipos: *inteiro*, *real*, *literal* e *logico*. Há, também, suporte para a declaração de vetores de uma dimensão ou duas dimensões. A Tabela 4.1 apresenta os tipos existentes no Português Estruturado e suas respectivas faixas de valores.

Tabela 4.1: Tipos e seus valores no Português Estruturado

Tipo	Valores
inteiro	valores inteiros de 64 bits com sinal
real	valores reais com uma precisão de 7 dígitos
literal	armazena uma sequência de caracteres
logico	armazena um valor lógico, verdadeiro ou falso
vetor	cria um vetor de uma ou duas dimensões

Os tipos *inteiro* e *real* possuem representação direta na forma *bytecode*. Os valores do tipo *inteiro* são traduzidos para valores do tipo *long* em *bytecode*. Já os

valores do tipo *real* são traduzidos para o tipo *float* do *bytecode*. Para os tipos *literal* e *logico* foi necessário fazer uma modificação no tipo armazenado.

O tipo *literal*, no Português Estruturado, representa uma sequência de caracteres. Por não existir uma representação direta de sequência de caracteres em *bytecode*, esse tipo foi traduzido de forma a utilizar a estrutura da classe *String* da linguagem Java.

O tipo *logico*, no Português Estruturado, representa os valores lógicos *verdadeiro* ou *falso*. Como há um suporte limitado para valores do tipo lógico em *bytecode*, optou-se nesse trabalho por traduzir valores lógicos do Português Estruturado para valores do tipo *int* do *bytecode*. Uma vez que os *opcodes* de desvios condicionais verificam valores do tipo *int*, optou-se, assim, por fazer essa tradução dos valores *logicos*. O valor lógico *verdadeiro* é traduzido para o valor *inteiro* **1**, já o valor lógico *falso* é traduzido para o valor *inteiro* **0**.

Na Tabela 4.2 são representadas as traduções dos tipos do Português Estruturado para *bytecode*.

Tabela 4.2: Tradução dos tipos do Português Estruturado para *bytecode*

Português Estruturado	Bytecode
inteiro	*long*
real	*float*
literal	*reference*
logico	*int*
vetor	*reference*

Pelo fato de a representação do tipo inteiro do Português Estruturado ser mapeado para o tipo *long* da *JVM*, muitas instruções de operações precisam fazer conversão de tipo nas constantes envolvidas. Outros operadores produzem somente resultados em um determinado tipo, o que implica na conversão, também, do resultado da operação. A Tabela 4.3 mostra os principais *opcodes* utilizados para conversão de tipo. O *T* presente na coluna *opcode* é um indicador de tipo para o qual o tipo base está sendo convertido.

Tabela 4.3: Instruções para conversão de tipo

Opcode	int	long	float
i2T		i2l	i2f
l2T	l2i		l2f
f2T	f2i	f2l	

Para a declaração de vetores de uma dimensão é utilizado o *opcode* **newarray** seguido do *opcode* referente ao tipo do vetor. Este *opcode* desempilha da pilha de

operandos um valor, do tipo inteiro, representando o tamanho do novo vetor e, então, aloca um novo objeto vetor do tamanho definido. Em seguida, é empilhada na pilha de operandos uma referência, um valor do tipo **reference**, para o novo vetor. A Tabela 4.4 demonstra a declaração de um vetor.

Para declarar uma matriz, que em *bytecode* é representada como um vetor de duas dimensões, é utilizado o *opcode* **multianewarray** seguido de um valor representando o tipo de vetor de duas dimensões que será criado e um número de dimensões do vetor. A Tabela 4.5 demonstra a declaração de uma matriz (vetor de duas dimensões).

A declaração de matrizes e vetores em Português Estruturado permite que sejam definidos os valores dos limites iniciais e finais. Por exemplo, um vetor pode ser declarado da seguinte forma *v : vetor[3..10] de inteiro*, onde o número 3 indica que o primeiro valor armazenado no vetor *v* está indexado na posição 3 e o último valor armazenado em *v* está na posição 10. Como em *bytecode* os índices de vetores e matrizes possuem seu limite inferior em *zero*, as informações de limites inferiores e superiores na declaração de vetores e matrizes do Português Estruturado são armazenadas na tabela de símbolos do item declarado. Assim, na hora de traduzir o código fonte para a representação intermediária *bytecode*, são feitas as traduções dos valores conforme os valores dos mínimos e máximos declarados para o formato de índice zero do *bytecode*.

Tabela 4.4: Declaração de um vetor

Português Estruturado		Bytecode
Var	0	`iconst_4`
v : vetor[1..4] de inteiro	1	newarray 11
	3	astore 1

Tabela 4.5: Declaração de uma matriz

Português Estruturado		Bytecode
Var	0	`iconst_4`
m : vetor[1..4, 1..4] de real	1	`iconst_4`
	2	multianewarray 1 2
	6	astore 1

A Tabela 4.6 mostra a estrutura do acesso a elementos de vetores traduzido para *bytecode*. A Tabela 4.7 mostra a estrutura do acesso a elementos de matrizes traduzido para *bytecode*.

Tabela 4.6: Estrutura do acesso a elementos de vetores em *bytecode*

Carrega na pilha de operandos referência do vetor em memória
carrega na pilha de operandos índice acessado e valor do limite mínimo do vetor
Subtrai índice acessado pelo valor do limite mínimo para obter índice com limite inferior zero
laload *Opcode* para carregar um elemento do tipo *long* de um vetor

Tabela 4.7: Estrutura do acesso a elementos de matrizes em *bytecode*

Carrega na pilha de operandos referência da matriz em memória
carrega na pilha de operandos índice da linha acessada e valor do limite mínimo da linha da matriz
Subtrai índice da linha acessada pelo valor do limite mínimo da linha da matriz para obter índice com limite inferior zero
aaload *Opcode* para carregar o sub-vetor da próxima dimensão (colunas) da matriz
carrega na pilha de operandos índice da coluna acessada e valor do limite mínimo da coluna da matriz
Subtrai índice da coluna acessada pelo valor do limite mínimo da coluna da matriz para obter índice com limite inferior zero
laload *Opcode* para carregar um elemento do tipo *long* de uma matriz

A Tabela 4.8 mostra um exemplo de acesso a um elemento de um vetor, *b*, cujo limite mínimo foi definido como o valor inteiro 3. O segundo exemplo listado na Tabela 4.8 mostra um exemplo de acesso a um elemento de uma matriz.

Para a declaração de vetores, são utilizados, em *bytecode*, *opcodes* específicos para especificar o tipo dos elementos armazenados no vetor. A Tabela 4.9 apresenta os

Tabela 4.8: Exemplos de acesso a elementos de um vetor e de uma matriz

Português Estruturado	Bytecode	
a <- b[6] + 5	0	aload 2
	2	bipush 6
	4	i2l
	5	`iconst_3`
	6	i2l
	7	lsub
	8	l2i
	9	laload
	10	`iconst_5`
	11	i2l
	12	ladd
	13	lstore 1
a <- b[6][4] + 5	0	aload 2
	2	bipush 6
	4	i2l
	5	`iconst_3`
	6	lsub
	7	l2i
	8	aaload
	9	`iconst_4`
	10	i2l
	11	`iconst_2`
	12	i2l
	13	lsub
	14	l2i
	15	laload
	16	`iconst_5`
	17	i2l
	18	ladd
	19	lstore 1

opcodes utilizados em conjunto do *opcode* de declaração de vetor, para definir o tipo dos elementos do vetor.

Tabela 4.9: Tipos para declaração de vetores em *bytecode*

Português Estruturado	Bytecode
inteiro	11
real	6
literal	5
logico	10

Muitas das instruções de *opcode* carregam consigo informações de tipo de operando na qual são executadas suas operações[1]. Por exemplo, o *opcode* **iadd** executa uma operação de soma sobre dois operandos, porém, os mesmos devem ser ambos do tipo *int*. O *opcode* **fload** carrega o valor de uma variável local para a pilha de operandos, porém, essa variável local deve ser do tipo *float*.

Para a maioria das instruções com tipo, a informação do tipo é informada explicitamente no início do *opcode*, na forma de uma letra. Ao longo deste capítulo, nas demonstrações de tradução de código em Português Estruturado para *bytecode*, serão utilizados muitos desses *opcodes* com tipo. A Tabela 4.10 apresenta os tipos utilizados nesse trabalho e seus respectivos caracteres indicativos.

Tabela 4.10: Tipos e suas representações em letras

Tipo	Letra
int	i
float	f
char	c
reference	a
long	l

4.2 Comandos

Nessa seção serão listados os códigos em *opcode* e sua correspondência com o Português Estruturado.

4.2.1 Comando de atribuição

Os *opcodes* listados na Tabela 4.11 executam a atribuição de um valor, do tipo correspondente, do topo da pilha de operandos para uma variável especificada através de um valor numérico, do tipo *inteiro*, que vem logo em seguida do *opcode*. Os comandos *iastore, fastore e castore* são utilizados especificamente para armazenamento de valores em elementos de vetores. Para esses comandos, ao invés da posição da variável que receberá o valor no topo da pilha de operandos, deve estar na pilha de operandos o valor da referência para o vetor, o valor numérico, do tipo *inteiro*, correspondente ao índice em que o valor deve ser armazenado no vetor e o próprio valor, do tipo correspondente, que será armazenado na posição do vetor.

[1] `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.11.1`

Tabela 4.11: Opcodes para atribuição

Mnemônico	Opcode
istore	*54 (0x36)*
lstore	*55 (0x37)*
fstore	*56 (0x38)*
iastore	*79 (0x4f)*
lastore	*80 (0x50)*
fastore	*81 (0x51)*
castore	*85 (0x55)*

Na Tabela 4.12 são demonstrados exemplos de códigos de atribuição em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.12: Exemplos de atribuição

Português Estruturado	Bytecode	
a <- 8	0	ldc 1
	2	lstore 1
a[1] <- 0	0	aload 1
	2	`lconst_1`
	3	l2i
	4	`lconst_0`
	5	lastore
b <- a[5]	0	aload 1
	2	`iconst_5`
	3	i2l
	4	l2i
	5	laload
	6	lstore 2

4.2.2 Expressões

Nesta subseção serão listados os *opcodes* referentes a expressões aritméticas, literais, relacionais e lógicas, bem como seus correspondentes com o Português Estruturado.

4.2.2.1 Expressões Aritméticas

Serão demonstrados nesta subseção os *opcodes* referentes a operadores de expressões aritméticas, como por exemplo, soma, subtração, troca de sinal, multiplicação, divisão, quociente da divisão e resto da divisão. As expressões aritméticas também são representadas como *opcodes* devido à natureza da máquina virtual utilizada neste

trabalho ser em formato de pilha. Os *opcodes* das expressões aritméticas trabalham sobre a pilha de operandos, desempilhando os valores necessários para sua execução e empilhando os valores resultantes da operação executada.

4.2.2.1.1 Troca de Sinal

Os *opcodes* listados na Tabela 4.13 executam a operação de troca de sinal sobre um valor. A instrução desempilha um valor, do tipo correspondente, do topo da pilha de operandos e executa *-valor*. Em seguida, o resultado da operação é empilhado novamente na pilha de operandos.

Tabela 4.13: Opcodes para troca de sinal

Mnemônico	Opcode
lneg	*117 (0x75)*
fneg	*118 (0x76)*

Na Tabela 4.14 é demonstrado um exemplo de código de troca de sinal em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.14: Exemplo de troca de sinal

Português Estruturado	Bytecode
a <- -b	0 lload 2
	2 lneg
	3 lstore 1

4.2.2.1.2 Multiplicação

Na Tabela 4.15 estão listados os *opcodes* utilizados para efetuar a multiplicação de dois valores. Estes *opcodes* desempilham dois valores do topo da pilha de operandos, do tipo correspondente, e o valor empilhado novamente na pilha de operandos, após a execução da instrução, é o resultado de *valor1 * valor2*.

Tabela 4.15: Opcodes de multiplicação

Mnemônico	Opcode
lmul	*105 (0x69)*
fmul	*106 (0x6a)*

Na Tabela 4.16 é demonstrado um exemplo de código de multiplicação em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.16: Exemplo de multiplicação

Português Estruturado	Bytecode	
a <- a * 4	0	lload 1
	2	`iconst_4`
	3	i2l
	4	lmul
	5	lstore 1

4.2.2.1.3 Divisão

O *opcode* listado na Tabela 4.17 executa a divisão de dois valores. Essa instrução desempilha dois valores, que devem ser do tipo *float*, do topo da pilha de operandos e o resultado da expressão é *valor1 / valor2*. Esse resultado é, então, empilhado novamente no topo da pilha de operandos. A operação de divisão do Português Estruturado sempre produz como resultado um valor do tipo *real*, por esse motivo, o *opcode* selecionado para traduzir esse operador é o da divisão de valores do tipo *float*. Caso um dos operandos da divisão, ou ambos, não seja do tipo *float*, uma conversão para o tipo *float* deve ser feita.

Tabela 4.17: Opcode de divisão

Mnemônico	Opcode
fdiv	*110 (0x6e)*

Na Tabela 4.18 é mostrado um exemplo de código de divisão em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.18: Exemplo de divisão

Português Estruturado	Bytecode	
a <- a / b	0	fload 1
	2	lload 2
	4	l2f
	5	fdiv
	6	fstore 1

4.2.2.1.4 Resto da Divisão

A Tabela 4.19 lista o *opcode* utilizado para obter o valor do resto da divisão. Esse comando, em Português Estruturado, é executado apenas para valores do tipo *inteiro*. Este *opcode* desempilha dois valores do topo da pilha de operandos, ambos

do tipo *long*, e o resultado da expressão é *value1 - (value1 / value2) * value2*. Esse resultado é, em seguida, empilhado novamente no topo da pilha de operandos.

Tabela 4.19: Opcode de resto da divisão

Mnemônico	Opcode
lrem	*113 (0x71)*

Na Tabela 4.20 é demonstrado um exemplo de código de resto da divisão em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.20: Exemplo de resto da divisão

Português Estruturado	Bytecode
a <- a % 3	0 lload 1
	2 `iconst_3`
	3 i2l
	4 lrem
	5 lstore 1

4.2.2.1.5 Quociente da Divisão

Para este operador do Português Estruturado, é utilizado o *opcode* listado na Tabela 4.21, o qual opera uma divisão sobre dois valores inteiros e empilha o resultado na pilha de operandos.

Tabela 4.21: Opcode de quociente da divisão

Mnemônico	Opcode
ldiv	*109 (0x6d)*

4.2.2.1.6 Soma

Os *opcodes* listados na Tabela 4.22 executam a operação de soma de dois valores. Estes *opcodes* desempilham dois valores, do tipo correspondente, do topo da pilha de operandos e o resultado da expressão é *valor1 + valor2*. Esse resultado é, então, empilhado novamente no topo da pilha de operandos.

Tabela 4.22: Opcodes de soma

Mnemônico	Opcode
ladd	*97 (0x61)*
fadd	*98 (0x62)*

Na Tabela 4.23 é demonstrado um exemplo de código de soma em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.23: Exemplo de soma

Português Estruturado	Bytecode
a <- a + b	0 lload 1
	2 lload 2
	4 ladd
	5 lstore 1

4.2.2.1.7 Subtração

Na Tabela 4.24 estão listados os *opcodes* utilizados para a operação de subtração de dois valores. Esses *opcodes* desempilham dois valores do topo da pilha de operandos e o resultado da expressão é *valor1 - valor2*. Esse resultado é, então, empilhado na pilha de operandos.

Tabela 4.24: Opcodes de subtração

Mnemônico	Opcode
lsub	*101 (0x65)*
fsub	*102 (0x66)*

Na Tabela 4.25 é mostrado um exemplo de código de subtração em Português Estruturado e sua tradução para *bytecode*.

Tabela 4.25: Exemplo de subtração

Português Estruturado	Bytecode	
a <- a - 2	0	lload 1
	2	`iconst_2`
	3	i2l
	4	lsub
	5	lstore 1

4.2.2.2 Expressões Literais

O único operador sobre literais existente no Português Estruturado do Portal de Algoritmos da UCS é o operador de concatenação de literais. Esse operador traduzido para *bytecode* torna-se, assim, uma operação de concatenação de *Strings* existente na linguagem Java, visto que o tipo literal, quando traduzido para *bytecode*, torna-se uma estrutura do tipo *String*.

4.2.2.3 Operadores Relacionais

Nessa seção será demonstrada a tradução dos operadores relacionais, como o maior (>), menor (<), maior ou igual (>=), menor ou igual (<=), igual (=) e diferente (<>), do Português Estruturado para *bytecode*.

As expressões que envolverem operadores relacionais são traduzidas para *bytecodes* através de dois passos. Primeiramente, utilizando os *opcodes* listados na Tabela 4.26, é feita a comparação entre os dois valores. Em seguida, utilizando os *opcodes* listados na Tabela 4.27, é feito um desvio condicional com base no valor armazenado no topo da tabela de operandos após a comparação entre os valores.

Os *opcodes* que executam o primeiro passo em uma expressão relacional, listados na Tabela 4.26, desempilham dois valores da pilha de operandos, que devem ser do tipo correspondente, e executam a comparação entre os mesmos. Caso *valor1 = valor2*, é empilhado o valor **0** na pilha de operandos. Se o *valor1 > valor2*, então é empilhado o valor **1**. No caso de *valor1 < valor2*, é empilhado, então, o valor **-1**. Estes *opcodes* relacionados na Tabela 4.26, são os mesmos para todos os operadores relacionais.

Tabela 4.26: Opcodes para expressões relacionais

Mnemônico	Opcode
lcmp	*148 (0x94)*
fcmpg	*150 (0x96)*

A segunda etapa da avaliação dos operadores relacionais é feita utilizando os

opcodes listados na Tabela 4.27 e, diferentemente do que ocorria no primeiro passo, são diferentes para cada um dos operadores relacionais. Todas as instruções relacionadas na Tabela 4.27 desempilham um valor no topo da pilha de operandos e executam um desvio condicional caso o valor seja **igual** a **zero** (*ifeq*), **diferente** de **zero** (*ifne*), **menor** que **zero** (*iflt*), **maior ou igual** a **zero** (*ifge*), **maior** que **zero** (*ifgt*) ou **menor ou igual** a **zero** (*ifle*). Após a execução da instrução de desvio será empilhado na pilha de operandos o valor referente a *verdadeiro* (1) ou falso (0).

Tabela 4.27: Opcodes de desvio condicional utilizados nos operadores relacionais

Mnemônico	Opcode
ifeq	*153 (0x99)*
ifne	*154 (0x9a)*
iflt	*155 (0x9b)*
ifge	*156 (0x9c)*
ifgt	*157 (0x9d)*
ifle	*158 (0x9e)*

Com base nisso, pode-se observar como as instruções listadas nas Tabelas 4.26 e 4.27 são utilizados em conjunto para executar expressões relacionais. Por exemplo, considerando que os dois valores a serem comparados já estejam no topo da pilha de operandos, neste exemplo serão dois valores do tipo inteiro, e queremos executar uma operação de comparação para saber se o *valor1* é menor (<) do que o *valor2*. Primeiramente é executada a instrução de comparação de valores inteiros, que é a *lcmp*. Esta instrução irá desempilhar os dois valores do topo da pilha de operandos e irá empilhar o resultado da comparação (1, 0 ou -1). Logo em seguida é executado o *opcode* de desvio condicional para o operador menor (<), que é a instrução *iflt*. Esse *opcode* desempilha um valor do topo da pilha de operandos e verifica se o mesmo é menor que zero, que é exatamente o que o teste de comparação de valores empilha na pilha de operandos caso o *valor1* for menor que o *valor2*. Caso o valor que foi desempilhado seja menor que zero, é executado um desvio para a posição que contenha uma instrução que empilhe na pilha de operandos o valor *verdadeiro*, que é a instrução `iconst_1`, a qual empilha o valor numérico 1. Caso contrário, a operação que executa deve empilhar o valor *falso*, o *bytecode* `iconst_0`, o qual empilha o valor numérico 0.

Na Tabela 4.28 é mostrado a estrutura da tradução em *bytecodes* da operação de comparação de igualdade entre valores. As outras operações relacionais seguem o mesmo padrão, mudando apenas a instrução usada para a operação (a linha contendo o *bytecode lcmp* na Tabela 4.28).

Tabela 4.28: Estrutura do operador de igualdade em *bytecode*

	Carrega operandos na pilha de operandos
	lcmp
	ifeq [**verdadeiro**]
falso:	Instruções caso comparação seja falsa
	Desvio para o fim do comando
verdadeiro:	Instruções caso comparação seja verdadeira
fimse:	Bloco executado em seguida

Na Tabela 4.29 são demonstrados exemplos de códigos de comparações de valores em Português Estruturado e sua tradução para *bytecodes*.

Tabela 4.29: Exemplo de comparações de valores

Português Estruturado	Bytecode	
a <- b = 4	0	lload 2
	1	`iconst_4`
	2	i2l
	3	lcmp
	4	ifeq 11
	7	`iconst_0`
	8	goto 12
	11	`iconst_1`
	12	istore 1
a <- b <> 5.6	0	lload 2
	1	l2f
	2	`ldc2_w #4`
	3	fcmpg
	4	ifne 11
	7	`iconst_0`
	8	goto 12
	11	`iconst_1`
	12	istore 1
a <- b > 4	0	lload 2
	1	`iconst_4`
	2	i2l
	3	lcmp
	4	ifgt 11
	7	`iconst_0`
	8	goto 12
	11	`iconst_1`
	12	istore 1
a <- b < 5.6	0	lload 2
	1	l2f
	2	`ldc2_w #4`
	3	fcmpg
	4	iflt 11
	7	`iconst_0`
	8	goto 12
	11	`iconst_1`
	12	istore 1

4.2.2.4 Expressões Lógicas

Esta seção trata da tradução dos operadores de expressões lógicas do Português Estruturado (*e*, *ou* e *nao*) para seus equivalentes em *bytecode*. Neste trabalho, foi implementada a avaliação parcial, ou código em "curto-circuito"(AHO et al., 2008),

de cada lado da expressão, no caso dos operadores *e* e *ou*, a fim de avaliar o resultado da expressão, o qual empilhará o valor correspondente através de um desvio feito na avaliação de cada um dos lados da expressão lógica avaliada.

4.2.2.4.1 Operador Lógico E

Na Tabela 4.30 é mostrado um exemplo de código da expressão E em Português Estruturado e sua tradução para *bytecode*, utilizando avaliação parcial e desvios para empilhar o resultado da avaliação da expressão. Caso o primeiro lado da expressão seja falso, a execução sofre um desvio para o índice que contém o *opcode* que empilha o valor correspondente ao valor falso na pilha de operandos. Da mesma forma, no caso de o primeiro valor ser verdadeiro, é avaliado, então, o segundo lado da expressão e em caso do resultado da avaliação ser verdadeiro, é empilhado o valor correspondente, caso contrário, o valor que corresponde ao valor falso é empilhado na pilha de operandos.

Tabela 4.30: Exemplo da expressão *e*

Português Estruturado	Bytecode	
a <- b E c	0	iload 2
	2	ifeq 14
	5	iload 3
	7	ifeq 14
	10	`iconst_1`
	11	goto 15
	14	`iconst_0`
	15	istore 1

4.2.2.4.2 Operador Lógico OU

Na Tabela 4.31 é mostrado um exemplo de código da expressão *ou* em Português Estruturado e sua tradução para *bytecode*. Através da avaliação de cada um dos lados da expressão, caso o primeiro lado da expressão seja verdadeiro, a execução sofre um desvio para o índice que contém o *opcode* que empilha o valor correspondente ao valor verdadeiro na pilha de operandos. Da mesma forma, no caso de o primeiro valor ser falso, é avaliado, então, o segundo lado da expressão e em caso do resultado da avaliação ser verdadeiro, é empilhado o valor correspondente, caso contrário, o valor que corresponde ao valor falso é empilhado na pilha de operandos.

Tabela 4.31: Exemplo da expressão *ou*

Português Estruturado	Bytecode	
a <- b OU c	0	iload 2
	2	ifne 14
	5	iload 3
	7	ifne 14
	10	`iconst_0`
	11	goto 15
	14	`iconst_1`
	15	istore 1

4.2.2.4.3 Operador Lógico NAO

Para tradução do operador lógico *nao* do Português Estruturado para *bytecode* foi utilizada a instrução de desvio condicional apresentada na Tabela 4.32. Este *opcode* de desvio condicional é utilizado para empilhar na pilha de operandos o valor da negação lógica.

Tabela 4.32: Opcode de desvio condicional utilizado no operador lógico *nao*

Mnemônico	Opcode
ifeq	*153 (0x99)*

Por exemplo, se uma expressão avaliada dentro dos parênteses do operador *nao* resultar no valor *verdadeiro* (produzindo no topo da pilha de operandos o valor inteiro 1), a instrução listada na Tabela 4.32 irá desempilhar o valor no topo da pilha de operandos, 1 no nosso caso de exemplo, e, como irá falhar no teste do condicional visto que o valor esperado é 0, fará um desvio para uma posição que contenha uma instrução que empilhe o valor 0, que é a instrução `iconst_0`, finalizando a execução com o valor *falso* no topo da pilha de operandos.

A Tabela 4.33 demonstra um exemplo de código da expressão NAO do Português Estruturado e sua tradução para *bytecode*.

Tabela 4.33: Exemplo da expressão *nao*

Português Estruturado	Bytecode	
a <- NAO(b OU c)	0	iload 2
	2	ifne 14
	5	iload 3
	7	ifne 14
	10	`iconst_0`
	11	goto 15
	14	`iconst_1`
	15	ifeq 22
	18	`iconst_0`
	19	goto 23
	22	`iconst_1`
	23	istore 1

4.3 Funções de leitura, escrita e pré-definidas

O Português Estruturado do Portal de Algoritmos possui funções de leitura de dados e escrita de dados na tela, bem como algumas funções matemáticas pré-definidas. Algumas funções foram traduzidas para *bytecode* de forma a fazerem chamadas para funções semelhantes implementadas na linguagem Java, desta forma, o interpretador na hora de executar esse comando faz apenas a chamada da função correspondente na instrução. Outras funções, como por exemplo a função *Int(x)* (Tabela 4.34), foi traduzida de forma a gerar um *opcode*, por conter um correspondente direto em *bytecode*. A Tabela 4.34 mostra algumas das funções pré-definidas existentes no Português Estruturado.

Tabela 4.34: Exemplos de funções pré-definidas do Português Estruturado

Função	Descrição
Abs(x)	Retorna o valor absoluto do valor recebido. O tipo de retorno é o mesmo do valor de entrada
Raizq(x)	Retorna a raiz quadrada do valor recebido. O tipo do valor de retorno é *real*
Sen(x)	Retorna o seno do ângulo recebido em radianos. Tanto o valor de entrada quando o de retorno são do tipo *real*
Cos(x)	Retorna o cosseno do ângulo recebido em radianos. Tanto o valor de entrada quando o de retorno são do tipo *real*
Int(x)	Retorna o valor inteiro (truncado) do valor de entrada (*real*)

4.4 Comandos Condicionais

Nesta seção serão apresentados os comandos condicionais do Português Estruturado e sua tradução para *bytecode*.

4.4.1 Comando Se

O comando condicional *se* é usado quando um bloco de comandos deve ser executado apenas se uma condição for verdadeira. Existem muitos comandos que realizam o desvio condicional[2,3], porém o escolhido para a tradução neste trabalho foi o comando listado na Tabela 4.35.

Este comando desempilha um valor do topo da pilha de operandos, que deve ser do tipo inteiro, e verifica se o valor é diferente de **zero**. Caso o valor seja diferente de **zero**, significa que uma avaliação anterior de uma expressão lógica resultou em **verdadeiro**, sendo assim, é feito um desvio para um índice que contenha os *bytecodes* que serão executados. Se o valor que estiver no topo da pilha de operandos for igual a **zero**, então, prossegue a execução da próxima instrução.

Tabela 4.35: Opcode de desvio condicional

Mnemônico	Opcode
ifne	*154 (0x9a)*

Na Tabela 4.36 é mostrado a estrutura do comando *se* traduzido para *bytecode*.

Tabela 4.36: Estrutura do comando *se* em *bytecode*

	Avalia resultado da expressão de controle
	ifne [verdadeiro]
falso:	Bloco executado se falso
	Desvio para o fim do comando Se
verdadeiro:	Bloco executado se verdadeiro
fimse:	Bloco executado em seguida

Na Tabela 4.37 é mostrado um exemplo de um trecho de código contendo o comando condicional em Português Estruturado e seu equivalente traduzido para *bytecode*.

[2] `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.if_icmp_cond`

[3] `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.if_cond`

Tabela 4.37: Exemplo da expressão *se*

Português Estruturado	Bytecode	
se b = 0	0	lload 1
entao	2	`iconst_0`
b <- 5	3	i2l
senao	4	lcmp
b <- b + 2	5	ifne 12
fimse	7	`iconst_5`
	8	lstore 1
	10	goto 19
	12	lload 1
	14	`iconst_2`
	15	i2l
	16	ladd
	17	lstore 1
	19	nop

4.4.2 Comando Escolha

O comando *escolha* do Português Estruturado executa um comando dependendo da comparação de uma expressão com uma sequência de possíveis resultados. Existem dois *opcodes* para representar este comando em *bytecode*[4]: o *opcode* **tableswitch** e o **lookupswitch**.

A diferença entre os dois *opcodes* está na eficiência de busca do índice do próximo comando a ser executado. O *opcode* **tableswitch** é utilizado quando as opções de cada caso do comando *escolha* podem ser representadas eficientemente como índices de uma tabela. Como por exemplo o caso de cada opção ser uma sequência de números (1, 2, 3, 4). Já o *lookupswitch* é utilizado no caso de as opções de cada caso do comando *escolha* sejam esparsas.

Neste trabalho, optou-se por usar o *opcode* do **lookupswitch**, por ser mais abrangente e também pelo fato de que a otimização não é uma prioridade desse trabalho. A Tabela 4.38 mostra o comando **lookupswitch**.

Tabela 4.38: Opcode do comando *escolha*

Mnemônico	Opcode
lookupswitch	*171 (0xab)*

A Tabela 4.39 mostra a estrutura do comando *escolha* traduzido para *bytecode*.

[4] `http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html#jvms-3.10`

Tabela 4.39: Estrutura do comando *escolha* em *bytecode*

	Carrega na pilha de operandos valor da variável de controle
	Procura na tabela de chaves de desvios o valor correspondente e executa o desvio para a posição do *opcode* correspondente
	Bloco executado pelo desvio da tabela de chaves do comando
	Desvio para o fim do comando
fim:	Bloco executado em seguida

A Tabela 4.40 demonstra a tradução de um exemplo de código utilizando o comando *escolha* do Português Estruturado para seu equivalente em *bytecode*.

Tabela 4.40: Exemplo do comando *escolha*

Português Estruturado		Bytecode
escolha b	0	lload 2
caso 2 : a <- b * 2	2	lookupswitch 4
caso 4: a <- b + 2	4	2 : 12
caso 16: a <- b / 4	6	4 : 23
padrão: a <- 0	8	16 : 34
fimescolha	10	default : 45
	12	lload 2
	14	`iconst_2`
	15	i2l
	16	lmul
	17	l2f
	18	fstore 1
	20	goto 48
	23	lload 2
	25	`iconst_2`
	26	i2l
	27	ladd
	28	l2f
	29	fstore 1
	31	goto 48
	34	lload 2
	36	l2f
	37	`iconst_4`
	38	i2f
	39	fdiv
	40	fstore 1
	42	goto 48
	45	`fconst_0`
	46	fstore 1
	48	nop

4.5 Comandos de Repetição

Nessa seção serão demonstradas as traduções para *bytecode* dos comandos de repetição do Português Estruturado (*repita, enquanto* e *para*).

4.5.1 Comando Repita

O comando *repita* é usado quando uma lista de comandos deve ser executada ao menos uma vez até que uma condição seja verdadeira. A Tabela 4.41 demonstra a estrutura do comando *repita* para *bytecode*.

Tabela 4.41: Estrutura do comando *repita* em *bytecode*

repita:	Bloco executado no laço do comando *repita*
	Avalia expressão de controle do laço
	Desvio para [repita] caso expressão de controle seja falsa
	Bloco executado em seguida

A Tabela 4.42 mostra um exemplo de tradução de um comando *repita* do Português Estruturado para *bytecode*.

Tabela 4.42: Exemplo do comando *repita*

Português Estruturado	Bytecode	
a <- 0	0	`lconst_0`
repita	1	lstore 1
a <- a + 2	3	lload 1
ate a > 128	5	`iconst_2`
	6	i2l
	7	ladd
	8	lstore 1
	10	lload 1
	12	bipush 128
	14	i2l
	15	lcmp
	16	iflt 3
	19	nop

4.5.2 Comando Enquanto

O comando *enquanto* é usado quando uma lista de comandos deve ser executada enquanto uma condição for verdadeira. A Tabela 4.43 mostra e estrutura do comando *enquanto* traduzido para *bytecode*.

Tabela 4.43: Estrutura do comando *enquanto* em *bytecode*

enquanto:	Avalia expressão de controle do laço
	Desvio para [fim] caso expressão de controle seja falsa
	Bloco executado no laço do comando *enquanto*
	Desvio para [enquanto]
fim:	Bloco executado em seguida

A Tabela 4.44 mostra um exemplo de tradução de um comando *enquanto* do Português Estruturado para *bytecode*.

Tabela 4.44: Exemplo do comando *enquanto*

Português Estruturado	Bytecode	
a <- 0	0	`lconst_0`
enquanto a < 1000 faca	1	lstore 1
a <- a + 2	3	lload 1
fimenquanto	5	ldc 1
	7	lcmp
	8	ifge 21
	11	lload 1
	13	`iconst_2`
	14	i2l
	15	ladd
	16	lstore 1
	18	goto 3
	21	nop

4.5.3 Comando Para

O comando *para* é usado quando uma lista de comandos deve ser executada n vezes, sendo n um número conhecido de vezes. A Tabela 4.45 mostra a estrutura do comando *para* traduzido para *bytecode*.

Tabela 4.45: Estrutura do comando *para* em *bytecode*

	inicialização da variável de controle do laço
para:	Avaliação da expressão de controle do laço
	Desvio para [fim] caso expressão seja falsa
	Bloco executado no laço do comando *para*
	Incremento/decremento da variável de controle do laço
	Desvio para [para]
fim:	Bloco executado em seguida

As Tabelas 4.46, 4.47 e 4.48 demonstram a tradução do comando *para*, com diferentes valores de passo, do Português Estruturado para *bytecode*.

Tabela 4.46: Exemplo do comando *para* com passo 1

Português Estruturado	Bytecode	
c <- 0	0	`lconst_0`
para a de 1 ate b faca	1	lstore 3
c <- c + b / a	3	`lconst_1`
fimpara	4	lstore 1
	6	lload 1
	8	lload 2
	10	lcmp
	12	ifgt 37
	15	lload 2
	17	l2f
	18	lload 1
	20	l2f
	21	fdiv
	22	f2l
	23	lload 3
	25	ladd
	26	lstore 3
	28	`lconst_1`
	29	lload 1
	31	ladd
	32	lstore 1
	34	goto 6
	37	nop

Tabela 4.47: Exemplo do comando *para* com passo maior que 1

Português Estruturado	Bytecode	
c <- 0	0	`lconst_0`
para a de 1 ate b passo 3 faca	1	lstore 3
c <- c + b / a	3	`lconst_1`
fimpara	4	lstore 1
	6	lload 1
	8	lload 2
	10	lcmp
	12	ifgt 38
	15	lload 2
	17	l2f
	18	lload 1
	20	l2f
	21	fdiv
	22	f2l
	23	lload 3
	25	ladd
	26	lstore 3
	28	`iconst_3`
	29	i2l
	30	lload 1
	32	ladd
	33	lstore 1
	35	goto 6
	38	nop

Tabela 4.48: Exemplo do comando *para* com passo negativo

Português Estruturado	Bytecode	
c <- 0	0	`lconst_0`
para a de 100 ate b passo -2 faca	1	lstore 3
c <- c + b / a	3	`lconst_1`
fimpara	4	lstore 1
	6	lload 1
	8	lload 2
	10	lcmp
	12	iflt 38
	15	lload 2
	17	l2f
	18	lload 1
	20	l2f
	21	fdiv
	22	f2l
	23	lload 3
	25	ladd
	26	lstore 3
	30	lload 1
	28	`iconst_2`
	29	i2l
	32	lsub
	33	lstore 1
	35	goto 6
	38	nop

4.6 Funções e Procedimentos

Nessa seção será representada a tradução da declaração de **funções** e **procedimentos** do Português Estruturado para o *bytecode*. Tanto as **funções** quanto os **procedimentos** podem receber parâmetros por valor ou referência. **Funções** devem ter um tipo de retorno.

Quando uma função retorna um valor, este é empilhado na pilha de operandos do contexto que chamou esta função (LINDHOLM et al., 2015). Os parâmetros passados a uma função ou procedimento (assim como as variáveis globais do programa) são armazenados na tabela de variáveis locais da função ou procedimento, sendo numeradas, na tabela de variáveis, iniciando do índice 1 até n. A posição 0 da tabela é reservada para a instância da classe o qual a função ou procedimento estão declaradas (LINDHOLM et al., 2015).

Não há na especificação da *JVM* uma forma de passar valores por referência. Os valores passados por parâmetro para uma função ou procedimento são somente por valor (LINDHOLM et al., 2015). Uma forma de se simular a passagem por referência, utilizado nesse trabalho, é a criação de um vetor de uma posição, o qual contém o valor passado, quando este for por referência, para a função ou procedimento.

A Tabela 4.49 demonstra a tradução de uma declaração de uma **função**. A *função* recebe um valor *inteiro* como parâmetro e retorna um valor *inteiro* como resultado.

Tabela 4.49: Exemplo de função

Português Estruturado		Bytecode
Funcao fat(n:inteiro):inteiro		Method long fat(long)
var i,f : inteiro	0	`lconst_1`
inicio	1	lstore 3
f <- 1	3	`lconst_1`
para i de 1 ate n faca	4	lstore 2
f <- f * i	6	lload 2
fimpara	8	lload 1
retorne f	9	lcmp
	11	ifgt 29
	14	lload 3
	16	lload 2
	18	lmul
	19	lstore 3
	20	`lconst_1`
	21	lload 2
	23	ladd
	24	lstore 2
	26	goto 6
	29	lload 3
	30	lreturn

5 *FRONT-END* DO USUÁRIO

Este capítulo irá abordar a interface do usuário, constituída pelo editor de código fonte e pelo serviço *REST* (*Representational State Transfer*) responsável pela comunicação entre o editor e o interpretador. A interface do editor de código é onde o usuário irá interagir através da criação e edição de código escrito em Português Estruturado. Outras interações possíveis, as quais serão melhor detalhadas mais adiante neste capítulo, incluem a depuração do código linha a linha, inserção de pontos de parada no código e visualização dos valores armazenados nas variáveis em tempo de execução, entre outros.

5.1 Estrutura do editor de código

Como a tecnologia em que foi desenvolvido o Portal e, consequentemente, seu editor de código está se tornando obsoleta e sem suporte pelos navegadores, um dos grandes problemas enfrentados hoje no Portal é justamente em encontrar um navegador que execute, ou que contenha configurações de segurança que permita a execução do Portal.

Tendo em vista essa limitação tecnológica, tratou-se neste trabalho, também, o desenvolvimento de uma nova interface de edição de código, utilizando tecnologias e recursos que sejam suportados e executem nos navegadores sem grandes problemas.

A tecnologia que vem sendo utilizada atualmente em grande maioria das aplicações e páginas *web* é o *HTML5*[1] e o *Javascript*[2]. O *HTML5* é a quinta versão da linguagem utilizada para estruturar e apresentar conteúdos na *web*. Esta versão da linguagem foi finalizada e publicada em 28 de outubro de 2014 pela *W3C* (*World Wide Web Consortium*). Uma das grandes novidades da nova versão foi o suporte aos novos formatos de mídia de forma nativa, como por exemplo a execução de vídeos, áudio, entre outros. Outras novidades são referentes a elementos da estrutura da página (VAN KESTEREN; PIETERS, 2008). *Javascript* é uma linguagem de alto nível, não tipada e interpretada. Em conjunto com o *HTML* e *CSS*, é uma das

[1] `https://www.incore.com/Fortune500HTML5/#infographic`
[2] `http://w3techs.com/technologies/details/cp-javascript/all/all`

tecnologias consideradas essenciais no desenvolvimento de páginas e conteúdo *web*.

Com base na utilização dessas tecnologias anteriormente citadas e que encontram amplo suporte nos navegadores atuais e em grande parte dos navegadores mais antigos, a nova interface utiliza-se dessas tecnologias para o desenvolvimento do editor de código e suas funcionalidades. Como o escopo deste trabalho não é sobre o desenvolvimento da interface *web* e do editor de código, mas sim sua atualização tecnológica, foi utilizado um componente, de código aberto e livre utilização, para o editor de texto do código fonte. Este componente é desenvolvido utilizando as tecnologias atuais utilizadas em aplicativos e páginas da *web*. Este componente para o editor de texto do código fonte é o *Ace*[3]. Este editor de texto especificamente desenvolvido para código fonte é de código aberto e livre utilização e entre seus usuários estão o *Github*, *Codeacademy*, *Chrome Dev Editor*, *Wikipedia*, *Firefox Add-on Builder*, entre outros[4].

Entre as funcionalidades deste editor de texto do código fonte, são destacadas algumas[5]:

- Destaque com cores para as palavras reservadas da linguagem e para constantes;
- Identação do código automática;
- Destaque de abre e fecha parênteses;
- Funcionalidades de copiar, recortar e colar texto;

5.2 Funcionalidades do editor de código

As funcionalidades existentes no Portal de Algoritmos atual foram mantidas na nova interface, porém serão desenvolvidas na interface utilizando tecnologias mais recentes, como por exemplo *HTML5*, *CSS* e *Javascript*. As funcionalidades implementadas estão divididas em grupos:

- Execução do algoritmo;
- Entrada de dados;
- Visualização de variáveis/computação;
- Tratamento de erros durante a execução;

5.2.1 Funcionalidades de Execução do Algoritmo

Este grupo de funcionalidades atua na forma como o usuário pode executar o seu algoritmo. Dentre as formas possíveis de execução, serão implementadas as

[3] `https://ace.c9.io/#nav=about`
[4] `https://ace.c9.io/#nav=production`
[5] `https://ace.c9.io/#nav=about`

seguintes:

- Execução direta: ao selecionar esta opção, através do clique em um botão, o código será executado inteiramente, apresentando, ao final, os resultados gerados. Há a possibilidade de temporização, sendo definido um tempo para a passagem de cada linha na execução do código;
- Ativação de pontos de parada (*breakpoints*): esta opção permite que o usuário selecione, em uma linha do código fonte, um ponto no qual o programa, ao ser executado, irá parar a execução e aguardar a retomada da execução pelo usuário. Esta funcionalidade é ativada através de dois cliques sobre o número da linha;
- Execução passo a passo: esta funcionalidade é executada através do clique em um botão, e a cada vez que o botão desta funcionalidade é acionado, uma linha do código fonte é executada. Com isso, é possível analisar os valores das variáveis declaradas no programa.

5.2.2 Funcionalidades de Entrada de Dados

Estas funcionalidades tornam automatizado o processo de entrada de dados no programa. No momento em que um comando de leitura for encontrado, essas funcionalidades fazem com que os valores sejam atribuídos de forma automática nas variáveis que recebem o valor lido. Essas funcionalidades desenvolvidas são:

- Geração de dados aleatórios: esta opção seleciona valores aleatórios, contido em uma faixa de valores definidos na opção de dados aleatórios, no caso de a variável que recebe os dados ser do tipo inteiro ou real. Para valores do tipo literal serão atribuídos valores aleatórios dentro de um conjunto pré definido de 200 literais;
- Uso de dados de testes pré-definidos: com esta opção selecionada, serão atribuídos os valores do exemplo, definidos no enunciado do problema escolhido para resolução;
- Dados informados pelo usuário: o usuário pode definir os dados que deseja que sejam utilizados.

5.2.3 Funcionalidade de Visualização de Variáveis/Computação

Esta funcionalidade permite que o usuário possa verificar os valores que estão sendo armazenados nas variáveis definidas no programa, bem como um histórico dos valores armazenados nas variáveis para os últimos 1000 comandos.

5.2.4 Funcionalidade de Exibição de Mensagem de Erro de Execução

Com esta funcionalidade, os erros encontrados durante a fase de execução do programa são exibidos. Durante a execução, no momento em que um erro é encontrado, a execução é finalizada e uma mensagem informativa é exibida em um controle contendo um campo de texto para a exibição da mensagem de erro. Na mensagem de erro é informada a mensagem do erro e a linha do código fonte na qual o erro foi disparado.

5.3 Serviço *REST*

Como a organização do Portal proposta neste trabalho abrange a separação entre a interface do usuário e o interpretador de código, ficando o primeiro no navegador, local, e o segundo executando em um servidor, remotamente, é necessário que uma arquitetura de serviço *web* seja definida para fazer a comunicação entre os dois componentes do sistema. Para tal, foi desenvolvido neste trabalho um serviço do tipo *REST*.

Serviços *web* utilizando a arquitetura *REST* baseiam-se em definições de restrições que se corretamente aplicadas ao serviço *web*, induzem a propriedades como escalabilidade, performance e facilidade de modificação do sistema. Na arquitetura *REST*, dados e funcionalidades são considerados como recursos e, desta forma, são acessados de maneira semelhante, através de *URLs* (*Uniform Resource Locators*). Serviços *REST* devem se basear nos seguintes princípios (ORACLE, 2011):

- Identificação de recursos através de *URLs*;
- Interface uniforme;
- Mensagens auto-descritivas;
- Interações através do uso de estados.

Com base nessa utilização dessa arquitetura de serviço *web* é feita toda a comunicação entre o interpretador executando no servidor e a interface do usuário, executado localmente no navegador. Este serviço foi definido para fazer tanto a transferência do código fonte escrito pelo usuário para o interpretador, bem como para o envio de mensagens do interpretador para a interface.

Funcionalidades como a execução passo a passo, parada devido à execução ter atingido um ponto no qual foi definido um ponto de parada (*breakpoint*), parada da execução para realização de leitura de dados ou para escrita de informações devido à execução do comando de escrita são todos exemplos de funcionalidades que são implementadas utilizando a arquitetura *REST* para a comunicação entre o interpretador e a interface do usuário.

6 DESENVOLVIMENTO

Este capítulo irá abordar o desenvolvimento do compilador de Português Estruturado para *bytecode*, conforme as definições tratadas anteriormente neste trabalho, bem como da interface de edição de código do usuário.

Alguns itens definidos anteriormente na fase de projeto sofreram alterações no decorrer do desenvolvimento, como por exemplo, ao invés da implementação de um interpretador para os comandos utilizados neste trabalho, será utilizada a própria *JVM* desenvolvido pela Oracle, a mesma que executa programas desenvolvidos na linguagem de programação Java.

6.1 Desenvolvimento do Compilador

O desenvolvimento do compilador ocorreu em duas fases distintas. Conforme especificado anteriormente, primeiro foi desenvolvido o *front-end*, responsável pela análise léxica e sintática, bem como a tradução do código fonte de entrada na representação intermediária. Essa representação intermediária está representada, no compilador desenvolvido para este trabalho, em uma *AST*.

A segunda fase do desenvolvimento do compilador resultou no desenvolvimento do *back-end*, responsável por receber como entrada a representação intermediária, no caso deste trabalho a *AST* gerada pelo *front-end*, e transformá-la na linguagem objeto deste compilador, o *bytecode*.

O compilador esta estruturado em um projeto Java de nome PortAlg.Compiler. Neste projeto está contido o *front-end* e o *back-end* do compilador. O *front-end* está no módulo de nome *Parser* e o *back-end* no módulo chamado *AST*.

6.1.1 Desenvolvimento do *Front-End*

O *front-end* do compilador do Portal de Algoritmos foi desenvolvido em grande parte utilizando a ajuda do software *JavaCC*. Conforme já definido anteriormente, este software auxilia na criação de um analisador léxico e sintático. Para tal, é necessário, como entrada para o *JavaCC*, uma definição da gramática da linguagem

que se destina a criação do analisador.

A partir da definição deste arquivo contendo a definição da gramática da linguagem e das produções sintáticas que se aplicam, o *JavaCC* cria os arquivos utilizados para a criação de um analisador léxico e sintático. Esses arquivos são gerados na linguagem Java.

Neste trabalho, a definição da gramática da linguagem do Português Estruturado foi feita no arquivo chamado PortAlgParser.jj. Neste arquivo estão contidas todos os *tokens*, todos os símbolos representando literais, todas as regras da linguagem e todas as produções sintáticas para geração da representação intermediária.

No Trecho de Código 6.1 é exemplificado a definição gramatical dos tipos inteiro e flutuante no arquivo PortAlgParser.jj.

Trecho de Código 6.1: Exemplo de definição de literais do tipo inteiro e do tipo flutuante contidas no arquivo PortAlgParser.jj

```
 1 TOKEN :
 2 {
 3 < INTEGER_LITERAL: ["0"-"9"] (["0"-"9"])*  >
 4 |
 5 < FLOATING_POINT_LITERAL:
 6 (["0"-"9"])+ <DOT> (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
 7 | <DOT> (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
 8 | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
 9 | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
10 >
11 |
12 < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
```

O Trecho de Código 6.2 contém um exemplo da definição gramatical e de produção da *AST* de um bloco de comandos, contido no arquivo PortAlgParser.jj.

Trecho de Código 6.2: Exemplo de definição da gramática e de produção sintática para um bloco de comandos, contidas no arquivo PortAlgParser.jj

```
 1 SPStatement Statement() :
 2 {
 3     SPStatement statement = null;
 4 }
 5 {
 6     (
 7     statement = StatementExpression()
 8     |
 9     statement = SwitchStatement()
10     |
11     statement = IfStatement()
12     |
13     statement = WhileStatement()
14     |
15     statement = DoStatement()
16     |
17     statement = ForStatement()
18     |
```

```
19      statement = BreakStatement()
20      |
21      statement = ContinueStatement()
22      |
23      statement = WriteStatement()
24      |
25      statement = ReadStatement())
26
27      {return statement; }
28  }
```

A partir da entrada deste arquivo PortAlgParser.jj no *JavaCC*, as classes de análise léxica e sintática são geradas. Para este trabalho, as classes geradas pelo *JavaCC*, bem como uma breve descrição de cada uma, esta contida na Tabela 6.1.

Tabela 6.1: Classes geradas pelo *JavaCC* a partir da definição da gramática

Classe gerada	Definição
SimpleCharStream.java	Representação da sequência de caracteres
Token.java	Objeto contendo as propriedades de um *token*
PortAlgTokenManager.java	Responsável por ler a sequência de caracteres e retirar os *tokens* definidos na especificação
PortAlgParserConstants.java	Constantes representando cada um dos *tokens* definidos na especificação
TokenMgrError.java	Retorna uma mensagem detalhada de erro na análise léxica
ParseException.java	Exceção que é lançada quando de erro na análise sintática. Pode ser estendida e customizada
PortAlgParser.java	Programa principal que executa a análise sintática e quaisquer outras ações definidas na especificação

Estas classes, geradas pelo *JavaCC* através da especificação gramatical contida no arquivo PortAlgParser.jj, compõem a primeira parte do desenvolvimento do *front-end* do compilador.

A Figura 6.1 mostra um diagrama resumido das classes geradas pelo *JavaCC*.

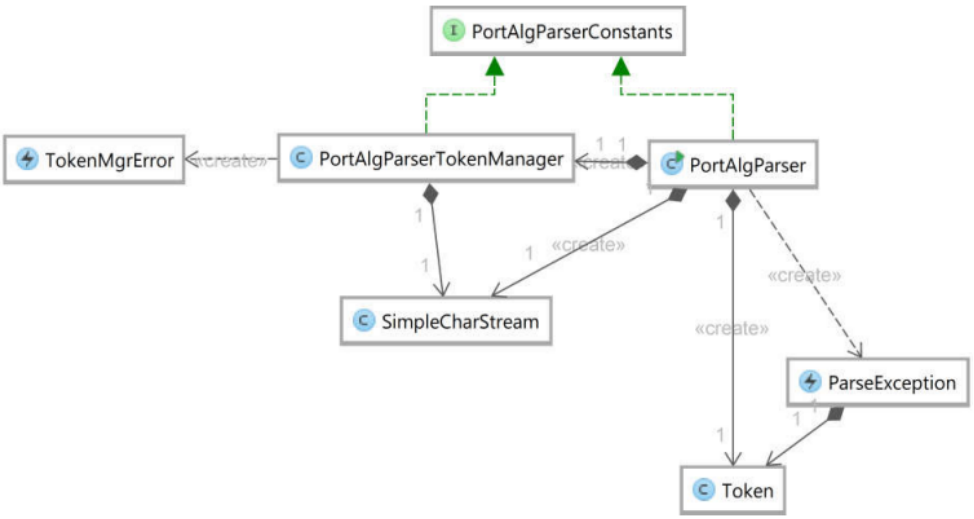

Figura 6.1: Diagrama de classes resumido do *front-end* do compilador.
Do próprio autor

6.1.2 Desenvolvimento do *Back-End*

O *back-end* do compilador compreende a parte da *AST* bem como as classes que traduzem a representação intermediária da *AST* em *bytecode*. Para o desenvolvimento da árvore foram definidas classes para representar os comandos e estruturas definidas para a linguagem do Português Estruturado, sendo o nodo mais importante, o nodo raiz da *AST*, a classe SPCompilationUnit.java.

O nodo raiz SPCompilationUnit.java contem as informações da classe que será gerada após a tradução da representação intermediária em *bytecode*. Esta classe contém informações como o nome da nova classe que será gerada, os campos que ela contém, os métodos que forem declarados em Português Estruturado e o método principal da classe. Para exemplificar a tradução de um trecho de código de Português Estruturado para a representação intermediária *AST*, a Figura 6.2 contém um exemplo da árvore que traduz os comandos e instruções contidos no Trecho de Código 6.3.

Trecho de Código 6.3: Código exemplo em Português Estruturado

```
1 algoritmo "meu alg"
2 var
3     a, b : inteiro
4 inicio
5     a <- 2
6     b <- 4
7     a <- a * b
8     escreva(a)
9 fimalgoritmo
```

Figura 6.2: Exemplo de AST gerada pelo *front-end* para o código do exemplo
Do próprio autor

Na Figura 6.2 é possível identificar o nodo raiz da *AST*, **SPCompilationUnit**, o qual contem toda a definição da classe que será criada futuramente após a tradução em *bytecode*.

Em seguida, vem o nodo **SPClassDeclaration**, contendo as informações essenciais da classe, como os campos da classe e os métodos definidos para a mesma. No desenvolvimento deste trabalho, todas as variáveis declaradas para o escopo principal, ou seja, as variáveis não definidas dentro de uma função ou procedimento, serão traduzidas para *bytecode* como um **campo** da classe.

O nodo **SPFieldDeclaration** contém todos os nodos responsáveis pela declaração de um campo, ou seja, uma variável declarada em um programa em Português Estruturado, na nova classe traduzida. Seus nodos filhos, são classes do tipo **SPVariableDeclarator**.

Em seguida, o próximo nodo filho do nodo **SPClassDeclaration** é o nodo **SPMethodDeclaration**. Este nodo é responsável por armazenar as informações e nodos que representam uma declaração de um método. No caso do exemplo do

Trecho de Código 6.3, este é a função principal do programa, que mais tarde será traduzida no método *main* da nova classe *bytecode*.

Neste exemplo, o nodo **SPMethodDeclaration** contém dois filhos: **SPFormalParameter** e **SPBlock**. O nodo **SPFormalParameter** representa os parâmetros que uma função possui em sua assinatura. O nodo **SPBlock** é o nodo que contém todas as definições dos comandos que compõem o corpo de uma função, como por exemplo, comandos de leitura, escrita, funções aritméticas ou comandos de repetição.

A Figura 6.3 mostra um diagrama compacto do *back-end* do compilador. Uma versão mais ampliada do diagrama encontra-se no cd da publicação deste trabalho.

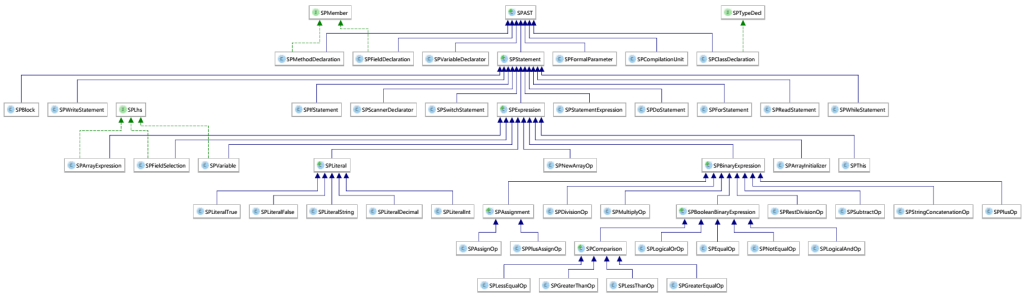

Figura 6.3: Diagrama de classes do *back-end* do compilador
Do próprio autor

6.1.3 Desenvolvimento do *Debugger*

O *debugger* é uma parte do desenvolvimento deste trabalho que difere do que foi especificado anteriormente. Ao invés do desenvolvimento de um interpretador dos códigos *bytecodes* conforme proposto, optou-se por usar a própria *JVM* da Oracle, a mesma que executa programas desenvolvidos na linguagem Java, uma vez que os programas escritos em Português Estruturado pelo Portal são compilados para a mesma representação que os programas escritos em Java.

Para poder fazer uso das capacidades de depuração da *JVM*, fez-se necessário o desenvolvimento de um programa que se comunicasse com uma *JVM* em execução e não apenas isso, mas que também pudesse receber avisos de escrita na saída padrão da execução, bem como poder informar a entrada de dados do usuário executando o Portal no navegador. Outras funções que o depurador do Portal deveria possuir é a capacidade de inserir pontos de parada nas linhas de código definidas pelo usuário e também ser informado quando a *JVM* atingisse os pontos de parada anteriormente definidos, a fim de poder informar ao usuário através da interface do Portal no navegador.

Para desenvolver este programa capaz de se comunicar com a *JVM* executando

o código escrito pelo usuário, foi feito uso da *JPDA* (*Java Platform Debugger Architecture*), que consiste de interfaces desenvolvidas pela Oracle que descrevem serviços que uma *JVM* deve prover para *debugging*. Contém também uma interface que define as informações que devem ser trocadas entre o processo que esta depurando e o *front-end* do *debugger*. São definidas também interfaces e requisições a nível de usuário [1].

Especificamente para este trabalho, foi utilizado o *JDI* [2], que constitui a *API* (*Application Programming Interface*) de mais alto nível de acesso a uma *JVM*. Com esta *API*, é possível obter informações sobre um processo executando em uma *JVM*, como por exemplo, os valores das variáveis do processo, parar a execução, inserir *breakpoints* e modificar a saída e entrada padrão de dados do processo da *JVM*.

Através da *JDI* é possível tanto para uma aplicação que irá depurar código conectar-se a uma máquina virtual que já esteja em execução (tanto localmente quanto em outro computador) quanto iniciar uma máquina virtual e conectar-se a ela posteriormente. Para este trabalho, optou-se por iniciar uma máquina virtual nova e conectar-se a ela. Isso deve-se ao fato de que um programa de depuração que inicia a sua própria máquina virtual pode exercer controle sobre o processo da máquina virtual, o que incide em poder alterar os métodos de entrada e saída (escrita e leitura de dados do usuário) da máquina virtual.[3] [4]

Para cada programa escrito e compilado de Português Estruturado para *bytecode*, é iniciada uma nova instancia do depurador de código, que por sua vez inicia uma instancia de uma máquina virtual *JVM* para poder executar o programa compilado e também depurá-lo, ou seja, executar os comandos linha-a-linha, inserir e parar em pontos de parada definidos pelo usuário e poder mostrar valores armazenados dentro de variáveis do programa executado.

Este programa foi desenvolvido em um projeto chamado PortAlg.Debugger. Este projeto cria um pacote, desenvolvido na linguagem Java, o qual depois é utilizado no serviço *web*, o mesmo que recebe o código do usuário e o envia para o compilador. Após o compilador produzir o código em *bytecode*, em um arquivo .class, este é, então, enviado para o programa de depuração (PortAlg.Debugger), que irá iniciar uma nova máquina virtual *JVM* que executará o programa, informando para o usuário as saídas do programa e informando as entradas que o mesmo precisa informar para a execução do programa.

A Figura 6.4 mostra um diagrama do *debugger* desenvolvido neste trabalho.

---

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/index.html

[2] http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html

[3] `http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachine.html#process()`

[4] `http://docs.oracle.com/javase/7/docs/api/java/lang/Process.html?is-external=true`

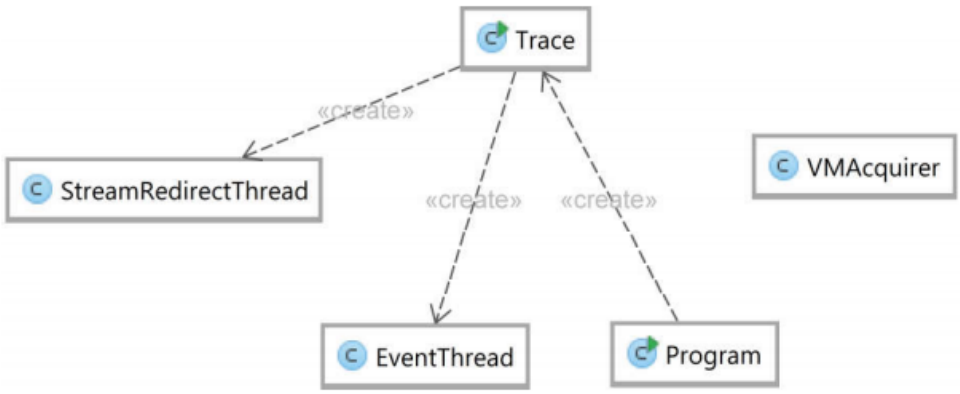

Figura 6.4: Diagrama de classes do *debugger* desenvolvido neste trabalho
Do próprio autor

6.2 Desenvolvimento do Serviço *Web*

O serviço *web* do Portal de Algoritmos é responsável pela comunicação entre a interface do usuário, que é o editor de código executado no navegador do mesmo, e o compilador do Portal de Algoritmos, executando em um servidor.

O serviço *web* foi desenvolvido utilizando a linguagem Java e o *Wildfly 10* como servidor da aplicação. A comunicação entre a interface do usuário e o compilador no servidor é feita utilizando serviços REST. Através do método compilar, um método do tipo *PUT* do protocolo *HTTP* (*HyperText Transfer Protocol*), a interface executada no navegador do usuário envia o código desenvolvido pelo mesmo para o serviço, o qual recebe esta requisição contendo o código do usuário e envia o mesmo para o compilador, que executará, então, a tradução do mesmo de Português Estruturado para uma classe contendo códigos *bytecodes* e em seguida executará este código na *JVM*.

A cada fase da compilação ou da execução do programa, desenvolvido pelo usuário e traduzido para *bytecode*, o serviço *web* comunica para a interface do usuário, no navegador, em que etapa se encontra a execução do compilador. Caso haja saída de dados, ou seja, um comando de escrita no código produzido pelo usuário, o serviço *web* irá retornar a mensagem para a interface do usuário, para que a mesma seja exibida para o usuário. Após informar a interface, o serviço *web* avisa para o *debugger* do compilador que deve continuar o processamento da execução do programa. O mesmo ocorre para o caso de o usuário ter escolhido a opção de executar linha-a-linha o seu programa. O serviço *web* fica responsável por avisar para a interface do usuário no navegador o momento em que cada linha do programa esta sendo executado, e aguardar que o usuário responda com o comando de continuar a execução, para então avisar ao *debugger* que deve continuar mais um passo na execução do

programa.

A entrada de dados em um programa, por exemplo quando um usuário insere um comando de leitura no seu código, também é gerenciada pelo serviço *web*. O serviço fica responsável por alertar o usuário executando o editor de código no navegador de que o mesmo deve introduzir dados para que o programa possa continuar sua execução. Após o usuário informar o dado e o navegador enviar o mesmo para o serviço *web*, o serviço transmite, então, para o *debugger* do compilador o dado de entrada recebido do usuário, para que possa continuar a execução do programa.

Outra comunicação feita pelo serviço *web* é quando o *debugger* atinge uma linha de código onde o usuário definiu um *breakpoint*. Neste caso, o serviço *web* avisa à interface do usuário que a linha contendo o *breakpoint* definido por ele mesmo foi alcançada.

6.3 Desenvolvimento do Editor de Código

O editor de código do Portal de Algoritmos é onde o usuário interage com o sistema. É onde o código em Português Estruturado é escrito pelo usuário. Na interface de edição de código, o usuário pode executar um programa desenvolvido, verificar o estado da execução, ou seja, se ocorreram erros, se o programa executou até o final, as mensagens e a saída do programa executado.

O editor de código foi desenvolvido utilizando *HTML* e *CSS* e outros componentes prontos, de código aberto e uso livre. Para o componente do editor de texto, foi utilizado o *Ace* [5], que é um editor de código para o uso em sites. Ele é totalmente desenvolvido em *Javascript*. Já vem com números das linhas, cores diferentes nos *tokens* da linguagem, temas diferentes, entre outros.

Para o layout geral do Portal, foi utilizado o *Bootstrap* [6], que é uma plataforma de desenvolvimento de sites totalmente responsíveis, isto é, que se ajustam e se adaptam facilmente à qualquer tamanho de tela do usuário. As funcionalidades do editor de código do Portal foram mantidas o mais fiéis possível. Na Figura 6.5 é apresentado um protótipo de interface para o editor de código do Portal de Algoritmos. Outras tecnologias envolvidas no desenvolvimento do editor são o *AngularJS*[7], *JQuery*[8]. Estes últimos, *frameworks* desenvolvidos baseados em *Javascript*, foram utilizados para estruturar o portal, facilitar a comunicação com o serviço *web* do Portal bem como para manipular de forma mais rápida e dinâmica os elementos *HTML* do site.

[5] https://ace.c9.io/
[6] http://getbootstrap.com/
[7] `https://angularjs.org/`
[8] `https://jquery.com/`

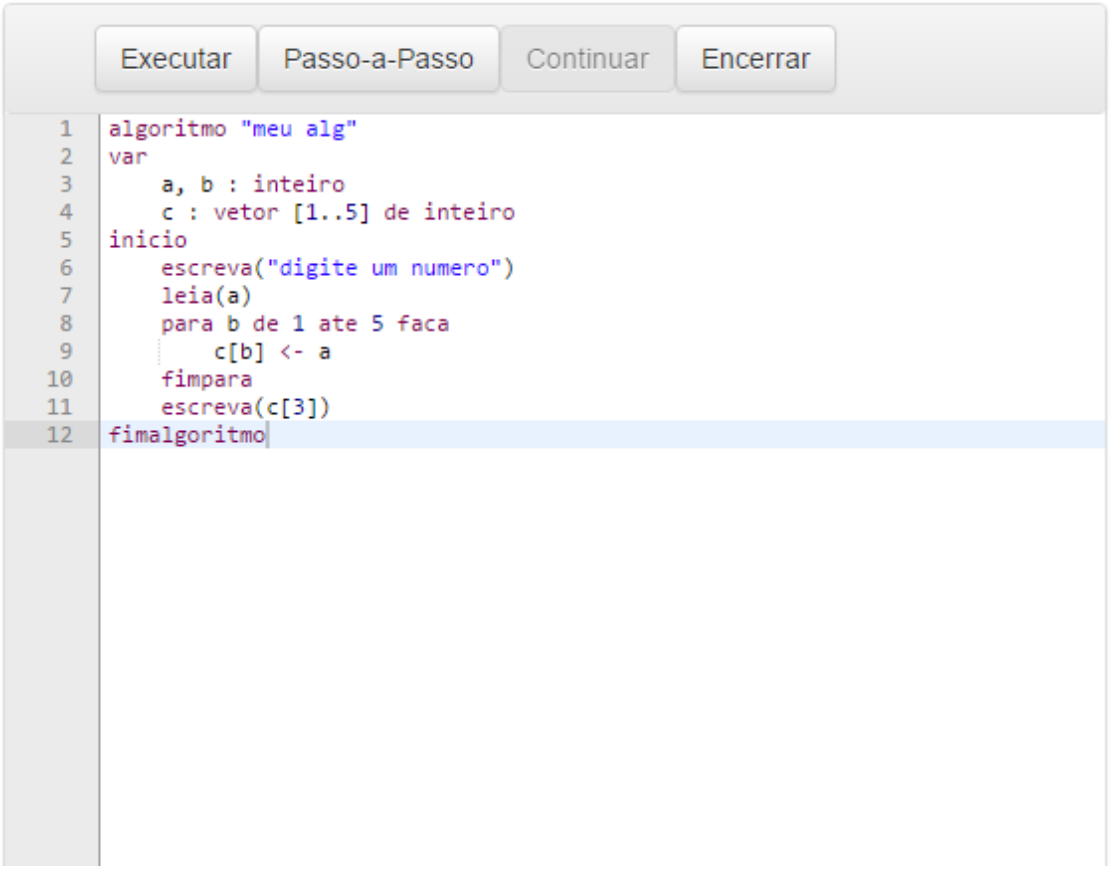

Figura 6.5: Protótipo de interface de editor de código
Do próprio autor

7 CONSIDERAÇÕES FINAIS

Neste trabalho foi desenvolvido um compilador de Português Estruturado para *bytecode* bem como um programa utilitário para execução e depuração do código compilado. A escolha pelo *bytecode* deu-se pelo fato de que a representação por *bytecodes* cria um código menor e compacto. Também por se tratar de uma representação bem definida através de um padrão especificado pela *The Java Virtual Machine Specification*[1]. Por ser uma representação padronizada e bem documentada, torna-se fácil a implementação de seus códigos e a posterior manutenção e o desenvolvimento de melhorias que possam vir a serem planejadas para o compilador do Portal. Também foi desenvolvido um serviço *web* que comunica-se com uma interface de edição de código que executa em um navegador.

Foram utilizadas tecnologias já existentes no mercado, todas de código aberto e especificações feitas para serem usadas pela comunidade de desenvolvedores justamente na criação de novos softwares que possam atender necessidades específicas, com as descritas neste trabalho. As tecnologias utilizadas em sua grande maioria derivam-se das tecnologias da família Java, mantida e desenvolvida atualmente pela *Oracle*.

Optou-se por utilizar tecnologias já existentes justamente para que os esforços empregados neste trabalho fossem totalmente redirecionados para alcançar os objetivos propostos no mesmo. Como já existem tecnologias que atendem o que se propunha realizar neste trabalho, optou-se, então, por usa-las. Não haveria motivos para que se criasse novamente componentes, representações ou arquiteturas para compilação ou execução de código, sem que as mesmas não produzissem novidades ou melhorassem algo nas tecnologias e recursos já existente.

Como sugestão para trabalhos futuros, fica a implementação do compilador aqui desenvolvido para integrar e ser utilizado na substituição do Portal de Algoritmos atualmente utilizado.

Propõe-se também, que seja refeita a interface de usuário, o editor de código, para que novas tecnologias sejam utilizadas, tecnologias essas que tenham suporte

[1] http://docs.oracle.com/javase/specs/jvms/se8/html/index.html

dos navegadores.

Ao final deste trabalho, conclui-se que, apesar de os objetivos propostos na Seção 1.1.1 não puderam ser atingidos em sua totalidade, foi possível demonstrar com este trabalho que um código em linguagem de Português Estruturado pode ser bem representado por *bytecodes*, bem como pode-se executar e depurar um programa, compilado de Português Estruturado para *bytecode*, utilizando uma versão comercial da *JVM*, como a da Oracle, por exemplo, que executa programas escritos na linguagem de programação Java. Com isso reduz a complexidade do Portal ao tirar a necessidade de se desenvolver também um interpretador para executar o código em *bytecode*. Isto abre novas possibilidades para se implementar no Portal, como por exemplo a portabilidade dos programas escritos em Português Estruturado (isto é, executar em qualquer plataforma ou sistema operacional), implementação de novas linguagens no Portal, bem como de novos recursos na linguagem do Português Estruturado.

REFERÊNCIAS

AHO, A. V. et al. **Compiladores**: princípios, técnicas e ferramentas. 2.ed. São Paulo: Pearson Addison-Wesley, 2008. tradução Daniel Vieira; revisão técnica Mariza Bigonha.

ALVES, L.; BRITO, M. O ambiente moodle como apoio ao ensino presencial. **Actas do 12º Congresso Internacional da Associação Brasileira de Educação a Distância**, Florianópolis, SC, 2005.

CAMPBELL, B.; IYER, S.; AKBAL-DELIBAS, B. **Introduction to Compiler Construction in a Java World**. Boca Raton, FL: CRC Press, 2012.

CHAVES, J. O. M. et al. MOJO: uma ferramenta para integrar juízes online ao moodle no apoio ao ensino e aprendizagem de programação. **HOLOS**, Natal, RN, v.5, p.246–260, 2014.

CHEANG, B. et al. On automated grading of programming assignments in an academic institution. **Computers & Education**, Oxford, Inglaterra, v.41, n.2, p.121–131, 2003.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2.ed. Rio de Janeiro: Elsevier, 2014. tradução Daniel Vieira.

DORNELES, R. V.; JUNIOR, D. P.; ADAMI, A. G. AlgoWeb: a web-based environment for learning introductory programming. **ADVANCED LEARNING TECHNOLOGIES (ICALT)**, Sousse, v.10, p.83–85, 2010.

FRANCO, M. A.; CORDEIRO, L. M.; CASTILLO, R. A. O ambiente virtual de aprendizagem e sua incorporação na Unicamp. **Educação e Pesquisa**, Campinas, SP, v.29, n.2, p.341–353, 2003.

HOSTINS, H.; RAABE, A. Auxiliando a aprendizagem de algoritmos com a ferramenta webportugol. **XV WEI**, Rio de Janeiro, RJ, 2007.

KAMIYA, R. R.; BRANDÃO, L. O. iVProg-um sistema para introdução à Programação através de um modelo Visual na Internet. **Anais do XX Simpósio Brasileiro de Informática na Educação**, Florianópolis, SC, 2009.

KUMAR, S.; GANKOTIYA, A. K.; DUTTA, K. **A comparative study of moodle with other e-learning systems.** , Kanyakumari, India, v.5, p.414–418, 2011.

LATTNER, C. et al. **The LLVM compiler infrastructure**. <Disponível em: `http://llvm.org`>. Acesso em: 10 de nov. de 2015.

LÈVY, P. **A inteligência colectiva - Para uma antropologia do ciberespaço**. Lisboa, Portugal: Ed. Instituto Piaget, 1994.

LINDHOLM, T. et al. **The Java Virtual Machine Specification**: java se 8 edition. <Disponível em: `http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`>. Acesso em: 9 de nov. de 2015.

MAK, R. **Writing Compilers and Interpreters**: an applied approach using c++. 2.ed. USA: Wiley Computer Publishing, 1996.

MANSO, A.; OLIVEIRA, L.; MARQUES, C. G. **Portugol IDE–Uma ferramenta para o ensino de programação**. Tomar, Portugal: PAEE, 2009.

NORVELL, T. S. **The JavaCC FAQ**. <Disponível em: `http://www.engr.mun.ca/~theo/JavaCC-FAQ/`>. Acesso em: 9 de nov. de 2015.

ORACLE. **The Java EE 6 Tutorial**: what are restful web services? <Disponível em: `https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html`>. Acesso em: 20 de nov. de 2015.

ORACLE. **Lesson - Java Applets**. <Disponível em: `https://docs.oracle.com/javase/tutorial/deployment/applet/`>. Acesso em: 23 de ago. de 2015.

REVILLA, M. A.; MANZOOR, S.; LIU, R. Competitive learning in informatics: the uva online judge experience. **Olympiads in Informatics**, Cairo, Egito, v.2, p.131–148, 2008.

RICCIO, N. C. R. Ambientes virtuais de aprendizagem na UFBA: a autonomia como possibilidade. **Doutorado–Universidade Federal da Bahia. Faculdade de Educação, Salvador**, Salvador, BA, 2010.

SMITH, J. E.; NAIR, R. The architecture of virtual machines. **Computer**, USA, v.38, n.5, p.32–38, 2005.

SOUZA, C. M. de. Visualg-ferramenta de apoio ao ensino de programação. **Revista TECEN**, Vassouras, RJ, v.2, p.1–9, 2009.

VAN KESTEREN, A.; PIETERS, S. **HTML5 differences from HTML4**. <Disponível em: `http://www.w3.org/TR/html5-diff/`>. Acesso em: 20 de nov. de 2015.

VARELLA, P. G. et al. Aprendizagem colaborativa em ambientes virtuais de aprendizagem: a experiência inédita da pucpr. **Revista Diálogo Educacional**, Curitiba, PR, v.3, n.6, p.11–27, 2002.

ZHIGANG, S. et al. Moodle Plugins for Highly Efficient Programmin Courses. **Moodle Research Conference**, Heraklion, Creta, v.1, p.157–163, 2012.