Tabulogo: linguagem de programação baseada em Logo para software de Geometria Dinâmica

Alexandre Ferreira Sardinha de Mattos

Orientador: Luiz Carlos Guimarães

Co-orientador: Oswaldo Vernet de Souza Pires

UFRJ / IM / DCC

2009

Tabulogo: linguagem de programação baseada em Logo para software de Geometria Dinâmica

Alexandre Ferreira Sardinha de Mattos

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Apresentado por:

Alexandre Ferreira Sardinha de Mattos

Aprovado por:

Prof. Oswaldo Vernet de Souza Pires, D.Sc.

Prof. Adriano Joaquim de Oliveira Cruz, Ph.D.

Prof. Miguel Jonathan, M.Sc.

RIO DE JANEIRO, RJ – BRASIL

Fevereiro de 2009

Resumo

O presente trabalho tem o objetivo de apresentar a linguagem de programação Tabulogo, uma linguagem baseada em Logo concebida para suprir algumas necessidades do software de Geometria Dinâmica Tabulæ. Apresentaremos os conceitos envolvidos no Tabulogo, sua implementação e alguns exemplos que ilustram o funcionamento da linguagem.

Abstract

This work aims to present the programming language Tabulogo, a language based on Logo, designed to meet some needs of Tabulæ, a Dynamic Geometry software. We present the concepts involved in Tabulogo, its implementation and some examples that illustrate the use of language.

Agradecimentos

Primeiramente, gostaria de agradecer à minha mãe Josefina e ao meu pai Manoel José, pelo exemplo de conduta e por tudo que fizeram por mim. Agradeço ao meu avô Domingos, que também apoiou minha formação. Também gostaria de deixar registrado um agradecimento a Dona Rita e família, Dona Maria, "tia" Kátia e Dona Mirabel pelo carinho especial quando mais precisei.

Agradeço aos meus amigos que me apoiaram e souberam compreender quando não pude estar presente: Alexandre Nascimento, Clarice, Eduardo, Gabriel Dottori, Leandro, Raphael Martins, Renato, Tatiana, Thaian, Thiago, Wando e Warny. Especialmente à minha namorada Eliza.

Gostaria de agradecer as instituições de ensino que me formaram. Ao Colégio Sagres, pela minha base e pela bolsa de estudos concedida. Ao Colégio Pedro II, que ampliou meus horizontes. E a UFRJ, que foi minha segunda (ou primeira) casa nestes 4 anos e meio de graduação.

Agradeço aos professores do curso de Ciência da Computação da UFRJ por todos os ensinamentos, em especial os professores Collier, Nelson Quilula, Ageu Cavalcanti, Adriano Cruz, Thomé, Paulo Aguiar, João Carlos, Márcia Cerioli e Gabriel Silva.

Agradeço aos meus companheiros de turma, em especial Bruno "Codorna", Alex Sanches, Thiago "Escazi", Anselmo Luiz, Douglas Cardoso, Carlos Henrique, Cristiano Expedito e Wendel. Não poderia deixar de citar Rafael Espirito Santo, pela companhia em tantos finais de semana de estudo. Um agradecimento também a outros colegas de curso: Thiago "Matrix", Denilson, Edno e Lúcio. Especialmente a Rafael Pinto, por toda a ajuda durante a graduação.

Quero agradecer ao pessoal do Labma: Rodrigo "Cuco", o guru do Latex; Thiago Guimarães, pelo auxílio com o Tabulæ; Rafael Barbastefano e Renato Mauro, por apontarem caminhos para este trabalho e Francisco Mattos, Ulisses, Rodrigo Devolder, Filipe Hasche, Daniela, Felipe "Moita" e Aline, por incentivos e sugestões.

Por fim, gostaria de agradecer também a meu orientador Luiz Carlos Guimarães e a Oswaldo Vernet por toda dedicação a este trabalho. Agradeço também aos professores Adriano Cruz e Miguel Jonathan pela participação na banca avaliadora.

Alexandre Ferreira Sardinha de Mattos

06/02/09

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Outras possibilidades 2

1.3 Sobre o trabalho 3

2 Conceitos **4**

2.1 Geometria Dinâmica 4

2.2 Tabulæ 5

2.3 Logo 6

2.4 Linguagens de Programação e Compiladores 9

2.4.1 Produção, Token, Variáveis 9

2.4.2 Gramáticas Livres de Contexto 10

2.4.3 Análise do Programa Fonte 11

2.4.4 Analisadores sintáticos *Bottom-up* e *Top-down* 12

2.4.5 Gramáticas LL, LR e LALR 13

2.4.6 *Parsers* Recursivos Descendentes 14

2.5 Geradores Automáticos de *Parsers* 15

3 Implementação **16**

3.1 Java 16

3.2 Javacc 17

3.2.1 Gramática Tabulogo 19

3.3 Padrões de Projeto 21
3.3.1 Padrão *Command* 21
3.3.2 Padrão *Façade* 23
3.4 Arquitetura Tabulogo 25
3.5 Características da Linguagem 26

4 Exemplos **28**
4.1 Rosácea 28
4.2 Planificação do Cone 30
4.3 Curva de Koch 32
4.4 Tangentes e Círculos 34
4.5 Integração Numérica 35

5 Conclusões e Trabalhos Futuros **39**
5.1 Conclusões 39
5.2 Trabalhos futuros 40

A Código fonte: cone.tlg **46**

B Código fonte: koch.tlg **49**

C Código fonte: tangente.tlg **51**

D Código fonte: integral.tlg **53**

Capítulo 1

Introdução

1.1 Motivação

É notória a influência da Ciência da Computação nos mais diversos setores da sociedade. Esta influência pode ser sentida principalmente na educação, pois houve uma transformação na maneira de o indivíduo interagir com a informação. Em particular, ambientes computacionais vêm transformando a prática e o ensino de Matemática.

Entre esses ambientes computacionais, destacam-se os softwares de Geometria Dinâmica, um tipo de software que permite que os usuários criem e manipulem construções geométricas dinamicamente. As construções são feitas predominantemente através do uso do mouse. Outra maneira de criar construções em um software de Geometria Dinâmica é através de macros.

Macro-construções são elementos importantes em qualquer programa de Geometria Dinâmica. Através delas, é possível encapsular diversas etapas de uma construção em um único comando, facilitando o processo de construções mais complexas e, por conseguinte, enriquecendo a lista de construções disponíveis aos usuários.

No entanto, existem alguns tipos de construções que são difíceis ou impossíveis de serem realizadas com o mouse ou até mesmo com as macros. Isto ocorre pois algumas construções dependem de uma condição e há a necessidade de uma estrutura de decisão como o *if*. Outras construções se tornam inviáveis pois há uso intenso de repetição ou até mesmo de recursão. Porém, nem todas macros de

programas de Geometria Dinâmica possuem esse tipo de funcionalidade.

Para isso, surgiu a idéia de construir uma linguagem de programação que fosse capaz de interagir com o programa de Geometria Dinâmica, no nosso caso o Tabulæ. Para facilitar sua disseminação, decidimos nos basear em outra linguagem, famosa no mundo da Geometria, o Logo. Assim, utilizando o ferramental presente em uma linguagem, expandimos as possibilidades de construções, tornando-as mais flexíveis e poderosas. Além disso, traz para os usuários de Logo a possibilidade de elaborar construções nas quais as propriedades geométricas são mantidas, por se tratar de um ambiente de Geometria Dinâmica.

1.2 Outras possibilidades

Uma linguagem de programação integrada com um software de Geometria Dinâmica também traz outras possibilidades. Além de suprir a falta de estruturas de repetição como o `for` e de controle como o `if`, o Tabulogo se torna uma alternativa à macro de teclado e mouse, já presente no Tabulæ. Virtualmente, tudo o que é possível de ser feito através da macro de teclado e mouse pode ser feito através do Tabulogo. Cabe ao usuário decidir o que mais conveniente para o seu propósito.

Uma outra possível aplicação do Tabulogo é no ensino de programação. Por exemplo, muitas vezes, estudantes de Matemática se sentem desestimulados em estudar programação por não enxergarem em como a programação pode auxiliar em suas carreiras. O Tabulogo oferece um incetivo para o aprendizado de programação de um estudante interessado em Geometria. Além disso, reduz a abstração da programação, já que o estudante consegue manipular diretamente os resultados (objetos geométricos) de seus programas.

Em última análise, também podemos pensar o Tabulogo em novo meio de explicitar o conhecimento sobre construções geométricas via texto. Quando descrevemos uma construção via Tabulogo, estamos introduzindo mais um recurso para alunos e professores poderem interagir com o texto que descreve construções geométricas. Assim, novas possibilidade pedagógicas podem ser exploradas.

1.3 Sobre o trabalho

Aqui descreveremos como o restante deste trabalho está organizado.

No capítulo 2, apresentaremos todos os conceitos envolvidos no Tabulogo, começando pela Geometria Dinâmica, passando pelo Tabulæ Logo, Linguagens de Programação e Compiladores e, finalmente, os Geradores Automáticos de Parsers.

No capítulo 3, abordaremos as tecnologias utilizadas na implementação da linguagem, decisões de projeto tomadas e as etapas de elaboração, além de uma breve análise do Tabulogo sob o ponto de vista das linguagens de programação.

No capítulo 4, ilustraremos com alguns exemplos de utilização da linguagem Tabulogo.

Para finalizar, no capítulo 5, são apresentadas as conclusões obtidas neste trabalho, bem como algumas possibilidades de trabalhos futuros.

Capítulo 2

Conceitos

O título deste trabalho é "Tabulogo: linguagem de programação baseada em Logo para software de Geometria Dinâmica". Para compreender melhor este título, primeiro é necessário esclarecer que o nome Tabulogo é uma composição dos termos Tabulæ e Logo. Mais que isso, Tabulogo é uma ferramenta que une elementos de Logo e Geometria Dinâmica, tentando utilizar o melhor de cada. O objetivo deste capítulo é abordar estes conceitos presentes no Tabulogo, a fim de elucidá-los e preparar terreno para o entendimento de como o trabalho foi desenvolvido.

2.1 Geometria Dinâmica

Geometria Dinâmica (GD) é um conceito de geometria implementada em um programa de computador que permite que objetos geométricos sejam construídos e posteriormente movidos (utilizando, em geral, o mouse), mantendo todos os vínculos estabelecidos inicialmente na construção. Isso permite que o usuário teste conjecturas e procure descobrir propriedades a partir destas construções.

Por exemplo, em um software de GD, podemos criar uma reta a e uma reta b definida como paralela à reta a. Ao mover a reta a, a reta b também se move paralelamente a a, para manter o vínculo pré-estabelecido.

Vale lembrar também, que um software de GD difere totalmente de um software gráfico matricial, como o *Microsoft Paint*. Um software matricial apenas armazena a informação de cor sobre cada *pixel* no desenho (utilizando para isso uma matriz de *pixels*) enquanto o software de GD guarda informações

vetoriais sobre cada objeto criado e sobre a relação entre esses objetos. Isso torna o desenvolvimento desse tipo de software razoavelmente mais complexo.

Um software de Geometria Dinâmica também não é apenas um software gráfico vetorial. Apesar de também guardar informações vetoriais sobre cada objeto, um formato vetorial, como o SVG, não possui primitivas que estabeleçam relações entre objetos. Além disso, um programa gráfico convencional também não possui um "estatuto geométrico", isto é, algo que mantenha as propriedades geométricas estabelecidas. Mas, como o software de GD possui as informações vetoriais sobre os objetos, uma construção geométrica de Geometria Dinâmica pode ser facilmente convertido para um formato vetorial.

Os primeiros softwares relevantes de GD foram o Cabri e o Sketchpad. A primeira versão do software Cabri foi lançada em 1988 por um grupo de pesquisa francês liderado por Jean-Marie Laborde [6]. Também criado na década de 80, o Sketchpad é fruto de um projeto de pesquisa para criação de novos materiais para ensino de geometria. Este projeto foi desenvolvido no Swarthmore College (EUA) sob a tutela de Nicholas Jackiw [13]. Existem outros softwares desenvolvidos mais recentemente, como o Geogebra que se destaca por, além da geometria, ter uma ênfase algébrica bastante desenvolvida.

A Geometria Dinâmica se destacou como tecnologia educacional para o ensino de Matemática [21], atraindo a atenção de grupos de pesquisa no mundo todo. Esse destaque se deve, principalmente, por possibilitar: a precisão e visualização (das construções), explorações e descobertas, transformações e lugares geométricos, prova de teoremas e simulação de micromundos [14].

2.2 Tabulæ

O Tabulæ é um software de Geometria Dinâmica desenvolvido no Projeto Enibam do IM/UFRJ. O desenvolvimento do software teve início em 1999 e continua em aprimoramento até hoje. Participaram deste projeto diversos alunos da graduação dos cursos de Engenharia, Matemática e Computação, além de alunos de mestrado e doutorado.

Entre as principais funcionalidades do Tabulæ, podemos destacar o núcleo matemático bem de-

senvolvido, com diversos tipos de objetos, desde retas, pontos, e círculos a cônicas e *loci*. Também foram implementados vários tipos de transformações como homotetia e projetividade. Possui também uma calculadora, onde é possível manipular grandezas referentes aos objetos. Além disso, o software possui interface ergonômica (que é separada do núcleo matemático) e é capaz de exportar figuras (nos formatos SVG, PS, GIF, JPG e WMF) e gerar relatórios sobre o uso dos alunos.

A partir do Tabulæ, surgiram outros projetos relacionados. Entre eles, o Tabulæ Colaborativo, um ambiente de aprendizagem colaborativa onde os usuários podem fazer construções geométricas conjuntamente. Outro projeto é o Tabulinha, uma versão do software voltado ao público infantil de 9 a 14 anos. E finalmente, temos o Tabulogo, o objeto de estudo em questão.

2.3 Logo

Logo é uma linguagem de programação derivada de LISP, idealizada em 1967, voltada para aprendizado de crianças. Inicialmente, a idéia não vingou devido aos altos custos dos computadores na época e inviabilizava o uso dos computadores em sala de aula. Na década de 80, Seymour Papert disseminou o Logo através do livro "Mindstorms: Children, computers, and powerful ideas" [19], e com a popularização dos computadores, Logo ganhou fervorosos adeptos.

A base ideológica do Logo é apoiada no construtivismo educacional, uma vertente ideológica que prega que a criança não deve ser um mero repositório de conhecimento e sim montar o conhecimento por conta própria. De acordo com Papert: "Knowledge is only one part of understanding. Genuine understanding comes from hands-on experience" [8]. Muitas crianças tiveram seu primeiro contato com o computador através desta linguagem.

A idéia por trás do Logo é simples. Trata-se de um ambiente virtual em que podemos comandar uma tartaruga, geralmente representada com um ícone, que recebe instruções de movimento e desenho.

Logo tem uma abordagem diferente da geometria cartesiana convencional. Os movimentos da tartaruga não são orientados a coordenadas absolutas e sim a comandos (por exemplo, `forward` e `right`, vide a figura 2.1). Isto significa que ao invés de usar uma equação para descrever a trajetória

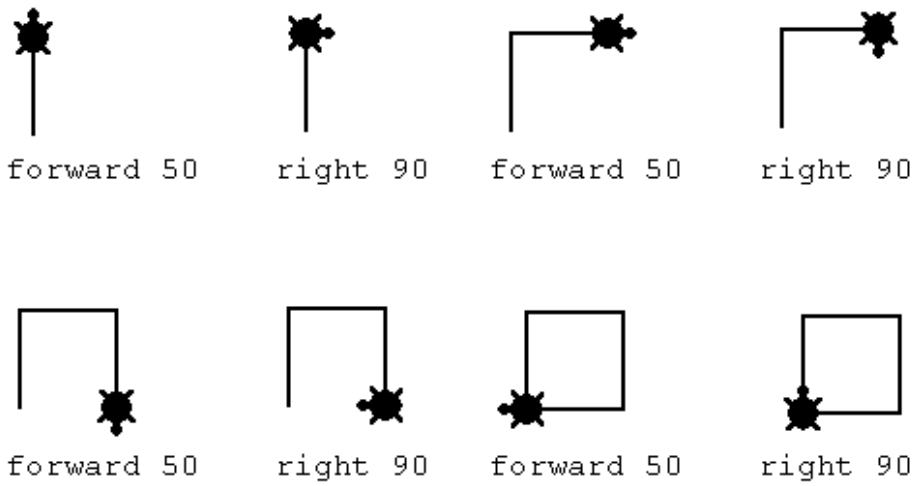

Figura 2.1: Exemplo de comandos Logo [16].

da tartaruga, usaremos um conjunto de comandos. Por isso, podemos dizer que a representação de uma figura é mais local que global [7]. Além disso, em Logo puro, não é possível manipular as figuras depois de sua criação, da mesma maneira que programas gráficos matriciais como o *Microsoft Paint*.

Com o passar do tempo, Logo foi evoluindo. Foram criados diversos dialetos de Logo, e cada um deles foi incorporando novas funcionalidades para atender seu público. Foram adicionadas cores e suporte a gráficos 3D, programação orientada a objetos, múltiplas tartarugas e concorrência, além de elementos de interface com o usuário [12]. Uma lista bem detalhada de implementações de Logo foi documentada pelo *Logo Tree Project* e pode ser encontrada em [5].

Assim como a Geometria Dinâmica, Logo também é um assunto constantemente pesquisado. Tanto do ponto de vista pedagógico, através de pesquisas que tentam medir a eficácia pedagógica do Logo [23], até pesquisas em Computação Gráfica que comparam a recursão em Logo com transformações afins iterativas, provando sua equivalência [11]. Em [7], teoremas são provados usando a chamada "Geometria da Tartaruga" e existe até uma discussão sobre a Teoria da Relatividade Geral sob a ótica do Logo!

Ao falar de Logo e Geometria Dinâmica, não podemos deixar de mencionar Boxer [22], um descendente de Logo, que além das funcionalidades tradicionais, possui um ambiente de programação

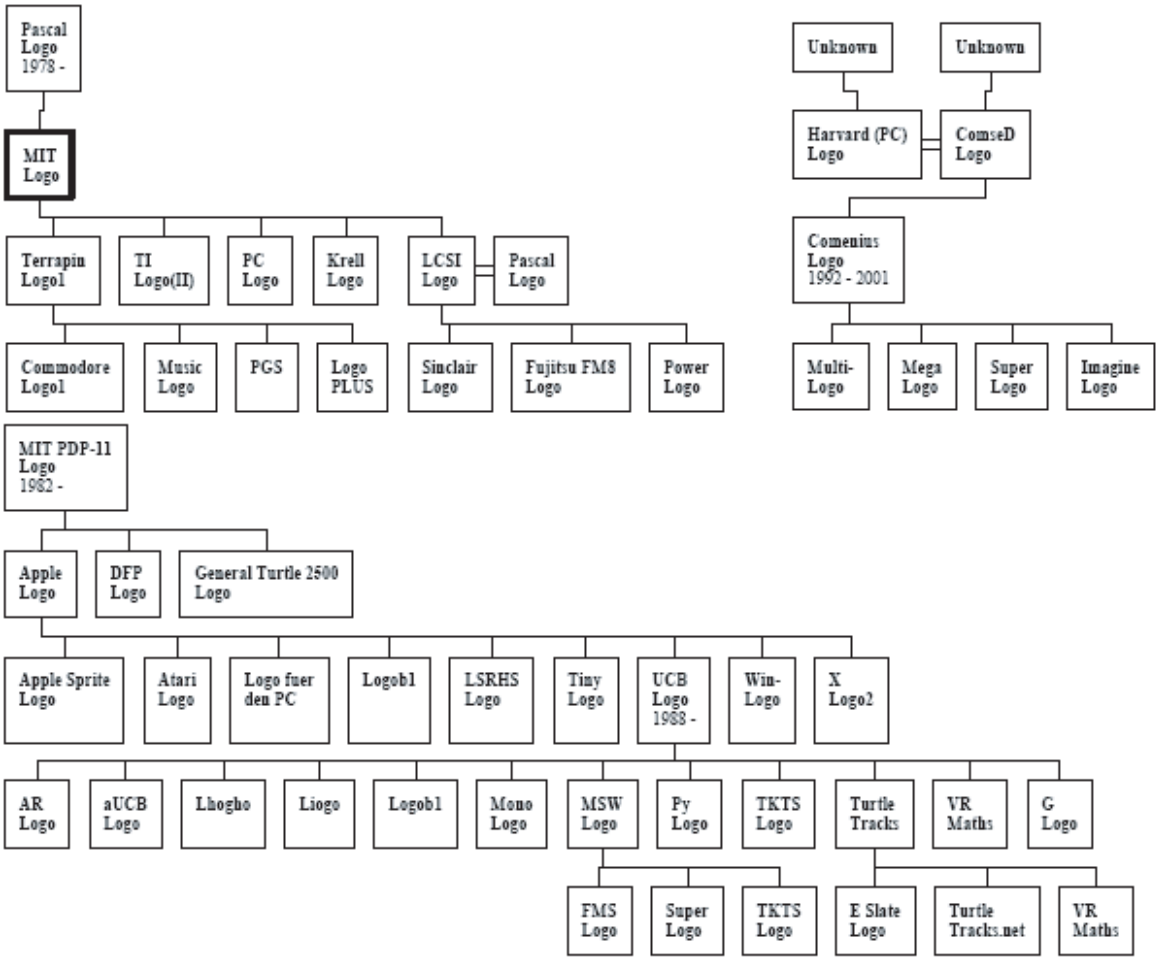

Figura 2.2: Árvores genealógicas da linguagem [4].

estruturado em caixas (*boxes*, daí o nome) e também procurou incorporar elementos de Geometria Dinâmica. No entanto, possui a desvantagem de que a cada mudança no contexto da construção é necessário a executar do algoritmo novamente. Essa abordagem difere do Tabulogo. No Tabulogo partimos de um ambiente de Geometria Dinâmica incorporando elementos de Logo, e por isso os objetos acompanham automaticamente a mudança de contexto na construção.

2.4 Linguagens de Programação e Compiladores

Uma linguagem de programação é um conceito abstrato. Comandos e primitivas podem ser descritos formalmente e não necessitam de um computador para existir. No entanto, para implementar uma linguagem de programação como o Tabulogo, é necessário um programa que reconheça o código-fonte, interprete os comandos e execute ações baseadas nessas interpretações.

Todavia, antes de nos preocuparmos em como implementar uma linguagem de programação, devemos resolver o problema de como definir essa linguagem. Aho *et al*. [1, p. 12] descrevem a seguinte abordagem para resolver este problema:

> Uma linguagem de programação pode ser definida pela descrição da aparência de seus programas (a sintaxe da linguagem) e do que os mesmo significam (a semântica da linguagem). Para especificar a sintaxe, apresentamos uma notação amplamente aceita, chamada de gramática livre de contexto ou BNF (para a Forma de Backus-Naur).

Mas antes de definirmos uma gramática livre de contexto (GLC), é necessário revisar alguns conceitos básicos.

2.4.1 Produção, Token, Variáveis

Em uma GLC, um dos conceitos chaves são os *tokens* ou símbolos terminais. Os *tokens* são seqüências de caracteres válidas na linguagem e que possuem um valor semântico. Eles são as unidades atômicas de uma linguagem.

Variáveis ou símbolos não-terminais, servem para denotar um conjunto de *tokens* e/ou variáveis. Além disso, instituem uma relação hierárquica de como *tokens* e outras *variáveis* estão relacionadas.

Uma GLC é formada por *produções*. Uma *produção* é uma regra sintática, isto é, algo que descreve um comportamento específico da linguagem. As *produções* são compostas por *tokens* e *variáveis* e definem como eles podem ser dispostos. Conseqüentemente, um conjunto de *produções* define quais programas são aceitos ou não pela gramática.

Aho *et al*. [1, p. 13] utilizam o seguinte exemplo para esclarecer como os conceitos de *Produção*, *Token* e *Variáveis* estão relacionados:

> $cmd \rightarrow$ **if** $(expr)$ *cmd* **else** *cmd*
>
> onde a seta deve ser lida como "pode ter a forma". Tal regra é chamada de *produção*. Numa produção, os elementos léxicos, como a palavra-chave if e os parênteses, são chamados de *tokens*. As variáveis como expr e cmd representam seqüências de emphtokens e são chamadas de não-terminais.

Este exemplo ilustra a definição da produção do comando *if* presente na maioria das linguagens de programação.

2.4.2 Gramáticas Livres de Contexto

De acordo com Aho *et al*. [1, p. 13], uma gramática livre de contexto possui quatro componentes:

1. Um conjunto de tokens, conhecidos como símbolos terminais.
2. Um conjunto de não-terminais.
3. Um conjunto de produções, onde uma produção consiste em um não-terminal, chamado de lado esquerdo da produção, uma seta e uma seqüência de de tokens e/ou não terminais, chamado de lado direito da produção.
4. Uma designação a um dos não-terminais como símbolo de partida

E o principal motivo para se construir uma GLC é que "a sintaxe das construções de uma linguagem de programação pode ser descrita pelas gramáticas livres de contexto ou pela notação BNF (Forma de Backus-Naur)." [1, p. 72] Vale lembrar que, a notação BNF nada mais é que uma forma

padrão de representar uma GLC. A gramática livre de contexto da linguagem Tabulogo pode ser encontrada em [17].

2.4.3 Análise do Programa Fonte

Uma vez definida a linguagem, devemos nos ater em como essa linguagem será implementada em um computador. Para isto, é necessário construir um compilador ou interpretador. Sua tarefa será ler um programa escrito na linguagem fonte, no nosso caso o Tabulogo, e gerar uma estrutura de dados que possa ser utilizada para realizar as operações definidas no programa fonte. Diferentemente dos compiladores tradicionais, no caso do Tabulogo, não há necessidade de gerar uma linguagem alvo, já que ela é uma linguagem interpretada. A motivação de a linguagem ser interpretada será explicada melhor na seção 3.5.

Para transformar um programa fonte que, em última análise, é simplesmente uma seqüência de caracteres, em uma estrutura de dados válida e com significado, é necessário executar uma série de análises sobre o texto de entrada. Classicamente, a análise do programa consiste em três etapas, a saber: análise *léxica*, análise *sintática* e análise *semântica*.

Mais uma vez, conforme Aho *et al*., "o analisador léxico é a primeira fase de um compilador. Sua tarefa principal é a de ler os caracteres de entrada e produzir uma seqüência de *tokens* que o *parser* utiliza para a análise sintática." [1, p. 38]. Ou seja, o analisador léxico lê caractere por caractere, formando *tokens*, que repassa para o analisador sintático (Figura 2.3).

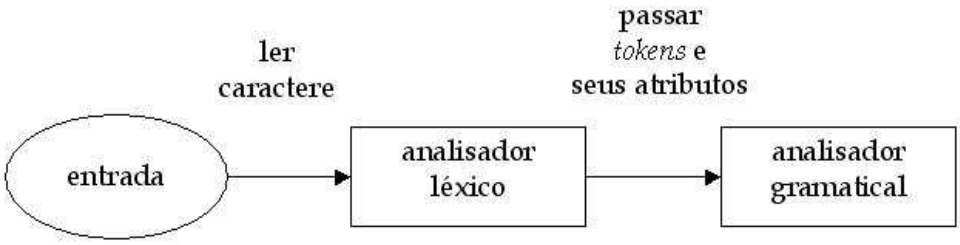

Figura 2.3: Função de um analisador léxico (baseado em [1, p. 25]).

Os *tokens* são definidos através de expressões regulares. “As expressões regulares são uma notação importante para especificar padrões. Cada padrão corresponde a um conjunto de cadeias e, dessa forma as expressões regulares servirão como nomes para conjuntos de cadeias.” [1, p. 42]. Vale lembrar que, para determinar se uma determinada cadeia é descrita por uma expressão regular basta verificar se a mesma é aceita pelo autômato finito determinístico (AFD) gerado a partir da expressão regular.

$$(0 \bigcup 1 \bigcup ... \bigcup 9)\,(0 \bigcup 1 \bigcup ... \bigcup 9)^*$$
sendo o alfabeto, $\Sigma = \{0, 1, ..., 9\}$.

Figura 2.4: Exemplo de expressão regular que denota os inteiros positivos aceitos por uma linguagem de programação.

A análise *sintática* ou *gramatical* é responsável por montar a *árvore gramatical*, uma estrutura de dados que contém informações hierárquicas sobre os *tokens* que compõem o programa. É através da árvore gramatical que o programa alvo é sintetizado ou que operações são executadas (no caso de linguagens interpretadas como o Tabulogo). Analisadores sintáticos constroem a árvore sintática e também são conhecidos como *parsers*. Os dois principais métodos de análise sintática serão discutidos na seção 2.4.4.

A análise *semântica* consiste em verificar se ”os componentes do programa se combinam de uma forma significativa” [1, p. 2]. Um dos principais componentes da análise semântica é a verificação de tipos. A verificação de tipos garante que operações sejam realizadas apenas pelos tipos apropriados. Ela não permite e adverte o usuário, por exemplo, caso ele tente executar uma divisão de uma variável do tipo inteira por uma *string*. Dependendo da linguagem, a análise semântica pode ser realizada em tempo de compilação ou execução.

2.4.4 Analisadores sintáticos *Bottom-up* e *Top-down*

Aho *et al.* [1, p. 18] descrevem diferentes métodos de análise sintática:

> A maioria dos métodos de análise gramatical cai em uma dentre duas classes, chamadas

de *top-down* e *bottom-up*. Esses termos se referem à ordem na qual os nós da árvore gramatical são construídos. No primeiro, a construção se inicia na raiz e prossegue em direção às folhas, enquanto que no último, a construção se inicia nas folhas e procede em direção à raiz. A popularidade dos analisadores gramaticais *top-down* é devida ao fato de que analisadores eficientes podem ser construídos mais facilmente à mão utilizando-se métodos *top-down*. A análise *bottom-up*, entretanto, pode manipular uma classe mais ampla de gramáticas e esquemas de tradução e, então, as ferramentas de software para gerar de analisadores gramaticais, diretamente a partir das gramáticas, tendem a usar os métodos *bottom-up*.

Para implementar os dois métodos e suas variantes, diferentes tipos de algoritmos são empregados. Detalhar estes algoritmos foge do escopo desse trabalho, mas uma descrição minuciosa pode ser encontrada em [1].

2.4.5 Gramáticas LL, LR e LALR

Em geral, um método de análise sintática define uma classe de gramáticas aceitas por ele. Ou seja, de acordo com o método de análise sintática utilizado, diferentes tipos de gramáticas serão aceitas pelo analisador sintático.

Entre os métodos de análise sintática *top-down*, a análise LL(1) tem destaque e define uma classe de gramáticas homônima, como descrito por Aho *et al.* [1, p. 85]:

> Uma gramática cuja tabela sintática não possui entradas multiplamente definidas é dita LL(1). O primeiro "L" em LL(1) significa a varredura da entrada da esquerda para direita (*left to right*); o segundo, a produção de uma derivação mais à esquerda (*left linear*); e o "1", o uso de um único símbolo de entrada como *lookahead* a cada passo para tomar decisões sintáticas.

Esmiuçar cada detalhe desta descrição também está fora do escopo deste trabalho. O interessante para nós é que "as gramáticas LL(1) possuem várias propriedades distintivas. Nenhuma gramática ambígua ou recursiva à esquerda pode ser LL(1)." [1, p. 85].

Esta classe de gramáticas será importante neste trabalho pois a linguagem Tabulogo é definida através de uma gramática LL(1). Por isso, tivemos que nos adaptar às restrições oferecidas por esta classe de gramáticas.

Já entre as técnicas *bottom-up*, a técnica LR se destaca e é descrita por Aho *et al*. [1, p. 94]:

> ... uma técnica eficiente de análise sintática *bottom-up*, que pode ser usada para decompor uma ampla classe de gramáticas livres de contexto. A técnica chamada análise sintática LR(k); o "L" significa varredura da entrada da esquerda para a direita (*left-to-right*), o"R", construir uma derivação mais à direita ao contrário (*rightmost derivation*) e o *k*, o número de símbolos de entrada de *lookahead* que são usados ao se tomar decisões na análise sintática.

Um aprimoramento da técnica LR é a técnica LALR (*lookahead* LR), comentada por Aho *et al*. [1, p. 95]:

> Este método é frequentemente usado na prática porque as tabelas obtidas são consideravelmente menores do que as tabelas LR canônicas e, além do mais, a maioria das construções sintáticas comuns das linguagens de programação pode ser expressa convenientemente por gramáticas LALR.

A classe de gramáticas LALR é mais geral que a LL, isto é, o conjunto de gramáticas LL é subconjunto de LALR. No entanto, a classe de gramáticas LL é bastante expressiva e pode ser utilizada sem problemas para especificar uma linguagem de programação como o Tabulogo, como veremos na seção 3.2.

2.4.6 *Parsers* Recursivos Descendentes

Um caso especial de analisador sintático *top-down* (*parser*) é o *Parser Recursivo Descendente*. Este tipo de parser será importante neste trabalho pois o parser da linguagem Tabulogo é recursivo descente. "A *análise gramatical descendente recursiva* é um método *top-down* de análise sintática, no qual executamos um conjunto de procedimentos recursivos para processar a entrada." [1, p. 20]. Em

outras palavras, em um parser recursivo descendente cada produção na BNF da gramática se torna uma subrotina.

Ao mencionar *parsers* recursivos descendentes, não podemos deixar de falar sobre recursão à esquerda. Recursão à esquerda ocorre em uma gramática "quando o símbolo mais à esquerda do lado direito é o mesmo que o não-terminal no lado esquerdo da produção" Aho *et al.* [1, p. 21]. Na descrição dos *parsers* recursivos descendentes, a recursão à esquerda é proibida para prevenir que as subrotinas geradas se chamem recursivamente *ad-infinitum*.

Parsers recursivos descendentes são restritos às gramáticas LL(1) [27]. Repare que, de fato, a propriedade de nenhuma gramática recursiva à esquerda pode ser LL(1) é atendida.

2.5 Geradores Automáticos de *Parsers*

Um gerador automático de *parsers* é um programa de computador que recebe como entrada uma gramática em uma determinada sintaxe e gera um programa como saída. Este programa gerado é o *parser*. Um gerador de *parsers* facilita imensamente o trabalho de um desenvolvedor de uma linguagem de programação como o Tabulogo, pois programar um *parser* por conta própria pode ser uma tarefa árdua, dependendo da complexidade da linguagem em questão. Para se ter uma idéia, o Fortran, que foi desenvolvido antes da teoria dos geradores de automáticos de *parsers*, exigiu 18 homens-ano para ser desenvolvido, conforme [3] *apud* [1, p. 2].

Existem diversos geradores automáticos de *parsers*. Entre os principais, estão o YACC (*Yet Another Compiler-Compiler*) e o LEX. Eles trabalham em conjunto para gerar um compilador, sendo que o LEX gera o analisador léxico e o YACC gera o analisador sintático. O YACC converte uma descrição de uma GLC LALR(1) em um programa em C que reconhece esta gramática através de uma análise *bottom-up*.

Neste trabalho, utilizamos o equivalente do YACC e LEX para o Java, o Javacc [24], que será discutido em detalhes no próximo capítulo.

Capítulo 3

Implementação

Agora que analisamos os conceitos envolvidos no Tabulogo, podemos mostrar como eles foram utilizados neste trabalho. Além disso, neste capítulo vamos apresentar as ferramentas escolhidas para a construção do Tabulogo, mostrando suas principais características. Apresentamos também as etapas de elaboração da linguagem, desde sua concepção. Por fim, discutimos algumas características do Tabulogo enquanto linguagem de programação.

3.1 Java

Java foi a linguagem de programação escolhida para implementar o Tabulogo. Além de ser uma linguagem robusta, *open source* (desde 2006) e orientada a objetos, o motivo principal pelo qual foi adotada, é que o Tabulogo teria que interagir diretamente com o software Tabulæ, que foi escrito em Java. Portanto, escolher a mesma linguagem facilitaria bastante o desenvolvimento e evitaria possíveis problemas de incompatibilidade entre linguagens diferentes.

Além disso, outra característica importante de Java (e um dos motivos pelo qual o Tabulæ ter sido escrito em Java) é ser multi-plataforma. Isso permite que o Tabulogo seja utilizado em qualquer ambiente, independentemente de arquitetura do computador ou sistema operacional. Isso contribui para a universalização do acesso ao software nas escolas (muitas utilizam Linux) e atende os propósitos pedagógicos de ambos os softwares.

3.2 Javacc

O Javacc (*Java Compiler Compiler*) [24] é um gerador automático de *parsers* e foi escolhido principalmente porque gera código em Java, o que facilita a integração do *parser* com o Tabulæ (também escrito em Java). Além disso, o Javacc também foi desenvolvido em Java (compatível com a versão 1.1 e superiores) e portanto é multi-plataforma. Javacc também possui um *plug-in* para o Eclipse [15] que facilita bastante o desenvolvimento. Outra vantagem, é que seu código é livre desde 2003.

A descrição sintática e léxica é feita em um mesmo arquivo com extensão `.jj`. Isto significa que a descrição dos *tokens* em expressões regulares (descrição léxica) e a regras sintáticas estão juntas em apenas um arquivo. Isto torna a gramática mais fácil de ser lida e mantida. Ao submeter o arquivo `.jj` ao Javacc, ele gera os seguintes arquivos:

- SimpleCharStream.java - representa um *stream* de caracteres de entrada.
- Token.java - representa um único *token* de entrada.
- TokenMgrError.java - representa um erro lançado pelo gerenciador de *tokens*.
- ParseException.java - representa uma exceção indicando que o programa de entrada não está de acordo com a gramática do *parser*. Um exemplo deste será mostrado mais adiante.

Além de arquivos personalizados, sendo XXX o nome do *parser*:

- XXX.java - a classe principal do parser.
- XXXTokenManager.java - o gerenciador de *tokens* do parser.
- XXXConstants.java - uma interface associando *tokens* aos nomes simbólicos.

Além disso, o Javacc provê automaticamente resposta aos erros de sintaxe. Os *parsers* gerados pelo Javacc são capazes de detectar o erro de sintaxe lançar uma exceção (*ParseException*), detalhando a nível de *token* qual foi o erro encontrado e localizando-o através da linha e coluna do código "parseado".

Segue um código que provoca erro de compilação (falta o ponto e vírgula para delimitar o fim do comando):

```
a = b
```

O *parser* exibe como saída a mensagem da Listagem 3.1.

Listagem 3.1: Saída de erro do *parser*.

```
br.ufrj.labma.enibam.parser.ParseException: Encountered "<EOF>"
at line 1, column 5.
Was expecting one of:
    "[" ...
    ";" ...
    ">" ...
    "<" ...
    "==" ...
    "<=" ...
    ">=" ...
    "!=" ...
    "+" ...
    "-" ...
    "*" ...
    "/" ...
```

O Javacc também possui uma ferramenta para documentação, o JDoc. Esta ferramenta utiliza o arquivo com de especificação da gramática (arquivo com extensão .jj) como entrada e gera um arquivo HTML com a EBNF (Extended Backus-Naur Form) da gramática. Neste, arquivo HTML é possível navegar entre os símbolos não terminais através de *hyperlinks*, e assim compreender mais facilmente a estrutura da linguagem. Um exemplo de arquivo HTML gerado pode ser encontrado em [17].

Uma diferencial do Javacc em relação a outros geradores automáticos de *parsers* é que ele já traz nativos alguns conceitos presentes nas linguagens de programação. Por exemplo, a opção `SKIP` para declarar quais caracteres devem ser descartados no momento da compilação como espaços e quebras de linha. As opções `SPECIAL_TOKEN` e `MORE` servem para declarar *tokens* utilizados nos

comentários. Os demais *tokens* são declarados com a opção `TOKEN`. Estas opções possibilitam uma especificação mais clara da gramática e permitem mensagens de erro e avisos mais precisos.

Outra característica digna de nota, é que o Javacc permite utilizar a notação EBNF (Extended Backus-Naur Form). Essa extensão introduz elementos de expressões regulares dentro de uma gramática livre de contexto. Por exemplo, a expressão $(A)*$ serve para denotar 0 ou mais repetições de A e a expressão $(A)+$ denota uma ou mais repetições de A, sendo A uma variável. Esta notação confere legibilidade à gramática, visto que é muito mais simples entender uma regra do tipo $A ::= y(x)*$ que $A ::= Ax|y$.

Em relação à análise sintática, o Javacc gera *parsers* recursivos descendentes que, como dito no capítulo anterior, realizam análise *top-down*. Apesar deste tipo de análise só aceitar gramáticas mais restritivas (gramáticas LL(1)), já que não suportam recursão à esquerda, a análise *top-down* tem uma série de vantagens. Além disso, podemos evitar a recursão à esquerda utilizando os elementos da EBNF, convertendo regras do tipo $A ::= Ax|y$ para $A ::= y(x)*$.

Entre as vantagens da análise *top-down* mais exploradas neste trabalho estão a facilidade para depurar o parser gerado, pois em qualquer ponto de uma análise sempre sabemos qual produção está sendo derivada, e principalmente, a habilidade de passar atributos (valores) na árvore gramatical tanto de baixo para cima como de cima para baixo. Esta habilidade permitiu que os objetos que representam cada primitiva da linguagem sejam instanciados ao longo análise e sejam passados como parâmetro na instanciação de objetos de nível mais elevado na árvore gramatical. Essa estrutura será melhor detalhada na seção 3.4.

3.2.1 Gramática Tabulogo

Para implementar a gramática do Tabulogo, utilizamos uma gramática do Java (gramáticas de diversas linguagens de programação no formato Javacc podem ser encontradas em [25]) como base. Reaproveitamos a descrição léxica de números inteiros, decimais, strings, operadores e identificadores (descrição de *tokens* para nomes de variáveis ou funções). A parte de comentários também foi reutilizada. A partir disto, foram acrescidos os *tokens* com as palavras reservadas da linguagem Tabulogo.

```
< #DECIMAL_LITERAL: ["0-"9"] (["0-"9"])* >
```

descreve a seguinte expressão regular:

$$(0 \bigcup 1 \bigcup ... \bigcup 9)(0 \bigcup 1 \bigcup ... \bigcup 9)^*$$
sendo o alfabeto, $\Sigma = \{0, 1, ..., 9\}$.

Figura 3.1: Descrição de expressão regular no Javacc.

Listagem 3.2: Trecho da gramática com definições de palavras reservadas do Tabulogo.

```
TOKEN :
{
  < REPITA: "Repita" >
| < VEZES: "Vezes" >
| < LENUMERO: "LeNumero" >
| < ROTACAO: "Rotacao" >
| < SEGMENTO: "Segmento" >
| < DESLOCA: "Desloca">
| < PONTO: "Ponto" >
| < INICIO: "Inicio" >
| < FIM: "Fim" >
```

Na parte sintática, nada pode ser reutilizado, já que Java e Tabulogo têm estruturas completamente distintas. As regras sintáticas foram sendo inseridas incrementalmente conforme havia necessidade de novas primitivas.

Para inserir o comando `Repita` por exemplo, a seguinte regra sintática foi utilizada:

No bloco (1) (figura 3.2) são feitas declarações das referências aos objetos Java utilizados durante a execução da regra sintática. O bloco (2) é onde a regra sintática é efetivamente declarada. Os *tokens* (`Repita`, `Vezes`, `Inicio` e `Fim`) são posicionados convenientemente e outras regras sintáticas são referenciadas como `LeExpressao()` e `LeBloco()`.

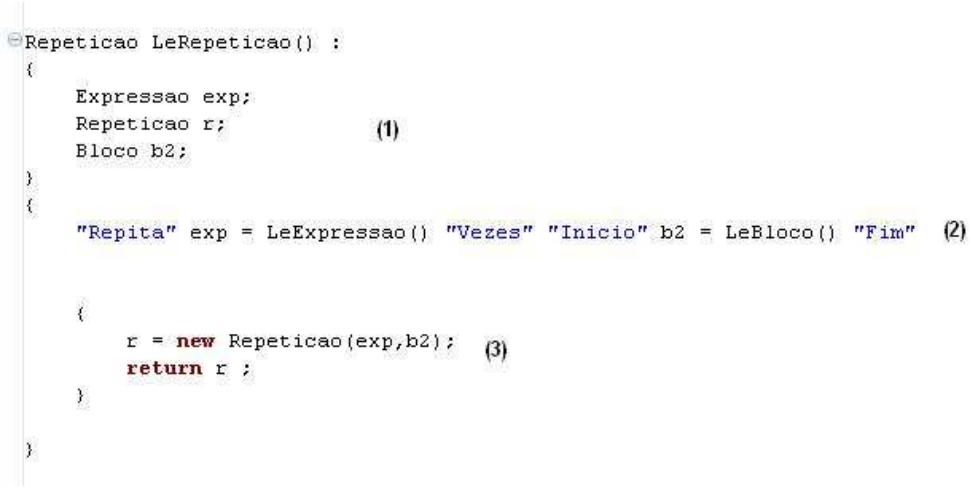

Figura 3.2: Produção do comando `Repita` na gramática do Tabulogo.

O bloco (3) é um trecho de código Java que será executado quando a regra for derivada. Neste caso, o objeto é apenas instanciado e retornado para a regra sintática superior, que referenciou a regra `LeRepeticao()`.

3.3 Padrões de Projeto

Freqüentemente em programação, utilizamos padrões de projeto. Padrões de projeto descrevem soluções clássicas para problemas usuais [9]. Durante a implementação do Tabulogo, alguns desses problemas clássicos se apresentaram e recorremos a alguns padrões de projeto para solucioná-los.

3.3.1 Padrão *Command*

Um dos problemas enfrentados foi trabalhar com objetos que eram comandos. Isto é, cada primitiva da linguagem executa uma ação no Tabulæ e isto deveria ser representado e armazenado de alguma forma. Além disso, era necessário criar uma estrutura que padronizasse a forma de execução de cada comando. Para resolver este problema, existe o padrão *Command*.

Gamma *et al*. [9, p. 223] definem o padrão *Command* da seguinte maneira:

> The Command pattern lets toolkit objects make requests of unspecified application ob-

jects by turning the request itself into an object. This object can be stored and passed around like other objects. The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation. Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing Execute to invoke the request. The receiver has the knowledge required to carry out the request.

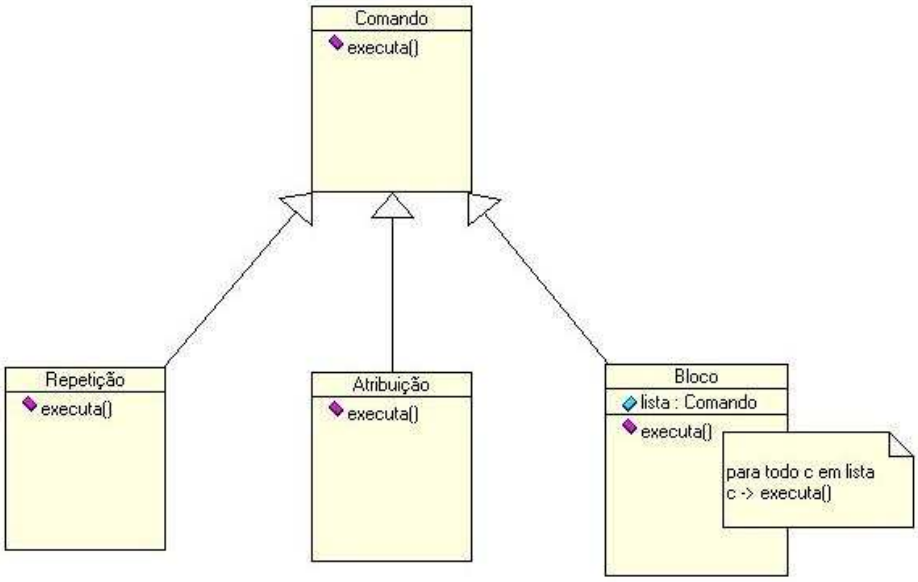

Figura 3.3: Diagrama de classes exemplificando como o padrão Command foi implementado no Tabulogo.

Note que no diagrama de classes da figura 3.3 todas as classes representadas (`Repetição`, `Atribuição`, `Bloco`) herdam da classe abstrata `Comando`. Além disso, todas elas tem sua própria implementação do método `executa()`. Note também que, apesar de não ser exatamente um comando no sentido literal, `Bloco` também é uma subclasse de `Comando`. Isso se deve, pois um `Bloco` possui uma lista de comandos, e sua implementação do método `executa()` itera sobre essa lista, e cada comando executa sua própria implementação do método `executa()`.

Portanto, este padrão sugere uma estrutura na qual cada objeto saiba se executar e por isso todos objetos de sua classe compartilham uma mesma função de execução.

Este padrão ainda tem uma vantagem de ser de fácil integração com a gramática do Javacc. Essa integração é facilitada já que a cada regra executada um novo comando é instanciado e enviado para a regra que o chamou. Por isso, o padrão *Command* atendeu bem a seu propósito e mostrou a importância de se utilizar padrões de projeto.

3.3.2 Padrão *Façade*

Outro problema enfrentado no projeto do Tabulogo, foi que possuíamos um grande sistema legado, o Tabulæ com inúmeras classes e métodos. E precisávamos de um meio de utilizar seu núcleo geométrico para criar retas, pontos, círculos, etc... sem interferir diretamente no Tabulæ.

Gamma *et al*. [9, p. 175] sugerem o uso do padrão *Façade* (Fachada) justamente quando: "... you want to provide a simple interface to a complex subsystem ... there are many dependencies between clients and the implementation classes of an abstraction ... you want to layer your subsystems.".

A sugestão de uso deste padrão foi aceita pois, também existem muitas dependências entre as primitivas do Tabulogo e o Tabulæ (muitas delas refletem diretamente as primitivas do Tabulæ). E era desejado que criássemos uma camada de abstração que separasse as duas aplicações, até para que houvesse independência no desenvolvimento dos softwares.

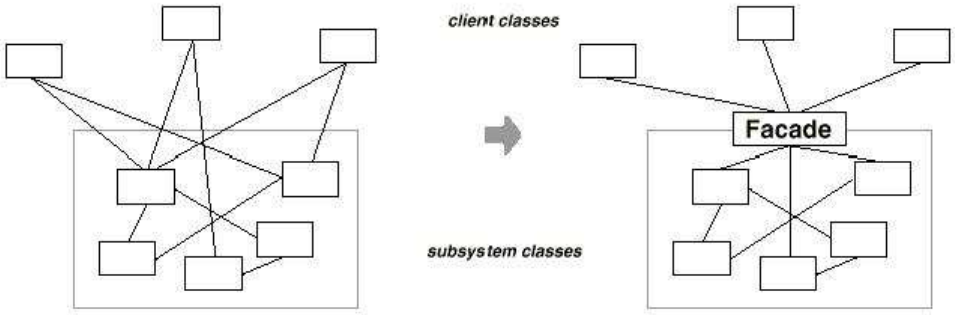

Figura 3.4: Arquitetura do padrão *Façade*. No caso, o cliente é o Tabulogo que faz as requisições aoTabulæ, que atua como servidor.

Para implementar o padrão *Façade* é necessário definir um ponto de entrada no sistema legado. No nosso caso, criamos a classe `ControleTabulae` que faz o contato direto com o Tabulæ, chamando suas máquinas de criação de objetos geométricos. Isto pode ser visto na figura 3.5.

```
public GraphicLine criaReta(GraphicPoint p1, GraphicPoint p2) {

    LineCreationMachine lcm = new LineCreationMachine();
    LineCreationMachine m = (LineCreationMachine)lcm.CreateObject(vp.getTM());
    return (GraphicLine) criaObjeto(m, new GraphicObject[](p1,p2));
}
```

Figura 3.5: Trecho da classe ControleTabulae em que encapsulamos a criação de uma reta.

O próximo passo foi a criação da classe de Fachada propriamente dita, a `FachadaTabulae`. Note que esta classe não faz referência direta às máquinas de criação do Tabulæ. Além disso, os parâmetros e retornos de seus métodos são apenas os *ids* (inteiro que identificam unicamente) dos objetos. Com isso, conseguimos construir uma interface para o Tabulæ e utilizá-la com transparência.

```
public Object reta(int id, int id2) {

    GraphicPoint go1 = (GraphicPoint) ct2.getObjeto(id);
    GraphicPoint go2 = (GraphicPoint) ct2.getObjeto(id2);
    GraphicLine go3 = ct2.criaReta(go1, go2);
    return go3.getID();
}
```

Figura 3.6: Trecho da classe `FachadaTabulae`. Ela utiliza o `ControleTabulae` através da referência `ct2`.

Outro exemplo de utilização do padrão *Façade* é a própria API do Java. Ela esconde os detalhes de implementação, por exemplo, de como são implementadas as janelas e caixas de diálogo do Swing. Com isso, o usuário não se preocupa com o sistema operacional nem a arquitetura da máquina. Tudo que ele precisa conhecer é a API do Java e qual método é responsável por renderizar o objeto gráfico desejado.

3.4 Arquitetura Tabulogo

A implementação da linguagem Tabulogo está presente em diversas camadas que vão desde o *parser* até a interação com o Tabulæ. Nesta seção, pretendemos detalhar essas camadas através de sua arquitetura, mostrando como o Tabulogo atua desde a escrita de um programa até sua execução.

Inicialmente, o usuário utiliza um editor de texto simples para escrever um programa em Tabulogo. Para fins de testes, utilizamos o editor Jext [10], um editor escrito em Java. Contudo, o uso do Tabulogo não está atrelado a nenhum editor específico. Adaptamos o Jext para que fosse possível invocar o Tabulæ a partir dele, e assim permitir maior integração entre a edição e a execução dos programas. Também foram utilizadas algumas funcionalidades do Jext para facilitar a programação em Tabulogo, como *code highlighting* e inserção automática de código.

Depois do programa escrito, o parser do Tabulogo é invocado. Ele recebe como entrada o texto do editor e o compila. Se o programa for compilado com sucesso, um objeto da classe `Programa` será instanciado. O objeto da classe `Programa` conterá referências para todos os objetos declarados no código e será utilizado posteriormente para a execução do programa. Caso haja algum erro de compilação, uma `ParseException` será lançada e uma mensagem de erro semelhante a listagem 3.1 será exibida para o usuário.

A etapa de execução ocorre logo após a de compilação. Os comandos contidos no objeto da classe `Programa` serão executados recursivamente, de acordo com o padrão *Command* (vide seção 3.3.1). Os comandos que utilizam as primitivas do Tabulæ, invocam os métodos da classe `FachadaTabulae`, que por sua vez invocam os métodos de `ControleTabulae`, conforme a nossa implementação do padrão de projeto *Façade* (vide seção 3.3.2).

Durante a execução do programa, atribuições de valores a variáveis são realizadas. Este valores são armazenados em uma *hashtable*, que mapeia os nomes das variáveis (*strings*) nos valores propriamente ditos. No caso dos objetos geométricos, apenas o *id* (um inteiro identificador atribuído pelo Tabulæ) é associado à variável, mas é o suficiente para referenciá-los posteriormente. Além disso, as informações de estado da tartaruga do Logo (posição e orientação) também são armazenadas em memória.

Depois da execução do programa, o Tabulæ se torna independente do Tabulogo e o usuário pode

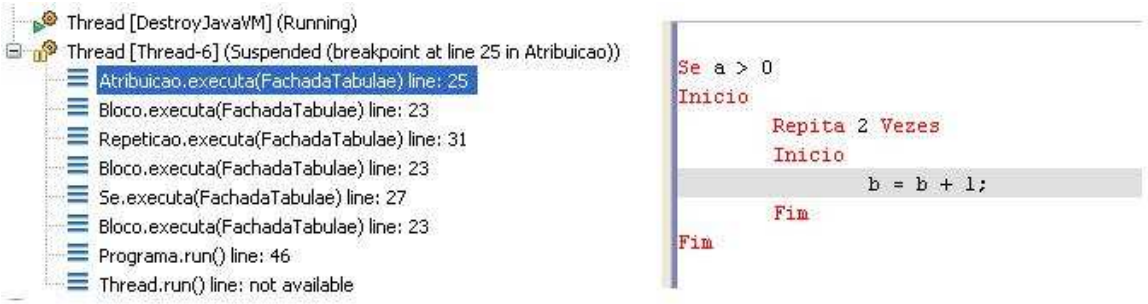

Figura 3.7: Depuração no Eclipse durante a execução da linha do programa em destaque (em cinza). A *stack* de métodos invocados reflete a estrutura de classes contidas na classe `Programa`.

manipular livremente os objetos geométricos criados. O usuário poderá também executar outros programas em Tabulogo ou salvar a construção em um arquivo `.ae` (arquivo Tabulæ).

3.5 Características da Linguagem

Agora, vamos analisar o Tabulogo do ponto de vista das linguagens de programação. Desde a concepção da linguagem, sempre tentamos obedecer aos princípios básicos do Logo e primar pela simplicidade.

Apesar de Logo ser uma linguagem derivada de LISP, que é uma linguagem funcional, o Tabulogo é uma linguagem predominantemente procedimental e estruturada. Até porque, uma das principais motivações da linguagem é poder utilizar as estruturas de repetição e condicionais dentro de um programa de GD.

A linguagem também se caracteriza por ser interpretada. Isto se deve ao fato que ao invés de produzir um programa alvo como resultado da compilação, o parser gera um objeto Java que representa o programa. Este objeto é interpretado e realiza as operações especificadas no código fonte.

Características como a tipagem fraca foram mantidas, isto é, não é necessário declarar explicitamente o tipo de uma variável e este tipo pode mudar durante a execução do programa. Os tipos suportados pelo Tabulogo são: inteiros, reais (*double*) e *strings*.

Em relação às funções, fizemos algumas modificações. Na maioria das implementações de Logo,

não é possível passar parâmetros para uma função ou receber valores retornados por funções e utilizam apenas variáveis globais. Apesar de o entendimento sobre escopo de variáveis não ser trivial, optamos por implementá-lo na linguagem principalmente para facilitar o uso da recursão. No Tabulogo, a princípio, todas as variáveis são globais, exceto os parâmetros de funções, mas podemos declarar explicitamente a variável como local através da sentença `Local nomeDaVariavel`.

Tabulogo tem suporte a listas que também fazem as vezes de vetores. É possível declarar listas explicitamente (`Lista nomeDaLista`) ou através de atribuições (`nomeDaLista = {1,2,3}`). As listas também são utilizadas quando uma primitiva Tabulæ cria mais de um objeto, e as referências aos objetos criados são retornadas em uma lista.

Capítulo 4

Exemplos

Neste capítulo, ilustraremos com alguns exemplos da utilização da linguagem Tabulogo. Através deles será possível conhecer em mais detalhes as principais características da linguagem, bem como algumas de suas decisões de projeto. Desta forma, pretendemos trilhar os caminhos que estabeleceram a atual forma da linguagem.

4.1 Rosácea

Antes de pensar em explorar as possibilidades de unir Geometria Dinâmica e Logo, primeiramente era necessário garantir um ambiente plenamente compatível com o Logo. Portanto, o primeiro desafio foi reproduzir um programa apenas com primitivas Logo. Para isso, precisávamos de um programa relativamente simples para dar o passo inicial e ver os primeiros resultados. Mas também, era necessário que utilizasse as principais características Logo. Por isso, escolhemos um dos programas vencedores de um concurso do site Mathcats [20] que trazia o seguinte desafio: "Escreva um programa em Logo de uma linha com 15 ou menos palavras, sem contar colchetes e parênteses, para produzir a mais bela, complexa e interessante figura". O programa escolhido foi o seguinte:

```
repeat 8 [rt 45 repeat 6 [repeat 90 [fd 2 rt 2] rt 90]]
```

Note que neste programa, o número 6 (parâmetro do segundo `repeat`) pode ser substituído por números de 1 a 7, alterando a forma da rosácea. Além disso, segundo o próprio autor do programa,

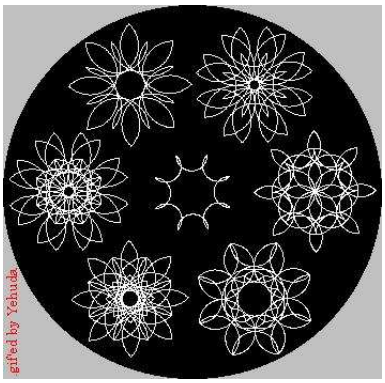

Figura 4.1: O desenho foi chamado de “Dahlia” pelo autor americano David Eisenstat.

este programa funciona na maioria das implementações de Logo. A “tradução” deste programa para Tabulogo é encontrada na Listagem 4.1.

Listagem 4.1: Programa para desenhar uma rosácea em Tabulogo.

```
Repita 8 Vezes
Inicio
Direita(45);
        Repita 6 Vezes
        Inicio
                Repita 90 Vezes
                Inicio
                        AndaSegmento(20);
                        Direita(2);
                Fim
                Direita(90);
        Fim
Fim
```

As maiores diferenças são em relação aos colchetes que serviam para delimitar os blocos de comandos e foram substituídos pelos *tokens* `Inicio` e `Fim` para facilitar a compreensão da sintaxe, principalmente dos não familiarizados com linguagens de programação. Outra diferença é que o comando RT(2) (mnemônico de *Right Turn*) foi traduzido para Direita(2), ou seja, roda a orientação da tartaruga em 2 graus para a direita.

Por fim, o comando FD (mnemônico de *Move Forward*) que faz a tartaruga se deslocar para frente e desenhar sua trajetória (caso sua “caneta” esteja ativada) foi substituído por `AndaSegmento`. Este

comando também altera a posição da tartaruga e cria um segmento de reta (objeto do Tabulæ) entre os pontos inicial e final da trajetória.

Este programa foi importante, pois serviu de prova de conceito do Tabulogo, mostrando seus primeiros resultados (Figura 4.2) e a compatibilidade com o Logo. Isto é evidenciado pela adaptação das primitivas de desenho e movimentação da tartaruga. Além disso, este exemplo começou a moldar a linguagem e apontar os caminhos para fazer a ponte entre Logo e a Geometria Dinâmica.

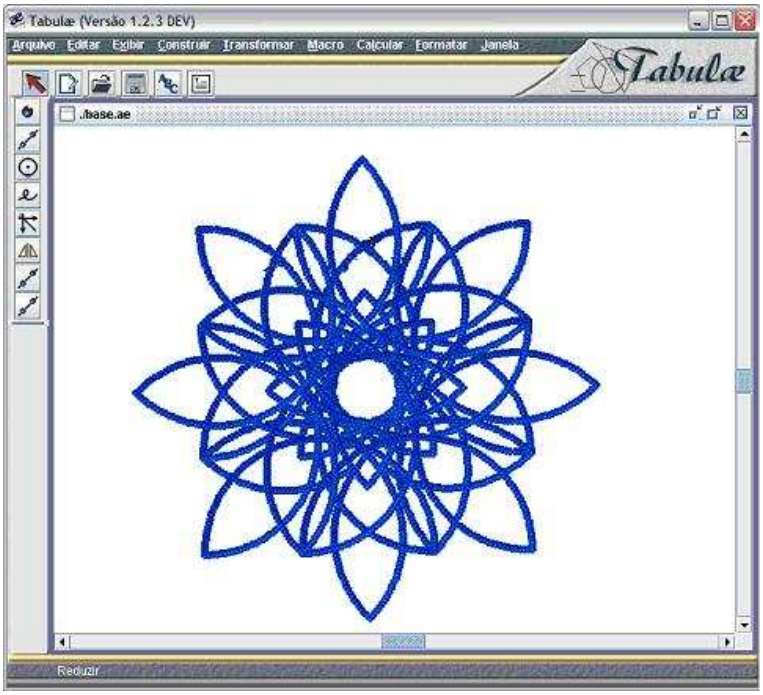

Figura 4.2: Rosácea gerada através do Tabulogo.

4.2 Planificação do Cone

Neste exemplo exploraremos a estrutura de repetição da linguagem na construção da planificação de um cone através de sua épura. O código fonte deste programa em Tabulogo consta no apêndice A.

A épura é uma técnica de representação geométrica em 2D para formas em 3D, utilizando a projeção ortogonal do objeto tridimensional. Já a planificação é um modo de representar em um plano 2D, todas as faces de um poliedro. Basicamente, esta construção faz a conversão da épura para

a planificação. Trata-se de uma construção inerentemente iterativa.

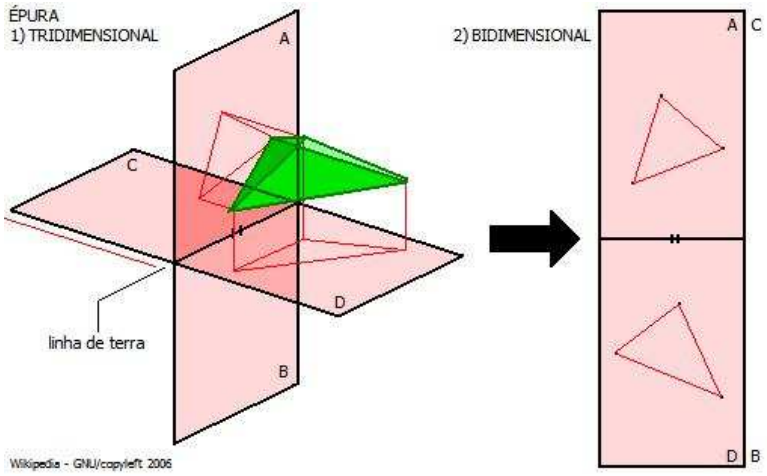

Figura 4.3: Exemplo da representação de um sólido em épura [26].

A entrada desta construção é o número n de iterações em que será feita a aproximação do cone em uma pirâmide. A partir dai, é aplicada uma função que leva um ponto da base da épura em um ponto da planificação. Repare que esta função depende de um ponto da base épura e do ponto da planificação construído em um passo anterior.

Além disso, neste exemplo contamos com o comando `Seleciona` que é utilizado para selecionar, em tempo de execução, objetos já criados previamente no Tabulæ e que podem ser utilizados como entrada nas construções programadas em Tabulogo. Esta possibilidade confere interatividade à construção. Com isso, conseguimos selecionar os objetos que representam a épura e servem de base para o cone planificado.

As listas também são exploradas neste exemplo. Elas são utilizadas para armazenar o conjunto de pontos criados na divisão do círculo da base do cone, através do comando `DivideEmSegmentos`. Em um segundo momento, essa lista de pontos é percorrida para criação de ângulos que serão utilizados na construção da planificação.

Este programa pode ser muito útil no estudo de desenho geométrico, já que a épura pode ser perturbada e vemos em tempo real o impacto na planificação. Mais que a planificação de um determinado cone, esta construção representa a planificação de todos os cones possíveis de serem representados

pela épura.

Este é um exemplo típico de construção que não seria possível (para um n grande)de ser feita utilizando apenas o mouse em um software de geometria dinâmica.

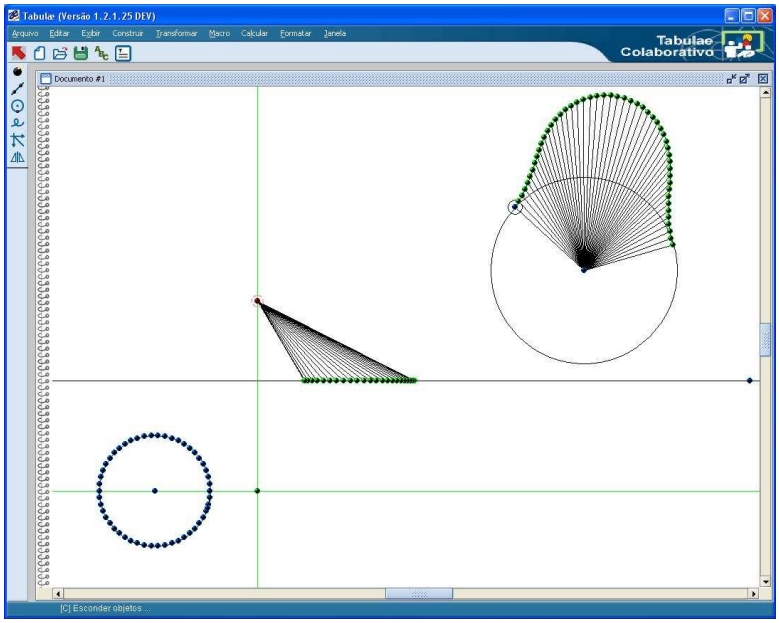

Figura 4.4: Planificação do cone no Tabulogo. A esquerda, a épura e a direita, a planificação.

4.3 Curva de Koch

Neste exemplo iremos explorar a recursão utilizando o Tabulogo através da construção da Curva de Koch. O código fonte deste programa em Tabulogo está no apêndice B.

A Curva de Koch, concebida em 1904, é um exemplo de curva contínua sem derivada em nenhum ponto que pode ser construída de maneira elementar. Sua elaboração depende do uso de ferramentas que possibilitem a construção recursiva de triângulos eqüiláteros no terço central de cada lado construído na etapa anterior.

A função `Koch` se auto-referencia 4 vezes, uma para cada trecho da curva. Nesta função, é passado como parâmetro um valor r que é o nível de recursão. A base da recursão é atingida quando o valor de r chega a 0. Note que a primitiva `Local` é utilizada para especificar o escopo de algumas

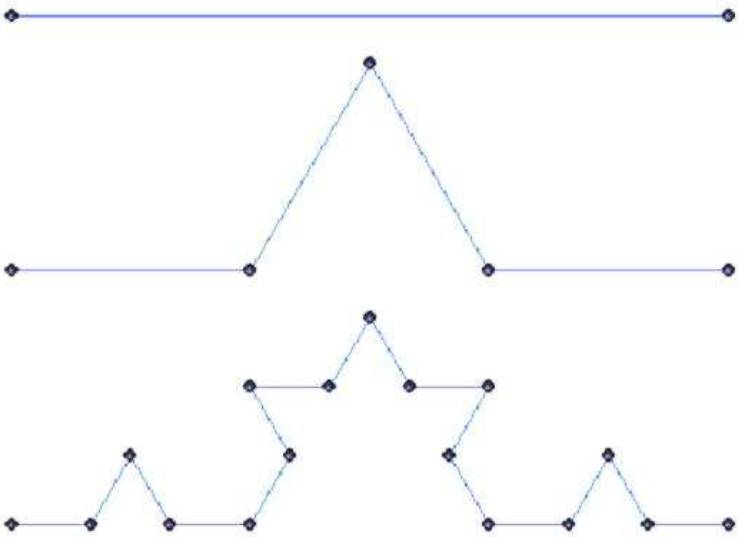

Figura 4.5: Os primeiros passos para elaboração da Curva de Koch.

variáveis dentro da função `Koch`.

Esta construção permite expandir e contrair a figura através da movimentação pontos selecionados inicialmente. Além disso, mesmo com a manipulação dos pontos livres, as propriedades da figura são preservadas como mostrado na figura 4.6.

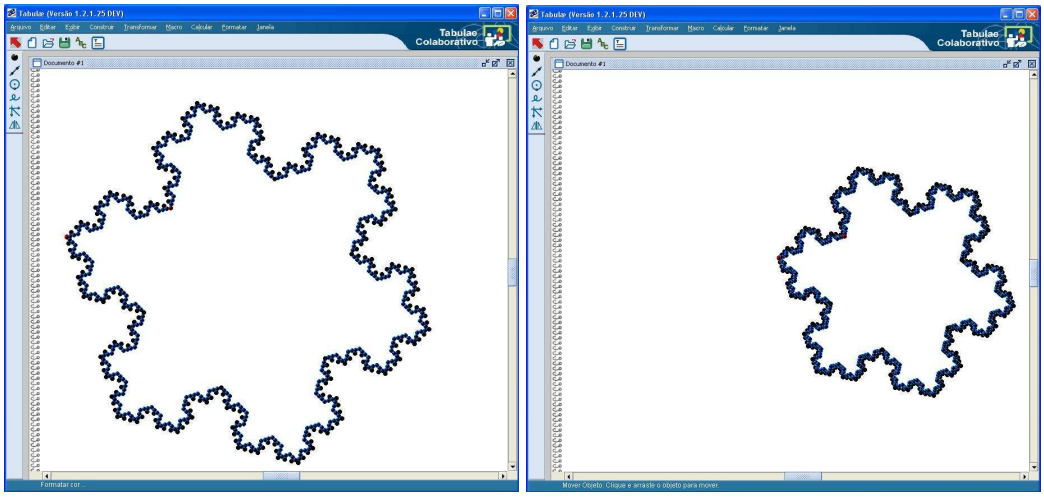

Figura 4.6: Curva de Koch gerada com o Tabulogo.

4.4 Tangentes e Círculos

Neste exemplo, vamos explorar a estrutura condicional (`Se`) da linguagem. O código fonte completo deste programa em Tabulogo consta no apêndice C.

A construção da tangente comum a dois círculos dados (Figura 4.7), $C1$ e $C2$ é muito difícil de ser executada com macros de mouse e teclado, já que são poucos os programas de Geometria Dinâmica que apresentam a possibilidade de construções condicionais. Isto ocorre em razão da construção demandar o conhecimento de qual dos dois círculos possui o menor diâmetro.

Listagem 4.2: Trecho do código do apêndice C.

```
c1 = Seleciona("Selecione o círculo C1:");
c2 = Seleciona("Selecione o círculo C2:");
r1 = Raio(c1);
r2 = Raio(c2);
Se r1 > r2
Inicio
        maior = c1;
        menor = c2;
Fim
Senao
Inicio
        maior = c2;
        menor = c1;
Fim
centroMenor = CentroCirculo(menor);
centroMaior = CentroCirculo(maior);
v1 = Vetor(centroMenor,centroMaior);
c3 = Translacao(v1,menor);
```

Ao observar o trecho de código da listagem 4.2, percebemos que toda a construção está pautada na

decisão de qual dos dois círculos possui o raio maior (`Se r1 > r2`). Esta decisão altera a sentido do vetor `v1` (que é sempre do menor para o maior) e qual deve ser o círculo transladado (círculo menor). Como as demais objetos dependem do círculo transladado, toda construção é afetada.

Contudo, depois de executado o programa, os raios dos círculos podem ser alterados e o estado de qual é o maior ou menor círculo pode mudar. Isto torna a construção inválida, já que as retas criadas deixam de ser tangentes. O Tabulogo nada pode fazer porque depois da execução do programa não está mais atuando. Isto parece violar o princípio da Geometria Dinâmica em que propriedades da construção são mantidas. Na verdade, a Geometria Dinâmica continua prevalecendo, já que as propriedades e relações dos objetos continuam mantidas. O problema ocorre porque a condição do `Se` só é avaliada uma vez, em tempo de execução do programa.

Para tratar essa questão, chegamos a pensar em tornar o Tabulogo orientado a eventos, no entanto, isto complicaria a sintaxe e iria de encontro aos ideais básicos da linguagem. Uma outra possível solução, é criar um objeto *if* no *kernel* do Tabulæ que seria responsável por monitorar sua condição e executar as ações a cada mudança de estado. Porém, esta solução vai na contramão das outras primitivas presente no Tabulogo, que foram desenvolvidas inteiramente sem serem intrusivas no código do Tabulæ.

4.5 Integração Numérica

Neste último exemplo, mesclamos elementos de repetição e condicionais para fazer a integração numérica de funções. Este exemplo também mostra a integração do Tabulogo com a calculadora do Tabulæ, responsável por manipular grandezas dos objetos geométricos como áreas e comprimentos. O código fonte deste programa em Tabulogo está no apêndice D.

A integração numérica consiste em achar um valor aproximado para uma integral definida sem ter que realizar a integração analiticamente. Existem muitos métodos para fazer a integração numéricas, entre eles, o método dos trapézios. Em linha gerais, o método do trapézio calcula o valor aproximado da integral através do somatório das área de trapézios sob o gráfico da função a ser integrada. E exatamente isto que o programa em Tabulogo faz.

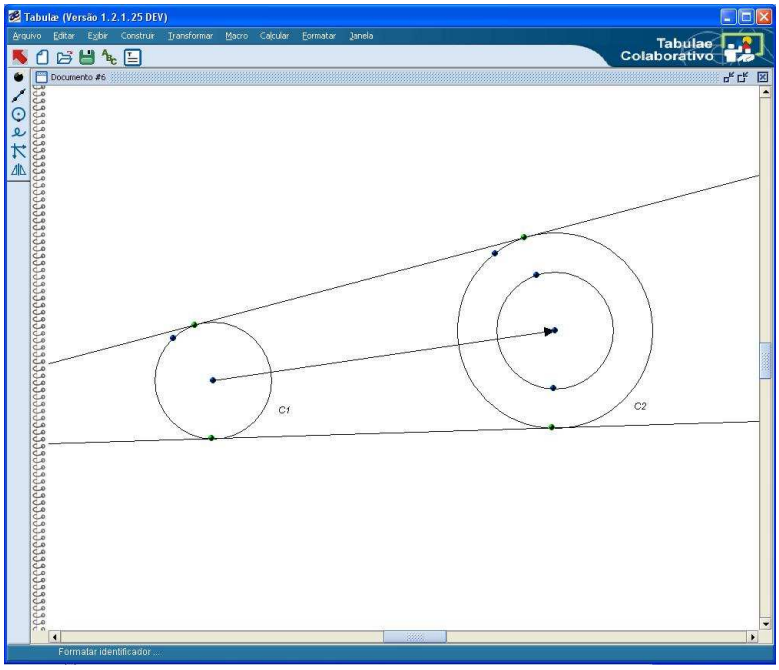

Figura 4.7: Tangente comum a dois círculos criada através de um programa Tabulogo.

As entradas deste programa são:

1. os pontos inicial e final da integral
2. as razões a, b, c
3. a expressão com a função a ser integrada
4. o número de iterações

Esta construção só é possível pela integração do Tabulogo com a calculadora do Tabulæ. Primeiramente, o comando `LeExpressaoCalculadora` exibe uma a interface gráfica de entrada que recebe a função a ser integrada. Esta função pode utilizar rótulos de expressões já criadas no Tabulæ(por exemplo, a, b e c). Depois disso, a calculadora do Tabulæentra em cena, e constrói um objeto expressão, que é dinâmico e atualiza seu valor a cada mudança dos valores que o compõe. O somatório de todas as áreas dos trapézios também é construído utilizando essa expressão dinâmica.

Depois dos trapézios criados e a aproximação da integral calculada, a construção permite que os parâmetros a, b e c sejam alterados, modificando a função dinamicamente. Também é possível alterar

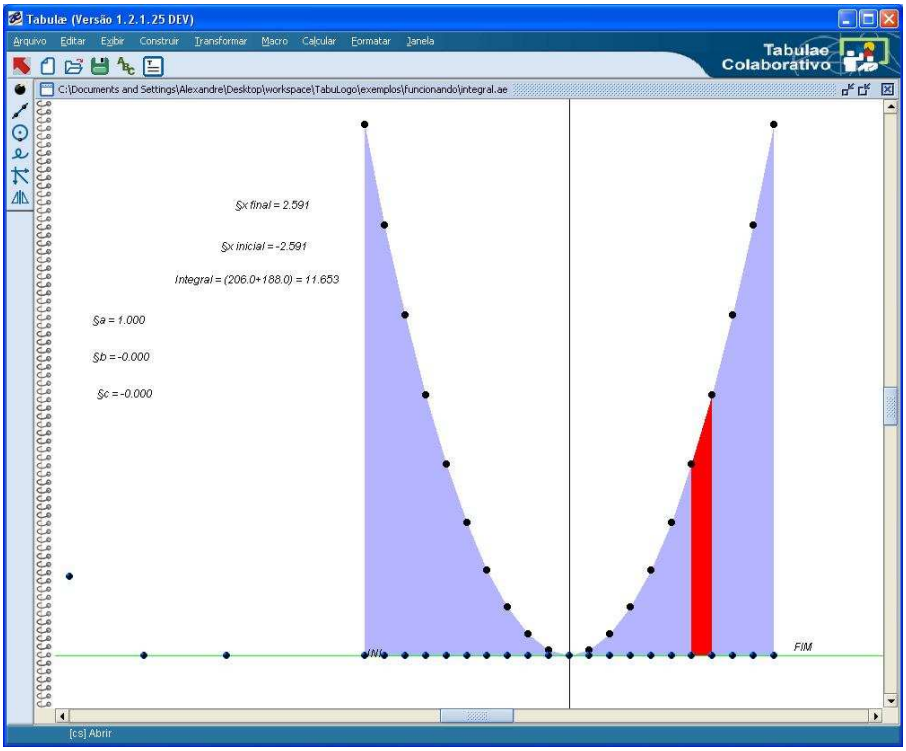

Figura 4.8: Método dos trapézios aplicado a função f(x) = $ax^2 + bx + c$, sendo a=1, b=0, c=0.

o intervalo da integral, perturbando os pontos inicial e final. Com isso, temos o valor da aproximação da integral de diversas funções e em diversos intervalos.

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Conclusões

A primeira conclusão a que chegamos é que, de fato, a utilização da linguagem permitiu a elaboração de construções mais flexíveis e de maior complexidade. Isso se deve principalmente à introdução das primitivas de repetição e controle condicional como o *if*, além da possibilidade do uso da recursão. Nesse sentido, a linguagem cumpriu bem seu papel e atendeu a sua motivação.

Além disso, o Tabulogo trouxe aos usuários familiarizados com Logo, a possibilidade de gerar construções com dependências geométricas, sem precisar aprender uma linguagem completamente diferente. Vale lembrar que o Tabulogo não é um simples dialeto de Logo, pois, além de implementar as primitivas da "Geometria da Tartaruga", ele incorpora as primitivas da Geometria Dinâmica e permite a manutenção das propriedades das construções, o que não é possível no Logo.

Logo e GD se assemelham, no sentido de proporcionar micromundos em que o usuário pode explorar e construir utilizando idéias matemáticas [2]. Ao unir esses dois conceitos, podemos explorar o melhor de cada ferramenta. Mais que isso, essa união também visa suprir suas deficiências. Por exemplo, uma das críticas feitas ao Logo é que para se aprender geometria com ele, é necessário já possuir uma boa bagagem geométrica. O Tabulogo, por ser uma linguagem que já traz conceitos geométricos explícitos ao invés de apenas comandos de desenho, torna mais fácil o aprendizado de Geometria.

Por fim, pudemos constatar que a fusão de programação e Geometria Dinâmica também cria novas maneiras de representar conhecimento matemático. Fazer a tradução da linguagem geométrica para comandos na programação é algo relativamente natural e ainda traz o benefício de introduzir uma notação compacta. Assim, conseguimos representar construções geométricas como programas.

5.2 Trabalhos futuros

É importante salientar que o Tabulogo ainda está em estágio em desenvolvimento, mas uma versão alfa pode ser obtida em [18]. Para se tornar uma versão 1.0, ainda temos que melhorar a resposta ao usuário de erros em tempo de execução. Também é necessário corrigir alguns *bugs*, melhorar o desempenho de algumas primitivas e fazer uma integração completa com a última versão do Tabulæ. Além disso, temos que resolver a questão da manutenção da propriedades da estrutura condicional, citado no exemplo da seção 4.4. Isto será resolvido, provavelmente, criando um objeto condicional no *kernel* do Tabulæ.

Como trabalhos futuros, podemos citar a criação de *plug-ins* ou bibliotecas para a linguagem. A estrutura do Tabulogo foi construída justamente para permitir futuras extensões, bastando apenas encapsular os subsistemas a serem integrados. Assim, conseguiríamos integrar o Tabulogo com diversos softwares das mais diferentes naturezas. Um *plug-in* interessante seria fazer uma integração com um CAS (*Computer Algebra System*), como o Maxima ou Matlab, e usufruir de todo o ferramental algébrico já implementado neles. Utilizaríamos o Tabulæ para a exibição gráfica de resultados. Isto abriria portas, inclusive, para a construção de uma interface visual para auxiliar na prova automática de teoremas.

Outra extensão do trabalho seria desenvolver uma interface amigável para depurar programas em Tabulogo. Atualmente é possível executar um programa passo-a-passo. No entanto, é desejável integrar o controle da execução do programa com uma IDE , facilitando a depuração para o usuário. Além disso, seria interessante implementar funcionalidades que permitissem inspecionar valores de variáveis e expressões.

É claro que uma grande prova de fogo da linguagem será quando houver uma experiência com

alunos. Só assim teremos uma idéia mais concreta da receptividade do Tabulogo e receberemos um *feedback* para aprimorar a linguagem. Também haverá trabalho para professores e interessados em educação matemática na produção de roteiros didáticos e na descoberta de melhores práticas na utilização o Tabulogo como ferramenta educacional.

Ainda podemos citar como um trabalho futuro, quando a linguagem atingir certo grau de maturidade, a possibilidade de se programar o próprio Tabulæ com a linguagem Tabulogo. Seria uma espécie de meta-programação. Com isso, um usuário leigo em Java poderia programar novas funcionalidades para o Tabulæ utilizando apenas conceitos geométricos.

Lista de Siglas

API Application Programming Interface

BNF Backus-Naur Form

CAS Computer Algebra System

EBNF Extended Backus-Naur Form

GD Geometria Dinâmica

GLC Gramática Livre de Contexto

HTML HyperText Markup Language

IDE Integrated Development Environment

Javacc Java Compiler-Compiler

YACC Yet Another Compiler-Compiler

Lista de Figuras

2.1 Exemplo de comandos Logo [16]. 7

2.2 Árvores genealógicas da linguagem [4]. 8

2.3 Função de um analisador léxico (baseado em [1, p. 25]). 11

2.4 Exemplo de expressão regular que denota os inteiros positivos aceitos por uma linguagem de programação. 12

3.1 Descrição de expressão regular no Javacc. 20

3.2 Produção do comando `Repita` na gramática do Tabulogo. 21

3.3 Diagrama de classes exemplificando como o padrão Command foi implementado no Tabulogo. 22

3.4 Arquitetura do padrão *Façade*. No caso, o cliente é o Tabulogo que faz as requisições aoTabulæ, que atua como servidor. 23

3.5 Trecho da classe ControleTabulae em que encapsulamos a criação de uma reta. . . . 24

3.6 Trecho da classe `FachadaTabulae`. Ela utiliza o `ControleTabulae` através da referência `ct2`. 24

3.7 Depuração no Eclipse durante a execução da linha do programa em destaque (em cinza). A *stack* de métodos invocados reflete a estrutura de classes contidas na classe `Programa`. 26

4.1 O desenho foi chamado de “Dahlia” pelo autor americano David Eisenstat. 29

4.2 Rosácea gerada através do Tabulogo. 30

4.3 Exemplo da representação de um sólido em épura [26]. 31

4.4 Planificação do cone no Tabulogo. A esquerda, a épura e a direita, a planificação. . . 32
4.5 Os primeiros passos para elaboração da Curva de Koch. 33
4.6 Curva de Koch gerada com o Tabulogo. 33
4.7 Tangente comum a dois círculos criada através de um programa Tabulogo. 36
4.8 Método dos trapézios aplicado a função f(x) = $ax^2 + bx + c$, sendo a=1, b=0, c=0. . . 37

Lista de Listagens

3.1 Saída de erro do *parser*. 18
3.2 Trecho da gramática com definições de palavras reservadas do Tabulogo. 20
4.1 Programa para desenhar uma rosácea em Tabulogo. 29
4.2 Trecho do código do apêndice C. 34

Apêndice A

Código fonte: cone.tlg

```
n = 50;
base = Seleciona("Selecione o círculo como base da épura:");
c = Seleciona("Selecione o centro do círculo como base da épura:");
vl = Seleciona("Selecione o ponto como vértice da épura:");
vertice = Seleciona("Selecione o ponto como vértice da planificação:");
lt = Seleciona("Selecione a linha de terra:");
r1 = RetaPerpendicular(lt,vl);
r2 = RetaParalela(lt, c);
ptint = PontosIntersecao(r1,r2);
v = ptint[1];
pa = RotacaoConstante(v, c, 180);
Esconde(pa);
l = DivideEmSegmentos(base,n);
Lista p;
i = n;
j = 1;
Repita n Vezes
Inicio
```

```
        p[j] = l[i];
        i = i - 1;
        j = j + 1;
Fim
p[n+1] = p[1];
s2 = Segmento(p[1],p[2]);
ang = Angulo(p[1],v,pa);
p1 = Rotacao(v,p[1],ang);
Esconde(p1);
r3 = RetaPerpendicular(r2,p1);
Esconde(r3);
ptint2 = PontosIntersecao(lt,r3);
p2 = ptint2[1];
s = Segmento(vl,p2);
c2 = CirculoCentroSegmento(vertice,s);
p3 = PontoSobreObjeto(c2,0,0);
s = Segmento(vertice,p3);
c3 = CirculoCentroSegmento(p3,s2);
j = 2;
Repita n - 1 Vezes
Inicio
        ang = Angulo(p[j],v,pa);
        p1 = Rotacao(v,p[j],ang);
        Esconde(p1);
        r3 = RetaPerpendicular(r2,p1);
        Esconde(r3);
        ptint2 = PontosIntersecao(lt,r3);
        p2 = ptint2[1];
```

```
        s = Segmento(v1,p2);
        c2 = CirculoCentroSegmento(vertice,s);
        Esconde(c2);
        ptint3 = PontosIntersecao(c2,c3);
        p4 = ptint3[2];
        s = Segmento(vertice,p4);
        Esconde(ptint3[1]);
        s = Segmento(p3,p4);
        c3 = CirculoCentroSegmento(p4,s2);
        Esconde(c3);
        p3 = p4;
        j = j + 1;
Fim
```

Apêndice B

Código fonte: koch.tlg

```
p = Seleciona("Ponto inicial");
q = Seleciona("Ponto final");
r = 3;
Vai Koch(p,q,r);
penultimo = p;
ultimo = q;
lados = 4;
ang1 = 240;
ang2 = 60;
flag = 1;
i = lados - 1;
Repita 11 Vezes
Inicio
        Se flag == 1
        Inicio
        ptemp = RotacaoConstante(ultimo, penultimo, ang1);
                flag = 0;
  Fim
```

```
    Senao
    Inicio
                    ptemp = RotacaoConstante(ultimo, penultimo, ang2);
                    flag = 1;
            Fim
            Vai Koch(ultimo,ptemp,r);
            penultimo = ultimo;
            ultimo = ptemp;
Fim
# Koch(p1,p2,r)
Local s, l, p3;
Se r > 0
Inicio
            s = Segmento(p1,p2);
            l = DivideEmSegmentos(s,3);
            Esconde(s);
            p3 = RotacaoConstante(l[1], l[2], 60);
            Vai Koch(p1,l[1],r-1);
            Vai Koch(l[1],p3,r-1);
            Vai Koch(p3,l[2],r-1);
            Vai Koch(l[2],p2,r-1);
Fim
Senao
Inicio
            s = Segmento(p1,p2);
Fim
Volta
```

Apêndice C

Código fonte: tangente.tlg

```
c1 = Seleciona("Selecione o círculo C1:");
c2 = Seleciona("Selecione o círculo C2:");
r1 = Raio(c1);
r2 = Raio(c2);
Se r1 > r2
Inicio
        maior = c1;
        menor = c2;
Fim
Senao
Inicio
        maior = c2;
        menor = c1;
Fim
centroMenor = CentroCirculo(menor);
centroMaior = CentroCirculo(maior);
v1 = Vetor(centroMenor,centroMaior);
c3 = Translacao(v1,menor);
```

```
rt =  RetaTangenteCirculo(c3,centroMenor);
rs = rt[1];
Esconde(rs[1]);
Esconde(rs[2]);
ps = rt[2];
p4 = ps[1];
p5 = ps[2];
r3 = Reta(centroMaior, p4) ;
r4  = RetaParalela(r3, centroMenor);
Esconde(r3);
Esconde(r4);
v1 = PontosIntersecao(r3,maior);
v2 = PontosIntersecao(r4,menor);
r5 = Reta(centroMaior, p5) ;
r6  = RetaParalela(r5, centroMenor);
Esconde(r5);
Esconde(r6);
v3 = PontosIntersecao(r5,maior);
v4 = PontosIntersecao(r6,menor);
r7 = Reta(v1[1],v2[1]);
r8 = Reta(v3[1],v4[1]);
Esconde(v1[2]);
Esconde(v2[2]);
Esconde(v3[2]);
Esconde(v4[2]);
```

Apêndice D

Código fonte: integral.tlg

```
ini = Seleciona("Ponto Inicial");
fim = Seleciona("Ponto Final");
v = RecuperaObjeto("vetory");
ori = RecuperaObjeto("ptoo");
uni = RecuperaObjeto("ptou");
seg = RecuperaObjeto("segu");
sai1 = RecuperaObjeto("saida1");
sai2 = RecuperaObjeto("saida2");
a = Seleciona("a");
b = Seleciona("b");
c = Seleciona("c");
UnidadeReferencia(seg);
laexpression = LeExpressaoCalculadora("função");
n = LeNumero("Quantas Iterações?");
z = n-1;
w = n+1;
s = Segmento(ini, fim);
ptstemp = DivideEmSegmentos(s,n);
```

```
Lista ptsx, pts, areas;
ptsx[1] = ini;
i = 2;
Repita z Vezes
Inicio
        ptsx[i] = ptstemp[i-1];
        i = i+1;
Fim
ptsx[i] = fim;
i=1;
Repita w Vezes
Inicio
        x = RazaoTresPontos(ori,uni,ptsx[i]);
        Esconde(x);
        Se i == 1
        Inicio
                Mostra(x);
                Rotula(x,"x inicial");
        Fim
        Se i == w
        Inicio
                Mostra(x);
                Rotula(x,"x final");
        Fim
        exp = ExecutaExpressaoCalculadora(laexpression);
        Esconde(exp);
        v2 = ProdutoVetorEscalar(v,exp);
        Esconde(v2);
```

```
        pts[i] = Translacao(v2,ptsx[i]);
        i = i + 1;
Fim
i = 1;
Repita n Vezes
Inicio
        ltemp = {ptsx[i+1], pts[i+1], pts[i], ptsx[i]};
        p = Poligono(ltemp);
        areas[i] = Area(p);
        Esconde(areas[i]);
        i = i + 1;
Fim
i = 2;
c = areas[1];
Repita z Vezes
Inicio
        c = ExpressaoCalculadora(c+areas[i]);
        i = i + 1;
        Esconde(c);
        Se i == w
        Inicio
                Mostra(c);
                Rotula(c,"Integral");
        Fim
Fim
```

Referências Bibliográficas

[1] AHO, A. V., SETHI, R., ULLMAN, J. D., *Compiladores: princípios, técnicas e ferramentas*. Rio de Janeiro, LTC Editora, 1995.

[2] AINLEY, J., PRATT, D., "Dynamic geometry in the Logo spirit", *Micromath. A Journal of the Association of Teachers of Mathematics*, v. 11, n. 1, pp. 19–24, 1995.

[3] BACKUS, J. W., BEEBER, R. J., BEST, S., R.GOLDBERG, HAIBT, L. M., HERRICK, H. L., NELSON, R. A., SAYRE, D., SHERIDAN, P. B., STERN, H., "The FORTRAN automatic coding system", *Wesrm Juinr Computer Conference*, 1957.

[4] BOYTCHEV, P., "All-in-one tree of MIT Logo". Disponível em: <`http://ccgi.frindsbury.force9.co.uk/greatlogoatlas/?download=LogoFamilyTrees.pdf`>. Acesso em Fevereiro 6, 2009.

[5] BOYTCHEV, P., "Logo tree project". Disponível em: <`http://www.elica.net/download/papers/LogoTreeProject.pdf`>. Acesso em Agosto 11, 2008.

[6] CABRILOG Company, "Project Cabri". Disponível em: <`http://www-cabri.imag.fr`>. Acesso em Fevereiro 6, 2009.

[7] DISESSA, A. A., ABELSON, H., *Turtle geometry: the computer as a medium for exploring mathematics*. MA, USA, MIT Press Cambridge, 1981.

[8] DUCASSE, S., *Squeak: learn programming with robots*. Apress, 2005.

[9] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[10] GUY, R., “Jext”. Disponível em: <http://www.jext.org>. Acesso em Fevereiro 6, 2009.

[11] JU, T., SCHAEFER, S., GOLDMAN, R., “Recursive turtle programs and iterated affine transformations”, 2003. Disponível em: <http://citeseer.ist.psu.edu/ju04recursive.html>. Acesso em Fevereiro 6, 2009.

[12] KAHN, K., “Should Logo keep going FORWARD 1?”, *Informatics in Education*, v. 6, n. 2, pp. 307–320, 2007.

[13] KCP Technologies, “The Geometer’s Sketchpad® resource center”. Disponível em: <http://www.dynamicgeometry.com/General_Resources/The_Sketchpad_Story.html>. Acesso em Fevereiro 6, 2009.

[14] KING, R., SCHATTSCHNEIDER, D., *Geometry turned on: dynamic software in learning, teaching, and research*. Cambridge University Press, 1997.

[15] KOUTCHERAWY, R., BEN, Q., MURRAY, P. M., “Javacc Eclipse plug-in”. Disponível em: <http://pagesperso-orange.fr/eclipse_javacc>. Acesso em Fevereiro 6, 2009.

[16] Logo Foundation, “A Logo primer or what’s with the turtles?”. Disponível em: <http://el.media.mit.edu/Logo-foundation/logo/turtle.html>. Acesso em Fevereiro 6, 2009.

[17] MATTOS, A. F. S., “Gramática tabulogo”. Disponível em: <http://www.dcc.ufrj.br/~alexandre/tabulogo/gramatica.html>. Acesso em Fevereiro 6, 2009.

[18] MATTOS, A. F. S., “Tabulogo”. Disponível em: <http://www.dcc.ufrj.br/~alexandre/tabulogo/tabulogo.zip>. Acesso em Fevereiro 6, 2009.

[19] PAPERT, S., *Mindstorms: children, computers, and powerful ideas*. Basic Books, January/1981.

[20] PETTI, W., “Math Cats”. Disponível em: <http://www.mathcats.com/gallery/15wordcontest.html>. Acesso em Fevereiro 6, 2009.

[21] SCHUMANN, H., "The influence of interactive tools in geometry learning.", In: *Intelligent learning environments. The case of geometry.* (LABORDE, J. M., ed.), pp. 157–187, Berlim, Springer, 1996. NATO ASI Series. Series F. Computer and Systems Sciences, v. 117.

[22] SHERIN, B., "Representing geometric constructions as programs: a brief exploration", *ijcml*, pp. 101–115, 2002.

[23] SIMMONS, M., COPE, P., "Working with a round turtle: the development of angle/rotation concepts under restricted feedback conditions", *Comput. Educ.*, v. 28, n. 1, pp. 23–33, 1997.

[24] SUN Microsystems, "Javacc™ – the Java parser generator". Disponível em: <https://javacc.dev.java.net>. Acesso em Fevereiro 6, 2009.

[25] Vários Autores, "Grammar contributions from the community". Disponível em: <https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110>. Acesso em Fevereiro 6, 2009.

[26] Wikipedia, "Épura". Disponível em: <http://pt.wikipedia.org/wiki/Épura>. Acesso em Fevereiro 6, 2009.

[27] Wöß, A., LöBERBAUER, M., MöSSENBöCK, H., "LL(1) conflict resolution in a recursive descent compiler generator", *LNCS*, v. 2789, pp. 192–201, 2003.