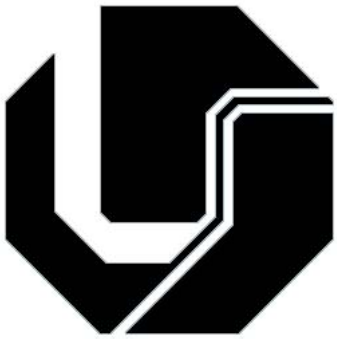
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Engenharia Elétrica



Descrição de documentos na internete e em eBooks

Mauro Jacob Honorato

Uberlândia – MG

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG - Brasil

H774d Honorato, Mauro Jacob, 1981-
Descrição de documentos na internete e em eBooks [manuscrito] / Mauro Jacob Honorato. - 2010.
96 f. : il.

Orientador: Luciano Vieira Lima.

Tese (doutorado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.
Inclui bibliografia.

1. Linguagem de programação (Computadores) - Teses. 2. Processamento de texto - Teses. 3. Livros eletrônicos - Teses. I. Lima, Luciano Vieira, 1960- II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 681.3.06:800.92

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Engenharia Elétrica

Descrição de documentos na internete e em eBooks

Dissertação apresentada à Faculdade de Engenharia Elétrica, Universidade Federal de Uberlândia por Mauro Jacob Honorato como requisito parcial para obtenção do título de Mestre em Ciências.

Banca Examinadora:

Luciano Vieira Lima (Orientador)

Antônio Eduardo Costa Pereira

Paulo Sérgio Caparelli

Reny Cury Filho

Uberlândia — 2010

"As misericórdias do SENHOR são a causa de não sermos consumidos, porque as suas misericórdias não têm fim;" Lamentações 3:22.

Agradecimentos

A Deus por seu infinito amor para com todos os homens.

Aos meu pais, Maurício e Maria, por seu amor e dedicação.

À minha esposa, Leiliane, por estar sempre ao meu lado.

Ao meu irmão, Márlon, pela grande amizade.

Aos professores e aos colegas do Laboratório de Inteligência Artificial da Faculdade de Engenharia Elétrica pelas produtivas discussões e pela oportunidade de trabalho conjunto.

Conteúdo

Lista de Figuras

Abstract

The Internet and ebooks are replacing traditional publishing houses. Encyclopedias, for instance, are published mainly in the Internet, since this media allows fast update, cooperation and corrections. Electronic readers decreased the cost of publishing books, and spare natural resources (wood for papers, for instance) needed for printting documents

Since long, publishing technology have advanced two domain specific languages for describing documents: LaTeX and XML. In recent years, XML became the choice tool for the Internet, text processing, and electronic edition of books. Therefore, this dissertation will offer tools for producing XML for www pages, and also for electronic publishing.

The Scheme programming language is a dialect of LISP well suited for text processing, since it has a good representation for structured text: Nested lists. Thus the author of this work will describe how to use Scheme for creating dynamic pages based on their XML description.

If global communication is to become feasible, we must adapt the technological advances that people who work with Artificial Intelligence made in the fields of knowledge representation, context capture, domain specific languages, reflexivity, and natural language processing. In general, AI technologies are expressed in languages of the LISP family, since they are homoiconic, i.e., their primary representation of programs is also a data structure in a primitive type of LISP. This makes metaprogramming easier than in a language without this property. This paper will show how to use metaprogramming and closures for skimming through a restricted domain text in order to grasp its syntactic structure and meaning. Once the software system obtains the general features of a text, it can process it to generate books that can be installed in dedicated reading devices; it can also generate dictionaries and digests to easy the process of reading.

Finally this work will address the problem of publishing ebooks, after specifying their structure in XML.

Resumo

A Internete e os livros eletrônicos estão substituindo as editoras tradicionais. Enciclopédias, por exemplo, são publicadas principalmente na internete, já que esse tipo de mídia permite melhoramentos, cooperação e correções rápidas. Os leitores eletrônicos diminuíram o custo da publicação de livros e o consumo de recursos naturais (por exemplo: a madeira para papel) necessários para a publicação de documentos.

Já faz muito tempo que a tecnologia de publicação alavancou duas linguagens de domínio específicas para a descrição de documentos: LaTeX e XML. Atualmente, a XML tornou-se a ferramenta preferida para a internete, o processamento de textos, e a edição eletrônica de livros. Assim sendo, essa dissertação vai oferecer ferramentas para a produção de XML para páginas www, e também para a publicação eletrônica.

A linguagem de programação Scheme é um dialeto da LISP bastante apropriada para o processamento de textos, já que ela tem uma boa representação para textos estruturados: as listas aninhadas. Assim o autor desse trabalho vai mostrar como usar a Scheme para a criação de páginas dinâmicas baseadas na descrição em XML.

Para que a comunicação global torne-se possível, precisamos adaptar-nos aos avanços tecnológicos que os cientistas da Inteligência Artificial fizeram nos campos da representação do conhecimento, captura de contextos, linguagens de domínio específico, reflexividade e processamento de linguagem natural. Em geral, as tecnologias ligadas à inteligência artificial são expressas em linguagens da família LISP. A principal razão disso é que essas linguagens possuem a propriedade do homoiconismo, ou seja, a representação primária dos programas é também uma estrutura de dados em um tipo de dado primitivo da LISP. Essa propriedade torna a meta-programação em LISP mais fácil do que seria em outras linguagens.

Este trabalho vai mostrar como utilizar a meta-programação e fechos funcionais para efetuar varreduras de textos pertencentes a domínios específicos, a fim de obter informações sintáticas e semânticas. Uma vez que um sistema

de *software* obtenha as características gerais de um texto, ele pode processá-lo para gerar livros que possam ser instalados em dispositivos dedicados à leitura. É também possível gerar dicionários e digestos que facilitam o processo de ler e interpretar textos.

Capítulo 1

Introdução

As tecnologias empregadas no desenvolvimento de soluções computacionais para a rede mundial de computadores (*internete*) estão em constante transformações. Desde que surgiu como linguagem para a criação de páginas *web* o HTML já foi alterado várias vezes. Outras tecnologias também foram aprimoradas ou criadas, como é o caso das Folhas de Estilo (*CSS*) e o JavaScript, para atender ao crescente interesse na Internet como veículo de propagação de informações e interface para negócios. A internete é usada hoje em aplicações que vão do entretenimento ao comércio eletrônico.

As páginas web comunicam todo tipo de informação, uma das possíveis utilizações é o livro, pode-se escrever um livro e disponibilizá-lo na rede, possibilitando que outras pessoas usando navegadores leiam este livro. As tags HTML são usadas para definir a estrutura do texto nas páginas *web*, esse tipo de arquivo pode ser salvo no computador do usuário permitindo uma leitura posterior mesmo não estando conectado à internete.

Neste contexto surgiu o livro eletrônico (*e-Book*) um arquivo digital com o mesmo intuito de um livro de papel, estabelecer um elo de comunicação entre o autor e o leitor, o conteúdo do livro eletrônico é exibido seguindo a mesma estrutura de um livro de papel. Existem parágrafos, capítulos e outras formas de ordenação das palavras. O livro eletrônico geralmente era impresso para facilitar a leitura, já que a leitura em monitores de computador geralmente é uma tarefa de muito esforço para a visão devido à emissão de luz do monitor.

Posteriormente ao estabelecimento da tecnologia de livros eletrônicos surgiu também um novo dispositivo, um computador de mão capaz de exibir o conteúdo de um livro eletrônico da maneira mais próxima possível que um livro de papel permite. A tela do leitor de livros eletrônicos (*e-Reader*) não emite luz como acontece em monitores de computador tradicionais, o que

permite uma leitura agradável do conteúdo do livro eletrônico. Isso devido a criação da tinta eletrônica (*e-Ink*) tecnologia empregada na tela do leitor de livros eletrônicos.

Atualmente a venda de livros eletrônicos supera a venda de livros de papel em livrarias *online* como a americana Amazon. É de extrema importância o domínio das ferramentas atuais de descrição de documentos para a utilização em arquivos feitos para servirem tanto páginas na rede mundial de computadores como também livros em dispositivos eletrônicos *e-Reader*s.

Dada a importância do assunto, *web* e *e-Books*, vamos abordar neste trabalho as maneiras de descrever os documentos utilizados nessas duas mídias eletrônicas.

1.1 Objetivo do Trabalho

1.1.1 Objetivo Geral

O objetivo dessa dissertação é trazer para a luz alguns problemas enfrentados por programadores que utilizam ferramentas impróprias de desenvolvimento de software, indicar um caminho a ser seguido para alcançar uma melhora na qualidade do software e uma diminuição no tempo de programação, apresentar ferramentas de desenvolvimento de documentos para a internet e para livros eletrônicos.

1.1.2 Objetivos Específicos

Para se atingir o objetivo geral traçado, delineiam-se os objetivos específicos da pesquisa, a saber:

- Expor alguns problemas nos métodos atuais de desenvolvimento de software.
- Identificar e compreender as ferramentas de programação mais úteis.
- Discutir a descrição de documentos eletrônicos.
- Conhecer as funções para visualização de documentos da *internet*
- Apresentar o formato de documentos utilizado nos livros eletrônicos.

Assim, esta pesquisa busca trazer uma contribuição ao processo de criação de documentos para a internete e livros eletrônicos que utilizam como base

a linguagem de marcação HTML, utilizando-se as melhores técnicas de programação e deixando pra trás aquelas que não adicionam valor ao processo de desenvolvimento de software para as aplicações sugeridas.

Capítulo 2

Metodologias de Desenvolvimento de Software

De todas as áreas nas quais uma equipe de programação pode atuar, a mais difícil é, sem dúvida, o processamento de documentação, assim como a comunicação homem máquina envolvida nessa atividade. Há várias razões para isso, das quais citamos algumas.

- Toda sociedade tem interesse em obter informação. Assim, até pessoas com pouco ou nenhum conhecimento de informática procuram documentos nas mais diversas mídias. Esses usuários podem cometer mais erros. O programa não deve simplesmente recusar entradas com erro; em vez disso, é preciso prever e realizar a intenção do usuário.

- Com a internete, o número de interessados no processamento da informação aumentou a ponto de incluir a metade da população do planeta. Além disso, a quantidade de documentos disponíveis é avassaladora. A tendência é que toda mídia escrita e visual migre para a internete. Filtrar, reformatar, protocolar, traduzir, interpretar e colocar legendas e distribuir esse enorme volume de informação é tarefa de grande dificuldade.

- Dar acesso ao conteúdo semântico da gigantesca massa de dados presente na internete exige complexos programas de processamento de linguagem natural, uma das áreas mais difíceis a inteligência artificial.

Da discussão acima, conclui-se que o moderno profissional da informática precisa de treinamento e ferramentas eficientes e que aumentem a produtividade em ordens de grandeza com relação aos métodos tradicionais.

Há muitas abordagens ao desenvolvimento de programas. Nesse capítulo, vamos elencar as metodologias tratadas nesse trabalho, apresentar a RATIONALE por trás delas e listar os prós e contras das tecnologias envolvidas, assim como resumir as críticas que os especialistas fazem a cada uma delas. Inicialmente, não reuniremos muitas evidências para as afirmações que faremos. Em capítulos posteriores, discutiremos cada metodologia em separado e apresentaremos as evidências favoráveis e desfavoráveis.

Para facilitar as citações de artigos, unificar a terminologia, evitar ambiguidades e esclarecer as siglas, utilizaremos de expressões inglesas com grande liberalidade. Nesse caso, as expressões aparecerão em itálico — conforme exigido pela norma culta do português — e, na primeira vez que forem utilizadas, sugeriremos uma tradução, uma ortografia em português, uma sigla ou um neologismo. Por exemplo, *software* (software), *flowcharts* (fluxograma), etc. Se a expressão tem uma sigla, passaremos a usar a sigla, sem colocá-la em itálico; e.g. *Domain Specific Languages* (DSL), *Domain Specific Modelling* (DSM), *Unified Modeling Language* (UML), etc.

As afirmações realizadas nesse trabalho serão justificadas ou (1) por citação bibliográfica de experimentos controlados, (2) estatísticas ou (3) casos particulares. As estatísticas terão por base dados colhidos na internete. Quanto aos experimentos controlados, forneceremos referências a trabalhos de outros pesquisadores, sempre que nessessário. Ex. Using UML takes 15% longer than just coding (vide [Kelly]).

Segundo a lógica da pesquisa científica de Carl Popper, casos particulares só têm valor para desmontar um modelo. Por exemplo, alguns casos bem sucedidos de profissionais que não projetam modelos dos sistemas de processamento da informação que vão programar provam que tais modelos não são necessários ao sucesso do empreendimento. Entretanto, não podemos afirmar que as ferramentas utilizadas por tais profissionais são, em geral, as mais produtivas.

O autor do trabalho tentará não deixar nenhuma afirmação sem justificativa de um tipo ou do outro.

Definição — Sucesso. Uma metodologia alcançou sucesso se satisfaz uma ou mais de uma das condições abaixo.

1. É amplamente adotada por algum segmento da indústria. Essa condição não exige que a adoção do paradigma tenha resultados palpáveis. Por exemplo, modelagem gráfica apresenta poucos resultados práticos. Entretanto, é uma metodologia de sucesso, pois há amplos segmentos da indústria que a usam no desenvolvimento de sistemas([Gat]).

2. Grande número de usuários finais adquirem produtos e serviços que dificilmente seriam desenvolvidos sem a metodologia. Um exemplo é a metodologia das linguagens de domínio específico (DSL). Essa metodologia foi difundida por Paul Graham [onlisp] e levou à criação do comércio pela internete ([Avg]), de ferramentas de busca como o Google([Spolsky]), etc. Os dois exemplos dados mostram que programas desenvolvidos com DSL estão entre os favoritos dos usuários finais.

3. A metodologia é ensinada nos meios acadêmicos, embora seja pouco utilizada na indústria e não tenha programas largamente utilizados para mostrar. Esse é o caso da lógica de Floyd-Hoare [Hoare].

Definição — Classes de aplicações. Uma classe de metodologia de desenvolvimento de *software* tem elementos ou atributos unificadores. Esses atributos permitem saber se a metodologia pertence à classe. Por exemplo, a classe da modelagem gráfica possui dois atributos:

- Uma especificação gráfica e informal do sistema.
- A especificação é feita antes de se iniciar o desenvolvimento.

Esses dois atributos colocam na mesma classe modelagem por *Unified Modeling Language* (UML), *Data Flow Diagrams* (DFD) e fluxogramas.

2.1 Ferramentas gráficas

A utilização de ferramentas gráficas para especificar ou modelar um sistema computacional é muito antiga. Os primeiros desses sistemas foram os fluxogramas (*flowcharts*), que foram propostos por Frank Gilbreth aos engenheiros reunidos em uma palestra realizada em 1921 na *American Society of Mechanical Engineers* (ASME) [Process Charts-First Steps in Finding the One Best Way]. Técnicas modernas, como os diagramas de atividade da UML são extensões dos fluxogramas.

Rationale. As ferramentas gráficas foram inspiradas nos projetos de engenharia civil. Na construção de um prédio, por exemplo, um arquiteto faz um desenho ou uma maquete com as especificações gerais da edificação. Um engenheiro civil faz uma planta baixa, mostrando as dimensões de cada parede, a colocação de tubulações e fiação elétrica, etc. O engenheiro civil também faz cálculos estruturais, de modo a indicar o material e a forma de construção de cada elemento, garantindo que o todo não vai desmoronar com o tempo

ou com eventos, tais como terremotos. Apenas depois que as especificações do prédio estão prontas, os pedreiros e mestres de obras e escultores entram em ação.

Todos sabemos que as principais ferramentas de especificação em engenharia civil e mecânica são gráficas. Então porque não seriam na engenharia de *software*? Aliás, como já foi dito nesse documento, a primeira ferramenta de especificação usada na programação foi herdada da engenharia mecânica.

Analogia não é um argumento forte. O fato de um engenheiro civil usar uma ferramenta gráfica de especificação não faz com que isso seja uma boa ideia para um cirurgião plástico ou um cozinheiro. Acontece que os defensores da especificação gráfica têm mais um argumento. É possível construir uma edificação sem especificação gráfica e cálculos estruturais. Os antigos egípcios erigiram as pirâmides sem nada disso, e elas estão de pé até hoje. Mas, argumentam, a especificação gráfica de um programa permite trazer, para o projeto, a colaboração de pessoas que não seriam capazes de programar. Um analista de sistemas, mesmo que não tenha talento nato para escrever código em uma linguagem de programação, pode levantar os requisitos e desenhar diagramas UML que ajudem os programadores a desenvolver o sistema (1) mais rapidamente e (2) com menos erros.

Criticismo. O criticismo ao UML e a outras ferramentas gráficas é simples: Se um modelo escrito em UML realmente ajuda a aumentar a qualidade do produto, então basta fazer um experimento que compare dois grupos, um utilizando um modelo UML e outro trabalhando diretamente no código final. Esse tipo de experimento já foi realizado várias vezes, mostrando que, pelo menos no caso da UML, a especificação gráfica não traz benefícios (vide [Kelly]).

Uma hipótese para o fracasso da modelagem gráfica reside justamente no axioma de que ela traz a colaboração de leigos em programação. Se a modelagem gráfica garantisse, realmente, a colaboração de um especialista no domínio mas leigo em programação, então ela seria desejável. Examinemos alguns exemplos.

- Um matemático, especialista em cálculo numérico ou em estatística, poderia escrever um modelo UML de um algoritmo de otimização pelo método Simplex, tão importante no planejamento e na administração de empresas. O modelo criado pelo matemático seria fornecido aos programadores que não precisariam entender o método Simplex e, assim, desenvolveriam o sistema mais rapidamente.

- Um biólogo, que estivesse resolvendo o difícil problema de determinar a estrutura molecular de um composto orgânico a partir de dados de um

espectrômetro poderia fazer um modelo em UML, que seria programado rapidamente em Blub (Blub é o nome de uma linguagem genérica, sem macros, proposta por Paul Graham).

O problema desse argumento é que a colaboração sempre vem de pessoas que (1) não sabem programar e (2) não sabem matemática nem bioquímica. Em geral, essas pessoas tem habilidades técnicas extremamente limitadas. Não conseguem nem mesmo aprender a programar. Como vão aprender o método Simplex ou bioquímica? De fato, esses *engenheiros de software* fazem o bioquímico (ou o matemático) perder seu tempo explicando-lhes o problema. Fazem também os programadores perderem tempo.

Na prática, desenvolvimentos de *software* complexo são, com frequência, resolvidos de duas maneiras.

- O bioquímico ou o matemático aprendem a programar. Especificam suas ideias em um programa que funciona. Os especialistas em ciência de computação apenas orientam sobre bibliotecas, algoritmos mais eficientes, melhoram a linguagem, ajudam a instalar o compilador na máquina, etc. Vários sistemas ilustram esse esquema de trabalho e demonstram que o especialista pode trabalhar sem ajuda de programadores. Joshua Lederberg aprendeu a programar em LISP e escreveu o Dendral diretamente nessa linguagem; por esses e outros trabalhos foi agraciado com o prêmio Nobel de Medicina em 1958. Existem centenas de exemplos como esse, dos quais citamos dois: o método da máxima entropia, na economia, a transformada rápida de Fourier e o filtro de Kalman, na engenharia. No Brasil, o professor Renato Sabatini é um médico que aprendeu a programar e desenvolve aplicações em informática biomédica. Atenção: *Esses casos particulares mostram que é falsa a premissa de que o concurso de profissionais de informática é necessário para desenvolver sistemas complexos.*

- O programador aprende o domínio. Então, sem entrevistas, faz o programa que, depois, oferece aos profissionais da área. Exemplo: Herbert Simon aprendeu administração e criou várias técnicas de tomada de decisões que lhe valeram o prêmio Nobel de Economia em 1978. Aqui também temos dezenas de exemplos. Tomografia computadorizada e Lithotripsia foram desenvolvidas por programadores que adquiriram conhecimentos de medicina. Esses fatos mostram que um programador não precisa do auxílio de especialistas para desenvolver seu trabalho.

Paul Graham apresenta o seguinte argumento contra metodologias baseadas na filosofia do planejamento seguido de implementação ([Lnotes]):

> *O método planeje-e-implemente pode ter sido um bom modo de construir barragens e iniciar invasões, mas a experiência não tem mostrado ele como uma boa maneira de escrever programas. Por quê? Talvez seja porque computadores são muito exatos. Talvez exista mais variações entre programas do que entre barragens ou invasões. Ou ainda os métodos antigos não funcionam porque os velhos conceitos de redundância não tem análogos no desenvolvimento de* software*: Se uma barragem contêm 30% a mais de concreto, isso é uma margem de erro, mas se um programa executa 30% a mais de trabalho, isso é um erro.*

Esse último ponto é interessante. Até áreas em que a modelagem gráfica era tradicional abandonaram essa técnica quando os sistemas cresceram muito. Tal é o caso da eletrônica digital, onde os circuitos lógicos eram especificados com diagramas. Hoje, circuitos lógicos são especificados em linguagens de programação textual como a Verilog. Na listagem 2.1 temos um exemplo de contador especificado na linguagem Verilog. Programas como esse são, modernamente, utilizados no lugar de diagramas.

Listing 2.1: code/contador.vl — Um contador em Verilog

```
1  module contador(out, clk, reset);
2    parameter WIDTH = 8;

4    output [WIDTH-1 : 0] out;
5    input  ________      clk, reset;

7    reg [WIDTH-1 : 0]    out;
8    wire _       clk, reset;
9    always @(posedge clk)
10     out <= out + 1;

12   always @reset
13     if (reset)
14       assign out = 0;
15     else
16       deassign out;

18 endmodule // contador
```

2.2 Programação exploratória

A programação exploratória surgiu entre pesquisadores de inteligência artificial programando em LISP, uma linguagem que tem as seguintes características:

Tipos de dados dinâmicos: não é necessário declarar o tipo de dado dos argumentos de um procedimento ou função. No momento da execução, o programa pode verificar o tipo do dado que está recebendo e processá-lo de acordo. Se o usuário final precisa de fornecer um número e fornece um símbolo, o programa pode reconhecer e processar o erro e pedir ao usuário que o corrija, fazendo sugestões.

Compilador é interativo: Lê uma expressão digitada pelo programador, avalia a expressão (executa) e imprime o valor. Ao interagir com o Lisp todas as construções do usuário são compiladas ao mesmo tempo em que são criadas.

Inspiração na matemática: A linguagem segue o modelo computacional conhecido como Cálculo Lambda criado por Alonzo Church. Isso permite um estilo construtivo de programar, ou seja, pode-se construir novos paradigmas de programação facilmente.

Sistema de macros: o que permite criar novas sintaxes e novas linguagens em cima da LISP. Esse sistema de macros requer uma linguagem cuja sintaxe represente uma estrutura de dados de fácil manipulação, as listas são

essa estrutura de dados. Em LISP, uma entidade sintática é chamada de Expressão Simbólica, ou seja, SEXPR.

Diagramas como UML levam a um estilo *top-down* de programação. Na construção de um prédio, o arquiteto concebe o edifício pronto (*top*), ou seja, pensa em alto nível. Só depois do prédio estar projetado é que os pedreiros começam a pensar nos detalhes, isto é, na colocação dos tijolos e no assentamento das portas. A programação exploratória é *bottom up*, ou seja, começa com os algoritmos mais básicos. Não há projeto. O explorador escreve programas simples, testa-os, e tenta combiná-los em estruturas mais complexas. Se o resultado for bom, parte para estruturas ainda mais complexas. Como disse Aristóteles, a exploração parte do conhecido para o desconhecido e é realizada pela ação.

Nas mãos de profissionais competentes, a programação exploratória aumenta a produtividade e leva a programas bastante sofisticados. Infelizmente, ela alcançou enorme popularidade entre amadores com conhecimentos mínimos de engenharia de software. Hoje esse tipo de metodologia é o mais popular no mundo. Claro que, para alcançar essa popularidade, os profissionais abandonaram LISP, de aprendizado difícil. Projetaram versões simplificadas da LISP, com sintaxe menos flexível, porém mais intuitiva. Duas dessas linguagens alcançaram imensa popularidade, Python e PHP. A linguagem Lua, projetada pelo brasileiro Roberto Ierusalimschy também é bastante popular entre os adeptos da programação exploratória.

Hoje em dia, no Brasil, a programação exploratória é utilizada por quase todas as empresas, no comércio e até na indústria. Basta entrar em uma página de comércio eletrônico para encontrar exemplos da utilização desse tipo de técnica (ver [opencart]).

2.3 Linguagens de domínio específico

As linguagens de domínio específico formam a base de outra metodologia que começou com Lisp. Essa metodologia foi simplificada e tornou-se muito popular entre amadores. Paul Graham em [onlisp] explica como as linguagens de domínio específico são utilizadas em Lisp.

> *Porque a Lisp te deixa livre para definir seus próprios operadores, você pode moldá-la para a linguagem que você precisar. Se você estiver escrevendo um editor de textos, você pode transformar Lisp em uma linguagem para escrever editores de texto.*

> *Se você estiver escrevendo um programa CAD, você pode transformar Lisp em uma linguagem para escrever programas CAD. E se você não estiver certo ainda de qual tipo de programa você está escrevendo, é mais seguro apostar em Lisp para escrevê-lo. Qualquer que seja o seu tipo de programa, Lisp irá, durante a programação, evoluir para uma linguagem própria para escrever aquele tipo de programa.*

Para o grande público, uma linguagem de domínio específico facilita a implementação de aplicações em uma área particular. A primeira dessas linguagens a tornar-se amplamente usada foi o Autolisp, em sistemas de CAD para engenharia civil e mecânica. Várias linguagens de editoração de textos seguiram os passos da Autolisp. Finalmente, as linguagens de marcação (*markup languages*) conquistaram o público leigo. Não podemos nos esquecer do Matlab, entre engenheiros.

Rationale. Qualquer que seja o sistema que estamos desenvolvendo, a abstração deve sempre começar no lado do problema. Aliás, esta é uma das principais críticas que os especialistas fazem a linguagens de modelagem gerais, como UML: Como elas devem ser gerais (precisam modelar qualquer área do conhecimento), são forçadas a fazer abstrações na ponta errada, qual seja, no lado do computador e da programação.

Em resumo, os projetistas da UML não sabiam, de antemão, em que área do conhecimento ela seria utilizada, se no comércio, na engenharia, na física, na comunicação, ou na medicina. Só sabiam que ela serviria para criar programas. Então, a linguagem especifica e descreve programas; suas abstrações são diagramas de classes, diagramas de objetos, diagramas de componentes, diagramas de instalação, diagramas de casos de uso de programas, diagramas de transição de estados, diagramas de atividade, etc.

Acontece que precisamos de uma linguagem que faça abstração no lado do problema. Se estamos escrevendo um editor de textos, é preciso falar em salvar arquivos, inserir texto, desfazer alterações, procurar palavras no dicionário, corrigir ortografia, etc. Se nosso problema for biologia, precisamos falar em taxonomia, genoma, alelos, características recessivas, equilíbrio populacional, etc. Em linguística, a linguagem específica fala de regras de produção e gramáticas.

As linguagens de domínio específico foram projetadas para uma dada área. Assim, elas especificam bem problemas daquela área. Nas mãos de um programador Lisp, capaz de criar abstrações e desenvolver linguagens conforme seja necessário, DSM (*Domain Specific Modelling*) é uma das mais eficientes metodologias de programação e desenvolvimento de sistemas. Paul

Graham mostra a dinâmica das linguagens de domínio específico:

> *It's a long-standing principle of programming style that the functional elements of a program should not be too large. In accordance with this principle, a large program must be divided into pieces. The traditional approach (to do this) is called top-down design: the purpose of the program is to do these seven things, so I divide it into seven major subroutines. The first subroutine has to do these four things, so it in turn will have four of its own subroutines, and so on. This process continues until the whole program has the right level of granularity, each part large enough to do something substantial, but small enough to be understood as a single unit.*
>
> *In Lisp, you follow a principle which could be called bottom-up design, changing the language to suit the problem. You don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had suchand- such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.*

Lisp possui macros, que tornam eficiente o projeto de novas linguagens e abstrações. Em resumo, as linguagens de domínio específico que um programador Lisp desenvolve são tão eficientes quanto o próprio compilador Lisp. Além disso, Lisp é interativa, ou seja, você digita uma expressão e obtém a resposta imediatamente, sem passar pelo ciclo de compilação e execução. Assim, o programador pode criar a abstração, testá-la e incorporá-la à linguagem ou descartá-la, conforme o resultado. Praticamente todas as linguagens de domínio específico oferecem essa particularidade da interação. Entretanto, nas linguagens construídas sem Lisp, a interação é oferecida por um intérprete, lento e ineficiente. Em Lisp, o programador tem um compilador incremental. Expressões Lisp são transformadas em linguagem de máquina, executadas e o resultado é mostrado tão logo o programador aperte a tecla

Enter. Por exemplo, depois de definir a função *soma* em Lisp, podemos ver o código assembly que é gerado pelo compilador usando a função *disassemble*.

```
* (defun soma (x y) (+ x y))

SOMA
* (disassemble 'soma)

; disassembly for SOMA
; 23C8374D:     8B55FC     MOV EDX, [EBP-4] ; no-arg-parsing entry point
;       50:     8B7DF8     MOV EDI, [EBP-8]
;       53:     E878CA37FE CALL #x220001D0  ; GENERIC-+
;       58:     7302       JNB L0
;       5A:     8BE3       MOV ESP, EBX
;       5C: L0: 8BE5       MOV ESP, EBP
;       5E:     F8         CLC
;       5F:     5D         POP EBP
;       60:     C3         RET
;       61:     CC0A       BREAK 10         ; error trap
;       63:     02         BYTE #X02
;       64:     18         BYTE #X18        ; INVALID-ARG-COUNT-ERROR
;       65:     4F         BYTE #X4F        ; ECX
NIL
*
```

Crítica ao Domain Specific Modelling. As críticas ao DSM não são dirigidas à tecnologia tal como utilizada no mundo da Lisp. As dificuldades surgem quando essa tecnologia, que exige ferramentas e sintaxe apropriadas, é utilizada sem Lisp. A primeira dessas dificuldades surge na eficiência. Python, uma linguagem de Script, é 100 vezes mais lenta do que Lisp. Matlab é ainda mais lenta. Mesmo com perda de velocidade e eficiência, ainda seria possível usar DSM dentro dos domínios para os quais as linguagens e abstrações foram planejadas. Assim, seria possível usar Matlab para rodar programas em C da *toolbox*. Infelizmente, tal coisa não acontece. O usuário final acaba utilizando a linguagem de domínio específico em outras áreas, onde abstrações mal colocadas vão somar-se à ineficiência grosseira. Temos, então, programas de algoritmos genéticos representados como vetores na Matlab e que levam dias para apresentar resultados.

Outra dificuldade com as Domain Specific Languages (DSL) é que os programadores que as usam têm dificuldades em usar uma linguagem eficiente

e mudar as abstrações quando saem da área para a qual a DSL foi projetada.

Todas as críticas às DSLs não valem quando a metodologia é usada em Lisp, com modelos sob medida para a área de atuação e para o problema em questão. Infelizmente, tecnologia desenvolvida para Lisp não funciona sem Lisp. Da mesma forma, transporte naval precisa de navios.

Capítulo 3

Engenharia de Software

Conforme vimos ao tratar das ferramentas gráficas de modelagem e especificação, como é o caso da UML (*Unified Modeling Language*) e da IDEF (*Integrated Definition Methods*), várias críticas são feitas a essas tecnologias rígidas e que adicionam pouco valor ao ciclo de criação e desenvolvimento. Alguns críticos apontam também para o fato de que praticamente nenhuma aplicação complexa foi implementada usando essas tecnologias; por aplicação complexa entende-se sistemas operacionais, servidores de aplicação, como a internete, jogos de vídeo, grandes aplicações numéricas, etc. O peso da artilharia é dirigido contra a UML, mas a única razão para isso é a relativa popularidade dessa ferramenta entre professores de Engenharia de Software.

DSM (*Domain Specific Modeling*) neutralizou alguns pontos fracos da UML e da IDEF, mas ainda deixou várias pontas soltas. Para pronta referência, vamos resumir abaixo as críticas feitas à UML, sem nos esquecer de que a maioria dessas críticas aplica-se a outros conjuntos de diagramas.

Algumas críticas feitas a UML e outras ferramentas de desenvolvimento de Software podem ser descartadas sem um exame aprofundado; por exemplo, embora seja estatisticamente verdadeiro que programadores que usam UML cometem mais erros e desenvolvem aplicações de qualidade inferior, isso não se deve tanto a características intrínsecas da UML, quanto ao fato de que gerentes mal treinados em programação tendem a impor a utilização de UML a suas equipes. Mesmo desconsiderando as críticas apressadas, alguns pontos levantados pelos críticos da UML merecem um exame aprofundado. É o que faremos nas próximas seções.

3.1 Abstração na ponta errada

O termo *abstração* vem da palavra latina que significa *isolar e captar um conceito não associado a nenhum fato ou ocorrência particular.* O exemplo clássico de abstração é o número. De acordo com [Schmandt-Besserat], na Antiga Mesopotâmia, as pessoas utilizavam figuras simbólicas (*tokens*, em inglês) para efetuar contagens. Os *tokens*, feitos de argila, eram modelados em formas tais como cones, esferas, cilindros, discos e tetraedros. A contagem era realizada pelo emparelhamento de um *token* com cada objeto do conjunto que se queria quantificar. Havia um *token* diferente para cada tipo de coisas quantificáveis, ou mais precisamente, para cada tipo de coisas enumeráveis.

Alguns exemplos concretos tornarão mais claro o que ocorria na Mesopotâmia. O cone e o disco representavam várias medidas de grão. Já o tetraedro indicava um dia de trabalho braçal. A abstração aconteceu quando os babilônios (a Babilônia era um dos países que floresceram na Mesopotâmia) descobriram que podiam utilizar o mesmo símbolo para contar diferentes classes de coisas. Talvez tenha faltado os *tokens* específicos para contar jarros de óleo, e um varejista inventivo resolveu aproveitar os *tokens* reservados para sacos de trigo na tarefa; esse varejista descobriu, então, que não fazia a menor diferença que objeto usava na contagem. O processo de contagem por emparelhamento funcionava qualquer que fosse o *token* utilizado para levá-lo a cabo.

Aproveitando a experiência dos babilônios, os antigos egípcios substituíram os diversos *tokens* da Mesopotâmia por um traço vertical. Assim, se um egípcio queria contar quatro dias, ele escreveria ||||. Podemos afirmar que os egípcios tinham abstraído, de suas coleções de *tokens*, a ideia de número; para isso, eles descartaram os detalhes irrelevantes, tais como a forma e o tamanho de cada *token*. Além disso, os egípcios criaram um símbolo para seus números abstratos, a saber, o traço vertical.

Cedo em sua história, os egípcios perceberam que era conveniente contar os *tokens* de contagem. Em vez de escrever 32 traços verticais, era possível usar um símbolo para representar grupos de dez traços. Com esse avanço, os 32 traços passaram a ser escritos assim: $\cap\cap\cap$ ||.

Os indianos foram um passo além dos egípcios na estrada da abstração: Eles contavam os agrupamentos de *tokens* de contagem com os mesmos símbolos que usavam para contar outras coisas. Por exemplo, no numeral 32, o dígito 3 representa grupos com dez *tokens* de contagem cada um.

Na programação de computadores, uma das mais importantes abstrações é a abstração de dados. Sabemos que computadores podem manipular apenas sequências de *bits* organizadas em registradores e células de memória. Entretanto, pode-se construir padrões reconhecíveis e classificáveis com esses *bits*,

de modo que representem letras e dígitos. Com isso, o programador ganha a liberdade de esquecer-se da maneira de implementar células de memória e guardar para si apenas o fato de que células de memória representam caracteres.

As células de memória com seus *bits* podem também ser usadas para controlar a ordem na qual as instruções e chamadas de funções são executadas ou avaliadas. Desse fato, uma linguagem de computação abstrai as ideias de fluxo de controle e comandos de controle. Abstrações podem, também, ser aplicadas recursivamente às próprias linguagens de computação. Linguagens como LISP permitem a criação de novas linguagens, a fim de melhor expressar aspectos específicos de um dado sistema. Por exemplo, em Scheme (um dialeto da LISP), pode-se definir um novo comando de controle que suspende a execução, a não ser quando uma condição líquida e certa é satisfeita.

```
> (define-syntax unless
    (syntax-rules ()
      ((unless test consequent)
        (if (not test) consequent))))

> (define x 32)
x
> (unless (< x 32) (print x))
32

> (unless (< x 42) (print x))
#f
```

Uma linguagem de modelagem é um conjunto de ferramentas para criar abstrações linguísticas tendo em vista facilitar a preparação de projetos de sistemas. Evidentemente, essas abstrações devem abordar os conceitos envolvidos no domínio do problema tratado. [Lispy], entretanto, alega que UML é voltada para o mapeamento de estruturas arquitetônicas de codificação, e não de conceitos pertencentes ao domínio do problema. Vamos apontar, contudo, que a meta de UML é projetar programas; por isso, sua maneira de criar abstrações talvez não esteja completamente errada. É claro que, aceitando esse ponto de vista, UML é muito menos útil do que afirmam seus proponentes; afinal, ela só serviria para um cliente cuja empresa desenvolvesse *software* e tal cliente se sentiria pouco inclinado a contratar um concorrente para ensinar-lhe a fazer seu serviço.

No nosso enfoque, é irrelevante se [Lispy] está certo ou não no que tange à

UML. Nesse trabalho, o foco está direcionado para o fato de que uma linguagem de modelagem deve abordar o domínio do problema tratado, qualquer que seja ele (até mesmo planejamento de estrutura arquitetural de código está no menu). Assim sendo, o primeiro requisito de uma linguagem de engenharia de *software* é facilitar a criação de abstrações para o domínio do problema.

3.2 Modelos executáveis

A geração de código executável útil, se não é absolutamente necessário, seria uma característica no mínimo desejável em uma linguagem de modelagem. Entre outras vantagens, um modelo executável contribui para os esforços de teste e para verificar a completude de uma especificação. Além disso, o modelo executável permite traçar o perfil do algoritmo, descobrindo os pontos de engarrafamento, pontos esses que mereceriam maiores cuidados e afinação.

Podemos até mesmo visualizar o processo de desenvolvimento de *software* como sendo a criação de uma sequência de modelos que se tornam cada vez mais refinados, práticos e diversificados. Por esse ponto de vista, os modelos que não podem ser executados estariam reduzidos a mera documentação. Está claro que documentação é um aspecto que não pode ser deixado de fora de qualquer empreendimento; os modelos executáveis, porém, dão à documentação um maior impacto e poder de comunicação. Como todos sabemos, aeromodelos voam.

O resumo dessa seção é que o terceiro requisito de uma ferramenta de engenharia de *software* é gerar protótipos e modelos executáveis. A propósito, [Lispy] também critica UML nesse respeito. Segundo [acl1] ao invés de torcer para que o modelo feito por outras pessoas não tenha erro (como acontece usando diagramas UML) , é melhor fazer o custo do erro diminuir. Sendo o custo de um erro igual ao tempo que é necessário para corrigí-lo, o mais apropriado é usar uma ferramenta que cria o próprio programa ao invés de um modelo desejado.

3.3 Metodologia científica

[Karl Popper], em um livro revolucionário, colocou a pesquisa científica sobre bases lógicas. Sua proposta ganhou apoio de grandes cientistas, como Pedro Medawar e Albert Einstein. Para colocar o leitor no contexto, nos parágrafos que seguem, vamos citar Popper quase que VERBATIM.

Para [Karl Popper], um cientista, seja ele teórico ou experimental, formula enunciados, que devem ser verificados cuidadosamente. Por exemplo, nas ciências empíricas, o pesquisador formula hipóteses, que a observações e testes experimentais. Popper proporcionou-nos uma análise lógica desse procedimento, ou seja, conforme já dissemos, colocou o método das ciências empíricas sobre bases lógicas. Para isso, teve de resolver o **problema da indução**. Até Popper, a concepção mais aceita era a de que as ciências empíricas caracterizavam-se pelo emprego de "métodos indutivos"; assim, a lógica da pesquisa científica era identificada com a Lógica Indutiva; estudá-la era analisar esses métodos indutivos.

Mas o que, exatamente, significa indução? Uma inferência é indutiva se ela parte de enunciados singulares, tais como descrições dos resultados de observações ou experimentos, e conduz para enunciados universais, tais como hipóteses ou teorias.

Popper nega que, de um ponto de vista lógico, possa haver justificativa na inferência de enunciados universais a partir de enunciados singulares, independentemente de quão numerosos sejam estes. Para ele, qualquer conclusão colhida desse modo sempre pode revelar-se falsa. Não importa quantos casos de cisnes brancos um cientista tenha observado: ele simplesmente não tem o direito de concluir que todos os cisnes são brancos.

O problema que Popper se propôs a resolver é saber se as inferências indutivas se justificam e em que condições elas podem ser aplicadas.

O ponto de partida de Popper é que a teoria científica será sempre provisória. Para ele, não seria possível confirmar a veracidade de uma hipótese pela simples constatação de que os resultados de uma previsão com base nela tenham se verificado em um certo número de experimentos e observações. Essa hipótese deverá gozar tão somente da glória passageira de ser uma teoria que não foi contrariada pelos fatos por um certo tempo.

Experimentos e observações nunca poderão provar que uma hipótese científica é verdadeira. No mundo real, o que as observações podem e devem tentar fazer é tentar achar provas da falsidade de hipóteses e teorias. Este processo de confronto da teoria com as observações terminará por demonstrar a falsidade da teoria. Quando isso acontecer, é preciso ou eliminar a teoria falsa, ou determinar as condições em que ela ainda possa ser utilizada. Além disso, os cientistas devem procurar outra teoria que funcione onde a primeira falhou. Essa segunda teoria também deve ser submetida ao ciclo de testes, que provarão que ela é falsa.

A beleza da proposta de Popper, e um dos aspectos que poucos comentadores apontaram, é que a lógica de Popper garante o progresso contínuo da ciência. Nunca surgirá uma teoria final, a hipótese perfeita, que não pode ser melhorada. Se a teoria que não pode ser melhorada surgisse, o progresso

científico pararia. Por definição, *progresso* é melhoramento, e por etimologia é avançar no rumo da perfeição e da verdade. Uma vez atingida essas metas, não pode haver mais progresso. Para a ciência, o progresso é mais importante do que a própria verdade.

Se aceitamos as ideias de Popper, um cientista, que deseja ver sua área do conhecimento progredir, deve, além de propor teorias, sugerir métodos de como provar que suas propostas são falsas. Essas sugestões acelerariam o ciclo de desenvolvimento científico.

O autor dessa dissertação gostaria de propor um ciclo de desenvolvimento de *software* semelhante à lógica da pesquisa científica de Popper. O engenheiro de *software* deve propor e construir um modelo executável plenamente funcional, a fim de satisfazer às especificações e condições de uso. Entretanto, ele não deve esperar que esse modelo funcione para sempre. Com as mudanças nas condições de trabalho, com o aparecimento de novos usos, com o surgimento de necessidades inesperadas, com modificações no comportamento dos usurários, o modelo deixará de ser satisfatório. Quando isso acontecer, ele deve ser modificado e adaptado.

Essa vida progressiva do modelo muda inúmeros aspectos da engenharia de *software*. Em primeiro lugar, não queremos mais um projeto *top down*, em que um analista de sistema cria uma estrutura arquitetônica, que será implementada por programadores e entregue ao uso. Agora, os programadores deverão modificar continuamente as abstrações contidas no modelo, para mantê-lo sempre à superfície. Não precisamos mais daquele analista que faz um projeto não executável e final, que deve ser desenvolvido por outros. O desenvolvimento e as abstrações devem evoluir de forma incremental. Como disse Paul Graham, as abstrações linguísticas crescem em direção à aplicação, que avança contra as abstrações linguísticas, até que se aquietem entre rios e montanhas, as fronteiras naturais do projeto. De vez em quando, pequenas rixas de fronteiras devem retificar a divisa entre a realidade e o modelo.

3.4 Representação gráfica do modelo

Embora diagramas formem uma ferramenta útil para argumentar e resolver problemas, a complexidade dos diagramas tende a crescer muito rapidamente a medida que o tamanho do problema aumenta. [Amelsvoort] acredita que diagramas argumentativos devem ser preferidos em detrimento de outras formas de modelagem apenas quando a informação é de complexidade média. Se a complexidade é alta, torna-se difícil representar o problema com diagramas. Por outro lado, se os conceitos são muito simples, simplesmente não vale a pena traçar um diagrama.

Devido às limitações de seus diagramas tradicionais para problemas complexos, engenheiros eletrônicos partiram para a utilização de linguagens de programação textual, como é o caso da Verilog e da VHDL; com essas linguagens, eles são capazes de especificar, projetar e simular circuitos integrados e sistemas eletrônicos. Se até outras indústrias que não a de *software* acreditam que há ganho em usar linguagens textuais em modelagem, projetos e simulações, acreditamos que os engenheiros de *software* não deveriam abandonar suas ferramentas tradicionais, que são, afinal de contas, linguagens textuais.

3.5 Produtividade

Uma das metas da modelagem ou de qualquer outra ferramenta de engenharia de *software* é aumentar a produtividade na captura, representação e implementação de conceitos de domínio específico. Por isso, qualquer processo de desenvolvimento de *software* deve abordar o lado da produtividade.

Experimentos mostram que muitas ferramentas populares ou impostas por gerentes de projetos a suas equipes não aumentam a produtividade em nenhum modo detetável. Por exemplo, medidas realizadas por [Dzidek et al.] não conseguiram convencer muitos especialistas de que modelagem por diagramas garantem um aumento de produtividade (vide [Kelly]). Por isso, aumento em produtividade é a próxima entrada em nossa lista de necessidades para uma metodologia de desenvolvimento de *software*.

3.6 Colaboração

Um fato interessante é que existem pessoas programando em linguagens e usando metodologias que atendem todos os quesitos discutidos anteriormente. Por exemplo, se acreditarmos em Paul Graham, a Lisp e macros tornam fácil a criação de linguagens de domínio específico que representam bem os conceitos presentes no problema de maneira melhor do que ferramentas de prateleira. A propósito, macro é uma ferramenta LISP que possibilita a manipulação das estruturas da linguagem, a fim de moldá-las de uma forma que é mais adequada para determinada aplicação. Em LISP as estruturas de dados e as construções da linguagem têm a mesma forma (*lista*), o que permite uma fácil reestruturação de controle e abstração de dados em novas formas e avenidas.

Não há dúvidas de que a produtividade é alta entre os programadores LISP, já que eles fazem o possível para que todos saibam disso. Além disso,

programadores LISP são famosos por construir abstrações na ponta correta do problema; de fato, suas abstrações mapeiam diretamente os conceitos do domínio do problema. Claro que as linguagens de modelagem de domínio específico são executáveis, quando implementadas em LISP.

Infelizmente, nem mesmo a *metaprogramação* em LISP parece abordar o próximo item da nossa lista de desejos para uma nova metodologia de engenharia de software: Ela deve adicionar valor a um Ambiente de Colaboração Integrado (ICE - Integrated Collaboration Environment) na engenharia de software. Para nós, uma das principais razões para o sucesso da C é que essa linguagem torna possível o compartilhamento de suas bibliotecas de maneira portável, até mesmo para profissionais que não conhecem C.

3.7 Metaprogramação

Antes de avançar nessa discussão, vamos examinar o conceito de metaprogramação. Um metaprograma é um programa de computador que manipula a si mesmo e a outros programas, e faz isso como se os programas fossem dados.

Uma metalinguagem é qualquer linguagem que alguém pode usar para codificar metaprogramas. LISP é sua própria metalinguagem; essa propriedade é chamada de reflexividade. LISP é uma linguagem de computação reflexiva por excelência.

Há várias razões para usar uma ferramenta como LISP na engenharia de *software*. Dentre essas razões estão:

- Metaprogramação — cria uma ponte entre a linguagem de programação de computador e uma abstração de domínio específico. Em outras palavras, LISP executa abstrações na ponta correta do problema.

- Ela aumenta a produtividade, já que automatiza parte do esforço para a geração de código. De fato, metaprogramação pode capturar padrões de código, e aplicá-los em situações similares.

- Ela também tem o potencial de aumentar a eficiência de uma aplicação, porque o compilador pode executar parte do trabalho que de outra forma seria deixado a cargo do sistema de execução.

Há vários métodos para fazer metaprogramação. Vamos nos limitar, na presente discussão, aos fornecidos por Lisp.

Fechos de primeira classe. Um fecho (*closure* em inglês) é uma estrutura de dados que contém uma função e o ambiente em que ela é definida. Um exemplo clássico vai esclarecer esse ponto. Seja $\mathbb{N}$ o conjunto dos números naturais, $\mathfrak{int}$, o tipo associado a $\mathbb{N}$, e

$$\{\mathfrak{int}\ xx = 10; \mathbf{f} = \mathfrak{obj}\ (\mathfrak{int}\ x) \quad \{x + xx; \}; \}$$

a definição da função. A função **f** foi definida em um ambiente contendo a seguinte declaração: $\{\mathfrak{int}\ xx = 10 \ldots\}$. Portanto, não importa o valor de xx no momento em que **f** é utilizada. No corpo de **f**, xx será sempre a variável livre que foi declarada no ambiente onde **f** foi definida. Abaixo está uma possível implementação em C de **f**.

```
#include <stdio.h>

int call_clos(int fun(int), int x) {
int xx= 20; return(fun(x)); }

void main()
{ int xx=10;
  int foo(int x) { return (x+xx);}
  printf("%d\n", call_clos(foo, 5));}
```

Se alguém compilar e executar esse programa, a saída será 15, o que é compatível com `int xx=10`. O problema com C é que o ambiente não tem extensão indefinida. O ambiente é alocado na pilha e será destruído quando o sistema de execução retornar do procedimento contido na definição de **f**.

Em Lisp, *closures* são muito úteis para criação de abstrações de domínio específico porque o ambiente tem extensão indefinida, ou seja, podemos continuar a referir-nos a ele em qualquer momento no futuro. Lisp preservará o espaço ocupado por um ambiente até que o coletor de lixo prove que o programa não pode se referir a suas variáveis.

Lisp tem dois dialetos principais, Common Lisp e Scheme. Nossos exemplos são em Scheme. Em particular, usaremos o Gambit Scheme; entre outras razões para isso, o Gambit implementa melhor o *Revised(5) Report on the Algorithmic Language Scheme* e tem um *frontend* com uma sintaxe próxima do C.

```
> \ obj fuu;
> \{int xx=10; fuu= int (int y) {xx+y;};};
```

```
#<procedure #2 fuu>
> \ int xx= 20;
> (fuu 5)
15
```

Segue um exemplo de uso de *closures* para criar abstrações. Vamos definir uma função que cria neurônios artificiais que, colocados em uma rede, possam aprender como avaliar a temperatura e pressão parcial de CO_2; a função `new_neuron` usará *closures* para atingir seus objetivos.

```
obj new_neuron(double v, double ws) {

  double sig(double x) {
  1.0/(1.0+exp(-x)); };

  // The object below is a closure.
  double (obj xs) {
    double acc=0.0; // Comments
    obj i= ws;
    obj x= [1.0|xs];
    while (i!=[] && x!=[]) {
       acc= acc+ v[car(i)]*car(x);
       i=cdr(i);
       x=cdr(x);};
    sig(acc); }; };
```

Além da notação prefixa de Cambridge que fez a LISP ficar famosa, o (dialeto) Gambit tem também uma notação six, que é similar a do C. A *closure* `new_neuron` foi escrita em six; ela faz o que seu nome propõe, ou seja, ela cria uma função que age como um neurônio.

- `v` é um vetor que guarda os pesos.
- `ws` é uma lista contendo os índices em `v`; esses índices vão localizar as novas sinapses dos neurônios.
- O resultado de `new_neuron` é uma closure que calcula a saída de um neurônio definido em um ambiente onde o valor de `v` é conhecido

Pode-se usar `new_neuron` para construir uma rede neural. Assumindo essa rede neural como um modelo de portas lógicas. Abaixo encontramos o construtor de *closure*.

```
obj gate(obj vt) {
  obj input_1 = car;
  obj input_2 = cadr;
  obj neuron_1= new_neuron(vt, [4,5,6]);
  obj neuron_s= new_neuron(vt, [0,1,2,3]);
  obj (obj i) {
     if (i==[]) vt;
     else neuron_s([ input_1(i),
               neuron_1([ input_1(i),
                          input_2(i)]),
               input_2(i)]); }; };

obj xor = gate(vector(-5, -9, 19, -9, -4, 9, 9));
```

Podemos pensar em *closures* como sendo objetos com estados ocultos. Desde que podemos controlar a execução desses objetos, *closures* tornam possível a criação de novas abstrações e as linguagens que transmitem suas mensagens. De fato, um computador pode ser considerado uma unidade combinacional mais o controle. A unidade combinacional é representada pelas expressões funcionais. *Closures* nos permitem controlar a avaliação dessas expressões.

Várias linguagens de computação moderna oferecem ferramentas que modificam *closures* conforme a necessidade dos engenheiros de *software*. Todas as linguagens tem a mesma semântica no que diz respeito a *closures*. Existem três abordagens para trabalhar com a sintaxe da criação de *closures*; para ver a diferença entre elas, vamos considerar o problema da criação de linguagens de domínio especifico para o método de Newton-Raphson para encontrar aproximações sucessivamente melhores para as raízes de uma função de valor real.

Lazy evaluation. Linguagens como Clean e Haskell assumem que tudo é uma *closure*, que é avaliada quando se torna necessário. Essa estratégia é chamada *lazy evaluation.* Em um artigo que se tornou famoso, [John Hughes] mostra como esse esquema funciona para uma linguagem funcional *lazy*, como Clean e Haskell.

Sendo $x = x_0$ uma aproximação de uma raiz de $f(x) = 0$. A expansão de Taylor para $f(x)$ em x_0 pode ser escrita como:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots$$

Se guardarmos somente o termo de primeira ordem, a expansão acima se

torna:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

Se ajustarmos $f(x) = 0$ de forma a encontrarmos a próxima aproximação, x_1, a raiz, encontramos:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{3.1}$$

Em clean, o programa que implementa o algoritmo especificado pela equação 3.1 é dado na figura 3.1.

A função `next_root_approx f df x0` tem três argumentos; o primeiro argumento é a função `f` cujas raízes procuramos; o segundo argumento é a derivada de `f`; o terceiro argumento é uma aproximação da raiz. O único aspecto diferente dessa definição é que a função pode ser passada como argumento para outra função. De fato, `f` e `df` são argumentos de `next_root_approx`.

Vamos especificar `next_root_approx f df x0` para o caso das raízes

```
module newton
import StdEnv

next_root_approx f df x0
  = x0 - (f x0)/(df x0)

next_approx_sqrt n
  = next_root_approx
      (\x-> x*x - n)
      (\x-> 2.0 * x)

series f x = [x:series f (f x)]

square_root n x0=
   limit (series
           (next_approx_sqrt n)
           x0)

Start = square_root 4.0 0.5
```

Figura 3.1: Closures in a lazy functional language

quadradas. Se precisamos da raiz quadrada de a, devemos encontrar a raiz da equação $x \cdot x - a = 0$; que devolve $f = \lambda(x) \rightarrow x \cdot x - a$. Em lógica matemática e ciência da computação, cálculo-λ é um sistema para definição de funções; nesse sistema uma abstração $\lambda(x) \rightarrow \mathrm{E}(x)$ é uma função cujo parâmetro formal é x. A derivada de f é dada por $df = \lambda(x) \rightarrow 2.0 \cdot x$. Não existindo na maioria dos teclados de computador o símbolo λ, ele é trocado por uma barra invertida. Na listagem 3.1, O aspecto extraordinário de `next_approx_sqrt n` é que ele foi obtido fornecendo dois de três parâmetros presentes da definição de `next_root_approx f df x0`. Em Haskell ou Clean, e outras linguagens funcionais, se parte dos parâmetros são fornecidos, o compilador retorna uma função dos parâmetros que restam; a *closure* dessa função contém ponteiros para os valores fornecidos. Nesse caso, a *closure* conhece as funções $(\lambda(x) \rightarrow x \cdot x - n)$ e $(\lambda(x) \rightarrow 2.0 \cdot x)$, que ela precisa para calcular as aproximações da raiz quadrada.

A função `series f x = [x:series f (f x)]` cria uma série infinita de aproximações da raiz. Sua excepcional característica é que ela trabalha com recursão sem condição de interrupção. Ela simula uma série matemática infinita. Isso é possível em Haskel e Clean porque elas são linguagens *lazy*, que calculam um valor somente quando ele é necessário. Dessa forma, `series f x` não entrará em *loop* infinito. A raiz de `f(x)` é o limite de `series f x`. Embora `limit` seja fornecido pelo ambiente padrão, sua definição é dada abaixo.

```
limit::!.[a] -> a | Eq a
limit [a:cons=:[b:x]]
| a==b = a
| otherwise
        = limit cons
limit other = abort "incorrect use of limit"
```

O exemplo mostra que o uso de *closures* e avaliação *lazy*, torna possível a criação de excelentes abstrações no lado do problema, que é o ponto dessa discussão.

Blocos. Existem linguagens, como Lisaac e Self, que representam *closures* como blocos de C. Essas *closures* são tratadas como funções, que podem ser aplicadas em argumentos e avaliadas por um *slot* obrigatório.

Self e Lissac são linguagens de programação orientadas a objeto, baseadas no conceito de protótipos. Ambas linguagens usam blocos *closure*. Self é uma linguagem de altíssimo nível orientada a objeto que executa quase que

na metade da velocidade da C otimizada. Lisaac é semelhante a Self, mas sua intenção é executar tão rápido quanto C otimizada; ela gera executáveis pequenos e rápidos. Lisaac foi usada em campo na implementação de um sistema operacional completo.

Na implementação em Lisaac do algoritmo de Newton-Raphson, o *slot* `approximation` possui três argumentos; o primeiro argumento é uma *closure* que representa `f`; o segundo argumento é uma closure para `df`, ou seja, a derivada de `f`. O primeiro argumento é um valor real, que será usado para calcular uma melhor aproximação para a raiz. O *slot* `approximation` é semelhante a função `next_root_approx f df x0` usada na implementação em Clean do algoritmo.

A abstração em Lisaac `series` é equivalente a `series` em Clean. Entretanto, Clean avalia os elementos da série conforme é necessário, enquanto

```
Section Header
  +name := NEWTON;
Section Public
  -approximation
     ( f, df:{REAL; REAL},
       x:REAL) : REAL <- (
     x - (f.value x)/ (df.value x) );
   -arg i:INTEGER : REAL_64 <-
     (COMMAND_LINE.item i).to_real_16_16;
  - series condition:{BOOLEAN} next body:{}  <-
       ( condition.value
          .if { body.value;
             series condition next body;}; );
  -main <- ( +f, df : {REAL; REAL};
    +a, xi : REAL;
    a := arg 1;
    xi := a/2.0;
    f := {x:REAL; x*x - a};
    df := {x:REAL; 2.0 * x};
    series {(xi*xi - a).abs > 0.01}
      next {xi := approximation (f, df, xi);};
    xi.print_format_c " %8.4f "; );
```

Figura 3.2: Lisaac and Self — Closure blocks

Lisaac e Self requerem uma chamada explicita para avaliação, que é alcançada no slot `value`, que é herdado pelos blocos *closure*.

Macros. Linguagens semelhantes ao Lisp, como Scheme, podem criar uma *closure* simplesmente empacotando a expressão em uma abstração lambda. Em particular, uma *closure* sem argumentos pode ser alcançada com uma expressão lambda sem argumentos. Por exemplo,

Na figura 3.3, podemos ver a solução da Lisp para o problema da criação de uma abstração do problema de Newton-Raphson. A *closure* `expand-series` manipulando funções executa o mesmo papel de `series` em Lisaac e Clean. Entretanto, em Clean, avaliação *lazy* torna a manipulação de *closures* transparente; de fato, manipulação de funções *closure* são iguais a quaisquer outras funções presentes na linguagem. Mesmo em Lisaac, uma notação simples e concisa faz a manipulação de *closure* bastante transparente. Em Lisp, a necessidade de empacotar *closures* em expressões lambda é um requisito estranho. Felizmente, Lisp possui macros, que podem executar o empacotamento automaticamente. Na listagem abaixo, a macro cria uma abstração `series` que manipula *closures* Lisp tão bem quanto Lisaac ou Clean.

```
(define (approximation f df x)
   \x - f(x)/df(x);)

(define (expand-series condition body)
   (if (condition)
     (begin (body) (expand-series condition body) )) )

(define (root a)
   (let ( (xi  \a/2.0;)
          (f \double (double x) {x*x - a;};)
          (df \double (double x) {2.0*x;}; ) )
     (expand-series
        (lambda() \ (xi*xi - a) > 0.1;)
        (lambda() (set! xi (approximation f df xi)) ))
     xi))
```

Figura 3.3: Lisp — Lambda closures

3.8 Desenvolvimento cooperativo

Vimos que linguagens funcionais e linguagens orientadas a objetos modernas podem criar boas abstrações facilmente. Entretanto, devemos entender, não há muitas pessoas querendo aprender e programar em Lisp ou Haskell, muito menos em Self ou Lisaac. Essas linguagens estão muito longe do caminho principal da programação.

Acreditamos que qualquer linguagem que cumpra a lista aparentemente obvia dada acima vai parecer muito estranha para a maioria das pessoas. Vamos trabalhar nesse tópico.

Consideremos a abstração na ponta certa do problema. A única forma de alcançar esse objetivo é criando extensões da linguagem conforme as necessidades aparecem. Uma linguagem que pode ser facilmente manipulada e moldada em diferentes *avatares* deve usar o mesmo formato para representar programas e dados, ou ao menos permitir a manipulação de blocos de construção funcional. Afinal de contas, programas serão os dados para programas que irão modelar Lisp ou Haskell em uma linguagem de domínio especifico.

```
(define (make-approximation f df)
  (lambda(x)
     \x - f(x)/df(x);))

(define (square-root a)
   (make-approximation
      (lambda(x) \x*x - a; )
      (lambda(x) \2.0*x; ) ))

(define-macro (series x0 condition nxt ix)
   `(let loop ( (xx ,x0) )
       (if ((lambda(,ix) ,condition) xx) xx
           (loop (,nxt xx)) )) )

(define (root a)
     (series \a/2.0;
             \ abs((x*x - a)) < 0.001;
              (square-root a) x) )
```

Figura 3.4: Lisp — Lambda closures

Uma das abstrações de dados da Lisp é a lista, que pode ser vista como uma coleção ordenada de elementos; por exemplo:

```
(3 4 5 6 7 8) ;; a vector
```

Certamente vetores não são a única entidade que podemos representar com listas. Se usarmos listas de listas, podemos representar uma matriz bidimensional.

```
((2 3 4) (5 6 7)) ;; uma matriz
```

Os elementos de uma lista podem ser *symbols*, ao invés de números. E eles também não precisam ser do mesmo tipo.

```
(Lisp Java C C#) ;; coleção de linguagens
(Violeta Mimi Musetta Liu) ;; heroínas de opera
((Lisp 1.5) (Java 5) C++)
```

Várias outras linguagens possuem listas, sendo listas uma forma muito flexível de representar nosso conhecimento sobre o mundo. O problema é que Lisp representa seus programas como listas, o que nenhuma outra faz. Isso é necessário, porque programadores Lisp manipulam seus programas com as mesmas ferramentas criadas para processar listas como abstrações de dados. Isso torna o visual de um programa em Lisp bastante estranho para a maioria dos programadores. Uma multiplicação, por exemplo, é uma lista cujo primeiro elemento é o símbolo de multiplicação. Eis algumas operações expressas em LISP:

```
Gambit v4.6.0

> (* 3 4 5 6)
360
> (+ (* 3.1416 5 5) (* 2 3.1416 5 8))
329.868
> (exit)
```

Já observamos que LISP pode ativar macros para notação infixa sempre que esse tipo de convenção torna-se necessário ou conveniente. A realidade, contudo, é que programas em LISP não fazem uso da notação infixa com grande frequência.

Uma conclusão que podemos tirar da discussão acima é que LISP pode ser considerada a mais flexível das três famílias de linguagens com recursos para

criação de modelos abstratos no domínio do problema. A outra conclusão é que a maioria dos profissionais não usam LISP; além disso, esses profissionais não mostram a menor inclinação para trabalhar com Haskell, Clean, Lisaac ou Self. Essa afirmação talvez surpreenda o leitor, visto que a maioria dos sistemas importantes são escritos em LISP. O que podemos dizer é que a predominância da LISP apenas mostra sua superioridade sobre as outras linguagens: Mesmo tendo poucos adeptos, LISP domina o mercado.

A fim de obter aceitação, ferramentas pouco usadas precisam prestar serviço a linguagens e metodologias da vertente principal. Por exemplo, C tornou-se popular devido à enorme quantidade de bibliotecas que oferece a outras linguagens. Não interessa a linguagem que os gerentes de projeto escolham; os programadores acabarão por conectar seus procedimentos a bibliotecas em C e usá-las.

Lisaac pode seguir facilmente a rota traçada por C. O compilador de Lisaac gera código compacto e rápido. Além disso, esse projeto francês foi testado em campo através da implementação de um sistema operacional. Lisaac possui várias bibliotecas que poderiam ser úteis aos programadores de outras linguagens. Quanto a LISP, ela é uma linguagem rápida, mas requer um ambiente gigantesco para ser executada. Evidentemente, há versões da Scheme que empacotam um pequeno runtime na distribuição, o que torna possível usar a mesma estratégia que propusemos para Lisaac.

Outra solução para trabalho colaborativo em LISP é o uso da linguagem como um pre-processador capaz de gerar bibliotecas em C, ou como script para bibliotecas em C. Essa abordagem é bastante popular: Autocad, Emacs, Gimp, Maxima e até mesmo o compilador gcc utilizam uma forma ou outra de LISP como pre-processador, linguagem intermediário ou linguagem de *script*.

Capítulo 4

Descrição de Documentos

Uma das primeiras linguagens de descrição e estruturação de documentos foi a LaTeX, uma linguagem de anotação (inglês, *markup language*) e sistema de preparação de documentos para o TeX, uma ferramenta de tipografia eletrônica.

A SGML (inglês, *Standard Generalized Markup Language*) é uma linguagem de marcação geral, baseada em dois postulados:

- Anotação deve descrever a estrutura do documento e outros atributos, em vez de especificar o processamento a ser realizado sobre ele.
- Anotação precisa de ser rigorosa, de modo que ferramentas para processar objetos rigorosamente definidos, como programas e base de dados, possam ser aplicadas a documentos.

4.1 HTML

HTML é uma linguagem de anotação desenvolvida por Tim Berners-Lee, em paralelo com SGML. A intenção de Berners-Lee foi desenvolver uma aplicação da SGML. Assim, embora HTML (Hyper Text Markup Language) tenha recebido inspiração do processo de rotulação da SGML, a maior parte dos documentos descritos em HTML não são válidos em SGML. Mais tarde, HTML foi reformulada (versão 2.0) para aproximar-se da SGML. Mesmo assim, várias discrepâncias permaneceram. Finalmente, surgiu HTML-4, uma aplicação em SGML que se enquadra completamente na padronização ISO 8879-SGML. HTML-4 possui suporte a ferramentas tais como folhas de estilo (inglês, *style sheets*), SCRIPTA (inglês, SCRIPTING), FRAMES, objetos embarcados (*embedding objects*. HTML-4 também melhora recursos necessários

para preparar textos escritos da direita para a esquerda, como é o caso do árabe e do hebraico.

4.2 Introdução a XML e XSL

Existem dois paradigmas de programação para a Web: Funcional e Procedural. O paradigma procedural é representado por linguagens como PHP e Javascript. Este paradigma não nos interessa e não vamos discuti-lo por enquanto.

O paradigma funcional começou com a linguagem DSSSL, um dialeto da LISP. Nos últimos cinco anos, a sintaxe desta linguagem foi modificada. Vamos continuar este capítulo estudando as referidas modificações.

4.3 S-expressions e XML

Linguagens do tipo LISP e Scheme representam dados como *s-expressions*. Uma s-expression pode ser um valor atômico ou uma lista. Por definição, não podemos usar seletores para acessar as partes de um valor atômico. Afinal, atômico significa sem partes em grego. Em Scheme, os valores atômicos são os números e os símbolos (sequências de dígitos e letras).

As listas são sequências ordenadas de elementos. No Scheme e no LISP, os elementos das listas são colocados entre parênteses e podem ser acessados por dois seletores: `car` e `cdr`. Exemplos de listas:

(3 4 5 6 7)
(rosa cravo violeta orquídea)
(lápis caderno livro borracha caneta)

O seletor `car` acessa o primeiro elemento de uma lista. Por exemplo, se `xs=(3 4 5 6 7)`, então `(car xs)` é igual a `3`. O seletor `cdr` acessa a parte da lista que vem depois do primeiro elemento; assim, `(cdr xs)` é igual a `(4 5 6 7)`.

Uma coisa que torna LISP e Scheme interessantes é que tanto programas quanto dados são representados por s-expressions. Os dados necessários para resolver seus problemas são listas... e os programas também. Esta caraterística não é compartilhada com nenhuma outra linguagem de programação, exceto XSLT. Representar dados da mesma forma que programas permite escrever programas que manipulam programas, um recurso que torna LISP tão poderosa.

Em XSLT, dados são representados por elementos XML, que chamaremos de XML-elems. Um XML-elem é constituído por texto ou outros XML-elems colocados entre `<id>` e `</id>`, onde `id` é o identificador do elemento. No exemplo abaixo, `<aluno>...</aluno>` é um XML-elem cujo identificador é `aluno`. As partes de `<aluno>...</aluno>` são dois outros XML-elem, a saber, `<nome>...</nome>` e `<nota>...</nota>`. Símbolos como `<nome>` e `<aluno>` são chamados de abre elem, por analogia com o abre parênteses do LISP; por outro lado, `</nome>` e `</aluno>` são chamados de fecha elem.

```
<aluno>
   <nome>Rosa de Luxemburg</nome>
   <nota>15</nota>
</aluno>
```

Aqui vem uma coisa interessante: Os programas em XSLT também são constituídos de XML-elems. Estamos, portanto, na mesma situação do LISP e do Scheme: Programas e dados são representados pela mesma estrutura.

Um XML-elem, além do identificador, pode possuir atributos. Um atributo é constituído de um rótulo associado a um valor pelo sinal de igualdade. No exemplo abaixo, o XML-elem `xsl:for-each` tem um atributo `select` cujo valor é `"clean/aluno"`.

```
<xsl:for-each select="clean/aluno">
      <tr>
        <td><xsl:value-of select="nome"/></td>
        <td><xsl:value-of select="nota"/></td>
      </tr>
</xsl:for-each>
```

No exemplo acima, você deve ter notado, que alguns XML-elems não têm fecha elem. De fato, se não existe nada entre o abre elem e o fecha elem, você pode escrever o XML-elem de forma abreviada. Desta forma, a seguinte expressão `<xsl:value-of select="nome"/>` é uma forma abreviada de se escrever:

```
<xsl:value-of select="nome"></xsl:value-of>
```

4.4 Pattern match

Toda linguagem funcional que se preza tem *pattern match*, ou seja, casamento de padrões. No *pattern match*, um *template* (gabarito ou padrão) é comparado com uma estrutura de dados; se o padrão coincidir com uma parte da estrutura, a função produz um valor adequado. Vejamos um exemplo. Na figura 4.1, apresentamos dados sobre o desempenho dos alunos de programação funcional. Na figura 4.2, temos um programa que processa os dados.

A linguagem XSLT é uma linguagem funcional pura, ou seja, não tem efeitos colaterais. Na internete, isto é necessário porque o processamento é feito em paralelo, por milhares de máquinas.

O programa da figura 4.2 aplica uma linha de uma tabela html em cada elemento `aluno` da lista de alunos de `clean`. A aplicação é realizada pela função `for-each`:

```
<xsl:for-each select="clean/aluno">
    <tr>
       <td><xsl:value-of select="nome"/></td>
       <td><xsl:value-of select="nota"/></td>
    </tr>
</xsl:for-each>
```

O elemento `<xsl:template>` é usado para construir padrões, ou seja, gabaritos. A palavra *template* significa *gabarito*, em inglês. O atributo `match` associa um `template` a um elemento da estrutura de dados no formato `XML`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="for-each-notas.xsl"?>
<clean>
  <aluno>
    <nome>Rosa de Luxemburg</nome>
    <nota>15</nota>
  </aluno>
  <aluno>
    <nome>Thales de Mileto</nome>
    <nota>10</nota>
  </aluno>
  <aluno>
     <nome>Philippos</nome>
     <nota>14</nota>
  </aluno>
  <aluno>
     <nome>Frederico Chopin</nome>
     <nota>9</nota>
  </aluno>
</clean>
```

Figura 4.1: Dados para o programa de notas: xml-for-each-notas.xml

Tudo no formato `XML` começa com `<fn>` e termina com `<\fn>`, conforme mostrado na figura 4.1. O valor de `match` é uma expressão `XPath` (caminho na internete); no exemplo, `match="/"` define todo o documento, que deve estar no formato `XML`.

Em Scheme, os dados são listas e os programas são listas também. A linguagem funcional XSLT herdou esta caraterística da Scheme (da qual é filha). Os dados são estruturas `XML` e os programas também são estruturas `XML`. Isto significa que podemos escrever programa que escrevem programas. Abaixo apresentamos o resultado do programa da listagem 4.2.

Notas de Programação Funcional

Nome	Nota
Rosa de Luxemburg	15
Thales de Mileto	10
Philippos	14
Frederico Chopin	9

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
    <h2>Programação funcional: Turma 42</h2>
    <table border="1">
    <tr bgcolor="#acacac">
      <th align="left">Nome</th>
      <th align="left">Nota</th>
    </tr>
    <xsl:for-each select="clean/aluno">
      <tr>
        <td><xsl:value-of select="nome"/></td>
        <td><xsl:value-of select="nota"/></td>
      </tr>
    </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Figura 4.2: Programa em XSLT: for-each-notas.xsl

4.5 O elemento xsl:value-of

O elemento `<xsl:value-of>` é utilizado para obter o valor de um elemento XML e inserir o referido valor na `stream` de saída. O valor de um elemento `<fn> vvvv </fn>` é tudo que está entre `<fn>` e `</fn>`. Por exemplo, o valor de `"nome"` no elemento abaixo

```
<aluno>
    <nome>Eika Cunha</nome>
    <nota>15</nota>
</aluno>
```

é `Eika Cunha` e o valor de `"nota"` é `15`.

4.6 O elemento xsl:for-each

O elemento `<xsl:for-each>` é utilizado para aplicar uma transformação a todas as partes de um XML-elem e introduzir o resultado na `stream` da internete. Por exemplo, as linhas abaixo constroem uma linha de tabela para cada elemento `<aluno>...</aluno>` de `<clean>...</clean>`.

```
<xsl:for-each select="clean/aluno">
    <tr>
       <td><xsl:value-of select="nome"/></td>
       <td><xsl:value-of select="nota"/></td>
    </tr>
</xsl:for-each>
```

4.7 Filtros e ordenação

É possível aplicar filtros aos elementos obtidos por `xsl:for-each`. Os filtros possíveis são dados abaixo.

```
elem= val               Igual
elem != val             Diferente
elem &gt; val           Maior
elem &lt; val           Menor
elem >= val             Maior ou igual
```

As figuras 4.3 e 4.4 apresentam um exemplo de filtros e também de ordenação. De fato, o elemento `<xsl:sort select="nome"/>` coloca a saída em ordem.

Veja o resultado da listagem 4.3 e 4.4 na figura a seguir.

Programação funcional

Nome	Nota
Anna Comnena	12
Philippos	14
Rosa de Luxemburg	15
Thales de Mileto	10

4.8 Condições

O XML-elem `<xsl:if test="sold > 500">` coloca seu valor na *stream* da internete apenas se a condição expressa pelo atributo `test` for satisfeita. No exemplo das figuras 4.5 e 4.6, um livro só entra para a lista de bestsellers se vender mais de 500 cópias, o que para o Brasil é muita coisa.

4.9 Condições múltiplas

Condições com mais de uma entrada podem ser expressas pelo elemento `<xsl:choose>`. Ver figuras 4.8 e 4.7.

4.10 Templates

A última coisa que nos falta para aprender é templates. Um template é, basicamente, aplicado a todos os elementos do XML-elem que tiver sucesso no *match*. Por exemplo, na figura 4.9, página 48, usamos um template para mostrar todas as informações sobre a tabela de vinhos.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html><body>
    <h2>Programação funcional: Turma 42</h2>
    <table border="1">
    <tr bgcolor="#acacac">
      <th align="left">Nome</th> <th align="left">Nota</th>
    </tr>
    <xsl:for-each select="clean/aluno[nota &gt; '13.5']" >
      <xsl:sort select="nome"/>
      <tr>
        <td><xsl:value-of select="nome"/></td>
        <td><xsl:value-of select="nota"/></td>
      </tr>
    </xsl:for-each>
    </table>
  </body> </html>
</xsl:template>
</xsl:stylesheet>
```

Figura 4.3: Filtros e ordenação: filtro-notas.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="filtro-notas.xsl"?>
<clean>
     <aluno>
       <nome>Rosa de Luxemburg</nome>
       <nota>15</nota>
     </aluno>
     <aluno>
       <nome>Thales de Mileto</nome>
       <nota>10</nota>
     </aluno>
     <aluno>
        <nome>Philippos</nome>
        <nota>14</nota>
     </aluno>
     <aluno>
        <nome>Frederico Chopin</nome>
        <nota>9</nota>
     </aluno>
     <aluno>
        <nome>Anna Comnena</nome>
        <nota>12</nota>
     </aluno>
</clean>
```

Figura 4.4: Mais notas: xml-filtro-notas.xml

4.11 Comentários

Em XSLT, comentários são colocados entre `<xsl:comment>` e `</xsl:comment>`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <xsl:comment> match="/" unifica-se com todo o documento.
     Esta listagem está no arquivo books.xsl
  </xsl:comment>
```

```
  <html> <body>
    <h2>Livros Mais Vendidos</h2>
    <table border="1">
      <tr> <th>Título</th> <th>Autor</th> </tr>
      <xsl:for-each select="livros/bk">
      <xsl:if test="sold &gt; 500">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="author"/></td>
        </tr>
      </xsl:if>
      </xsl:for-each>
    </table>
  </body> </html>
</xsl:template>
</xsl:stylesheet>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="books.xsl"?>
<livros>
  <bk>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html><body> <h2>Livros Mais Vendidos</h2>
    <table border="1">
      <tr bgcolor="#9acd32"> <th>Título</th> <th>Autor</th> </tr>
      <xsl:for-each select="livros/bk">
      <xsl:if test="sold &gt; 500">
        <tr> <td><xsl:value-of select="title"/></td>
             <td><xsl:value-of select="author"/></td> </tr>
      </xsl:if>
      </xsl:for-each>
    </table>
  </body></html>
</xsl:template>
</xsl:stylesheet>
```

Figura 4.5: Imprime lista de bestsellers: if-books.xsl

```
    <title>Os Três Mosqueteiros</title>
    <author>Alexandre Dumas</author>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="books.xsl"?>
<livros>
  <bk>
    <title>Os Três Mosqueteiros</title>
    <author>Alexandre Dumas</author>
    <price>200</price>
    <sold>2000</sold>
  </bk>
  <bk>
    <title>A Dama das Camélias</title>
    <author>Alexandre Dumas Filho</author>
    <price>200</price>
    <sold>600</sold>
  </bk>
  <bk>
    <title>Odisséia</title>
    <author>Homero</author>
    <price>150</price>
    <sold>800</sold>
  </bk>
  <bk>
    <title>Poesias</title>
    <author>Olavo Bilac</author>
    <price>30</price>
    <sold>8</sold>
  </bk>
</livros>
```

Livros Mais Vendidos

Título	Autor
Os Três Mosqueteiros	Alexandre Dumas
A Dama das Camélias	Alexandre Dumas Filho
Odisséia	Homero

Figura 4.6: Base de dados da livraria: xml-if-books.xml

```
    <price>200</price>
    <sold>2000</sold>
  </bk>
  <bk>
    <title>A Dama das Camélias</title>
    <author>Alexandre Dumas Filho</author>
    <price>200</price>
    <sold>600</sold>
  </bk>
  <bk>
    <title>Odisséia</title>
    <author>Homero</author>
    <price>150</price>
    <sold>800</sold>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="wines.xsl"?>
<vinhos>
  <bt> <marca>Baron Philippe de Rothschild</marca>
       <price>500</price> <safra>1980</safra> </bt>
  <bt> <marca>Ibrapa</marca>
       <price>30</price><safra>2003</safra> </bt>
  <bt> <marca>Chateau Duvalier</marca>
       <price>20</price> <safra>2002</safra> </bt>
  <bt> <marca>Veuve Clicquot</marca>
       <price>200</price><safra>2001</safra> </bt>
</vinhos>
```

Vinhos franceses e brasileiros

Origem	Safra	Avaliação
Baron Philippe de Rothschild	1980	*******
Ibrapa		**
Chateau Duvalier		**
Veuve Clicquot	2001	*******

Figura 4.7: Vinhos bons e ruins: xml-choose-wines.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html> <body>
    <h2>Vinhos franceses e brasileiros</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Origem</th> <th>Safra</th> <th>Avaliação</th>
      </tr>
      <xsl:for-each select="vinhos/bt">
      <tr>
        <xsl:choose>
           <xsl:when test="price > 150">
             <td><xsl:value-of select="marca"/></td>
             <td><xsl:value-of select="safra"/></td>
             <td>******</td>
           </xsl:when>
           <xsl:when test="price > 50">
          <td><xsl:value-of select="marca"/></td>
          <td><xsl:value-of select="safra"/></td>
          <td>****</td>
           </xsl:when>
           <xsl:otherwise>
             <td><xsl:value-of select="marca"/></td>
             <td></td>
             <td>**</td>
           </xsl:otherwise>
        </xsl:choose>
      </tr>
      </xsl:for-each>
    </table>
  </body> </html>
</xsl:template>
</xsl:stylesheet>
```

Figura 4.8: Sommelier Uberlandense: choose-wines.xsl

```
  </bk>
   <bk>
      <title>O Vermelho e o Negro</title>
      <author>Stendhal</author>
      <price>38</price>
      <sold>5</sold>
  </bk>
  <bk>
    <title>Poesias</title>
    <author>Olavo Bilac</author>
    <price>30</price>
    <sold>8</sold>
  </bk>
</livros>


<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html> <body>
  <h2>Vinhos franceses e brasileiros</h2>
  <xsl:apply-templates/>
</body> </html>
</xsl:template>
<xsl:template match="bt">
  <p> <xsl:apply-templates select="marca"/>
      -- <xsl:apply-templates select="safra"/> <br />
      Preço: <span style="color:#ff0000">
               <xsl:value-of select="price"/>
             </span> </p>
</xsl:template>
</xsl:stylesheet>
```

Figura 4.9: Carrefour de Uberlândia: apply-templates-vins.xsl

4.12 Geração de elementos

`<xsl:element>` gera elementos durante a execução. No programa da figura 4.11, diferentes cabeçalhos de html são aplicados nos textos.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="vins.xsl"?>
<vinhos>
  <bt>
    <marca>Baron Philippe de Rothschild</marca>
    <price>500</price>
    <safra>1980</safra>
  </bt>
  <bt>
      <marca>Sangue de Boi</marca>
      <price>10</price>
      <safra>2008</safra>
  </bt>
  <bt>
      <marca>Ibrapa</marca>
      <price>30</price>
      <safra>2003</safra>
  </bt>
  <bt>
      <marca>Chateau Duvalier</marca>
      <price>20</price>
      <safra>2002</safra>
  </bt>
  <bt>
      <marca>Veuve Clicquot</marca>
      <price>200</price>
      <safra>2001</safra>
  </bt>
</vinhos>
```

Figura 4.10: Vinhos bons e ruins: xml-apply-templates-vins.xml

```
<xsl:stylesheet version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>

<xsl:template match="/">
  <html>
  <body>
     <xsl:for-each select="//text">
          <xsl:element name="{@size}">
               <xsl:value-of select="."/><br/>
          </xsl:element>
     </xsl:for-each>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Figura 4.11: eleme.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="eleme.xsl"?>
<source>

<text size="H1">Header1</text>
<text size="H3">Header3</text>
<text size="b">Bold text</text>
<text size="sub">Subscript</text>
<text size="sup">Superscript</text>
<text size="i">Texto em itálico</text>

<text size="H2">
  Muita água rolou por debaixo da ponte.
</text>
</source>
```

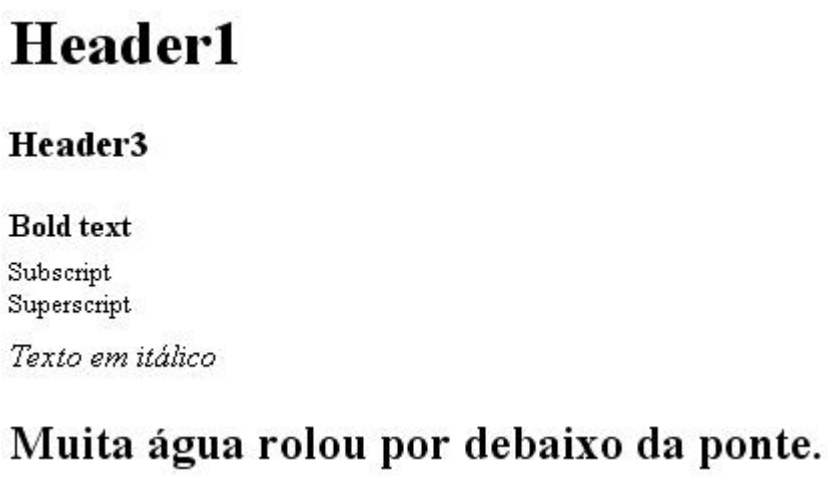


Figura 4.12: xml-elem.xml

Capítulo 5

Funções

Uma função é recursiva se aparece na própria definição. Seja, por exemplo, a função fatorial. Quando o parâmetro é 0, o fatorial é 1:

```
<xsl:when test="$number = 1">
      1
</xsl:when>
```

Quando o parâmetro do fatorial não é 0, a opção `xsl:otherwise` do elemento `xsl:choose` é executada. Inicialmente, cria-se uma variável `"f1"` para cujo valor será o fatorial de `$number - 1`. Para calcular o valor de `"x"`, a função `fatorial` é chamada recursivamente.

```
<xsl:otherwise>
      <xsl:variable name="f1">
        <xsl:call-template name="factorial">
           <xsl:with-param name="number" select="$number - 1" />
        </xsl:call-template>
      </xsl:variable>
      <xsl:value-of select="$number * $f1" />
</xsl:otherwise>
```

Observe que na linguagem `xslt`, o cálculo do valor de uma variável é colocado entre `<xsl:variable name="x">` e `</xsl:variable>`; uma solução original e elegante de atribuir valores a variáveis. Mas voltemos ao problema de calcular o fatorial. Já sabemos o fatorial `"f1"` de `$number - 1`. O fatorial de `$numer` é dado por `<xsl:value-of select="$number * $f1" />`.

Na listagem 5.2, você deve ter notado que a função fatorial é definida fora do `<xsl:template match="/"> ... </xsl:call-template>`. Nisto, XSLT não é diferente de outras linguagens, onde as funções e variáveis globais são, em geral, definidas fora da função principal.

Neste e em outros capítulos, utilizamos elementos sem maiores explicações. Além disso, admitimos que o leitor conhecesse um pouco de html. Quando aos elementos ainda não estudados do XSLT, vamos falar um pouco sobre eles neste capítulo.

O elemento `<xsl:text> xabca </xsl:text>` representa uma *string* (cadeia de caracteres). Ao contrário de outras linguagens, XSLT aceita a *string* tal qual você a escreveu. Isto significa que, se você colocar uma *quebra de linha* no texto, ela não só será aceita, como aparecerá na página da internete.

O elemento `<xsl:value-of select="3 * 4" />` não só permite efetuar cálculos e retirar valores de outros elementos. Para acessar o valor do laço `<xsl:for-each ...>`, o atributo `select` deve ser igual a `"."`.

5.1 Formulários

Tudo que fizemos até agora permite apresentar dados ao internauta. Se a internete só servisse para exibir informações, ela seria, nas palavras do poeta

português Manoel Maria du Bocage, uma ferramenta de se ver, não de se usar. Entretanto, a internete permite a comunição entre o internauta e a página XML. Vamos ver como é realizada a comunicação.

Há duas classes de sistemas computacionais na internete: os servidores e os clientes. Os servidores oferecem serviços, tais como gerenciamento de bases de dados, cálculos, buscas, curso, etc. Os clientes usam estes serviços. Assim sendo, devemos escrever dois tipos de programas:

- Programas do lado do servidor. Estes programas vão receber os pedidos dos clientes e realizá-los. Por exemplo, um programa do lado do servidor pode consultar uma base de dados, enviando o resultado ao cliente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="fat.xsl"?>
<source>

<number>40</number>
<number>5</number>
<number>6</number>


</source>
```

Tabela de fatoriais

Fatorial de 40 = 815915283247897700000000000000000000000000000000
Fatorial de 5 = 120
Fatorial de 6 = 720

Figura 5.1: xml-fat.xml

- Programas clientes. Estes programas ficam na máquina do internauta, apresentando-lhe formulários de requisição que, devidamente preenchidos, serão enviados para o sítio onde será prestado o serviço requisitado.

O leitor perspicaz perceberá imediatamente que características devem ter as linguagens utilizadas para escrever programas do servidor. Em primeiro lugar, estas linguagens devem ser incrivelmente rápidas. Um sítio como a Orbitz recebe milhões de pedidos mensais. Se não trabalhasse com uma linguagem agressivamente rápida, simplesmente sairia do ar por negação de serviço. Além de rápida, a linguagem servidora deve ser fortemente tipada, para detetar erros de tipo em tempo de compilação. Se a linguagem servidora detetar erros apenas na execução, e fizer a deteção de tipo dinamicamente, o sistema fatalmente falharia. Com milhões de usuários requisitando serviços, dificilmente um deles não enviaria um tipo errado, que derrubaria o sistema. Existe até a possibilidade de internautas malévolos provocar a negação de serviço (*Denial of Service attack* or DoS for short). Infelizmente, empresas e universidades não estão atentas para este problema. Escrevem os programas servidores em linguagens não tipadas e lentas, como PHP, ASP, Python e Javascript. O resultado é sistema lento e frequentes negações de serviço.

No lado do cliente, o programa deve ser não tipado e interpretado. Em resumo, o programa do cliente deve ser um *script*. Afinal, não podemos exigir que uma secretária, ou um blogueiro, saiba teoria de tipos, cálculo lâmbda, etc. Existem vários tipos de linguagens de script, cujos programas são executados no lado do cliente: Javascript, Java e XSLT. Destas, a mais adequada é XSLT, por várias razões:

1. XSLT é oficial. Isto significa que ela é a linguagem de *script* recomendada e oficialmente apoiada pelos administradores da *World Wide Web* (a organização que torna possível páginas na internete, blogs, emails, etc.). Sendo oficial, XSLT sempre vai funcionar impecavelmente na *Web* e estará sempre atualizada. Você deve ter notado que o html é uma estrutura de dados da XSLT, ou seja, uma página em html é um elemento XML.

2. XSLT é funcional e tem uma sintaxe interessante, flexível e simples, parecida com a do LISP.

3. XSLT é totalmente integrada e compatível sintaxicamente com outras ferramentas da Web, como o já citado html.

5.2 Lado do servidor

Muita gente pensa que programas do lado do servidor devem ser escrito em linguagens especiais para servidor, como PHP ou Pliant. Na realidade, a melhor linguagem para o lado do servidor é Clean, porque além de ser rápida e fortemente tipada, captura a maior parte dos erros em tempo de compilação. Talvez você pense que Clean é muito difícil de programar, por exigir um cérebro matemático. Entretanto, os programas do lado do servidor são de tal maneira simples que dificilmente provocarão superaquecimento do cérebro. De fato, até um anfioxo tem o cérebro bastante desenvolvido para programar do lado do servidor, se ele se interessasse por este gênero de trabalho. Assim, não há desculpa para escrever programas servidores em

PHP ou Python.

A comunicação entre um programa do lado do servidor e o cliente da internete faz-se pelo protocolo *Common Gate Interface*, ou CGI, para os íntimos. No CGI, existem dois métodos de comunicação, *get* e *post*. Vamos aprender o método *get*.

Como programação funcional é obrigatória em praticamente todos os cursos de computação do Brasil e do mundo, vou supor que todos aqui saibam como programar em Clean.

Para usar o método *get* em Clean, é preciso importar a biblioteca `ArgEnv`. O formulário enviado pelo cliente está na variável `"QUERY_STRING"` e é obtido pela expressão abaixo.

```
# xx= ans (getEnvironmentVariable "QUERY_STRING");
```

Os campos do formulário são separados uns dos outros pelo caracter `"&"`. Cada campo é representado por um par `atributo=valor`, conforme mostrado na linha abaixo.

```
prod=Martelo+M5A&cod=32456240&price=%2430.00&btn_sub=Submit
```

Para extrair os dados do formulário, precisamos de um analisador léxico. Da mesma forma que programação funcional, construção de compiladores é estudada em qualquer universidade do mundo. Então, torna-se desnecessário explicar os métodos usados no analisador léxico.

O protocolo CGI utiliza somente sete bits. Isto significa que certas letras do português não podem ser transmitidas. Em vez disso, o sistema transmite códigos de sete bits para representá-las. No servidor, o analisador léxico encarrega-se de restaurar as letras codificadas, conforme mostrado abaixo.

```
    tk ['%C3%', a, b:xs]
        # (w, r)= tk xs;
        = ([letra a b:w], r);

    ...

    letra 'A' '0' = 'à';
    letra 'A' '1' = 'á';
    ...etc.
```

```
definition module CGIparser;
tok :: String -> [String];

implementation module CGIparser;
import StdEnv;

tok :: String -> [String];
tok s= map toString (wds [x\\ x <-: s]);
where{ seps= ['&='];
  tk []= ([], []);
  tk [x:xs] | isMember x seps= ([], [x:xs]);
  tk ['+':xs]
    # (w, r)= tk xs;
    = ([' ':w], r);
  tk ['%C3%', a, b:xs]
    # (w, r)= tk xs;
    = ([letra a b:w], r);
  tk ['%24':xs]
    # (w, r)= tk xs;
    = (['$':w], r);
  tk [x:xs]
    # (w, r)= tk xs;
    = ([x:w], r);
  letra 'B' 'A' = 'ú';
  letra 'B' 'C' = 'ü';
  letra 'A' '7' = 'ç';
  letra 'A' '9' = 'é';
  letra 'A' 'D' = 'í';
  letra 'A' a = ['àáâã']!!(min (toInt (a-'0')) 3);
  letra 'B' '3' = 'ó';
```

```
  letra 'B' '4' = 'ô';
  letra a b= '*';
  wds []= [];
  wds [x:xs] | isMember x seps= [[x]:wds xs];
  wds [x:xs]
    # (n, r)= tk  [x:xs]
    = [n: wds r];}

module cleangi;
import StdEnv, ArgEnv, CGIparser;

Start w
  # (cgi, w)= stdio w;
  # ctype= "Content-Type:text/xml;charset=iso-8859-1";
  # cgi= fwrites ctype cgi;
  # cgi= fwrites ({toChar 13, toChar 10}+++"\n") cgi;
  # cgi= fwrites (bicudo (version "1.0")) cgi
  # cgi= fwrites (stylesheet "http://localhost/tag-val.xsl") cgi;
  # xx= ans (getEnvironmentVariable "QUERY_STRING");
  # cgi= fwrites "\n<attribs title=\"Produtos\">\n" cgi;
  # cgi= prt (tok xx) cgi;
  # cgi= fwrites "\n</attribs>\n" cgi;
= cgi;
where { ans EnvironmentVariableUndefined="";
        ans (EnvironmentVariable v)=v;}

// Better safe than sorry: Leave a space
// before and after each string.
bicudo xx = "<?xml "+++xx+++" ?>\n";
stylesheet href=
  "<?xml-stylesheet "+++
  "type=\"text/xml\" href=\""+++
      href +++"\" ?>\n";
version v= " version=\""+++v+++"\" "+++Latin1;
Latin1 :== "  encoding=\"ISO-8859-1\" ";

prt [] console= console;
prt ["&":xs] console = prt xs (fwrites "\n" console);
prt ["+":xs] console = prt xs (fwrites " " console);
prt [nm, "=", v:xs] console=
   prt xs (fwrites ("<att "+++"tag=\""+++nm+++"\"  "+++
```

```
            "val=\""+++v+++"\" />")
               console);
prt [x:xs] console= prt xs (fwrites x console);
```

O programa em Clean deve ser compilado com a opção `No Return Type`. Na janela principal do Clean, escolha a entrada `Project` do menu principal e vá para o submenu `Project Options`. Isto abrirá um diálogo de opções de projeto, onde você poderá marcar a escolha `No Return Type`. Após a compilação, transfira o programa `cleangi.exe` para o diretório `cgi-bin`.

Ao receber uma *string* no protocolo CGI, Clean gera, através da função `prt`, uma estrutura XML com o seguinte formato:

```
Content-Type:text/xml;charset=iso-8859-1

<?xml  version="1.0"   encoding="ISO-8859-1"  ?>
<?xml-stylesheet type="text/xml"
   href="http://localhost/tag-val.xsl" ?>

<attribs title="Produtos">
<att tag="prod"  val="Rosa" />
<att tag="price"  val="$20,00" />
</attribs>
```

Esta estrutura funciona para qualquer formulário: Você não precisa de escrever outro programa em Clean para cada página que processar. Para mudar a apresentação dos resultados, basta trocar o programa em `tag-val.xsl`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="attribs">
<html><body>
  <H2> <xsl:value-of select="@title"/>  </H2>
```

```
  <xsl:apply-templates/>
</body></html>
</xsl:template>

<xsl:template match="att">
  <xsl:apply-templates select="@tag"/>
  <xsl:text> = </xsl:text>
  <xsl:apply-templates select="@val"/> <br />
</xsl:template>

</xsl:stylesheet>
```

O programa `tag-val.xsl`, assim como os arquivos `xml-tools.xml` e `tools.xsl`, fornecidos abaixo, devem ir para o diretório `htdocs` do servidor Apache. Para testar os programas, você deve colocá-los nos locais indicados, entrar no *browser*, e digitar

```
http://localhost/xml-tool.xml
```

Aqui está o arquivo `xml-tools.xml`, que descreve uma ferramenta.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="tool.xsl"?>

<tool>
  <field id="prod">
    <value>Martelo M5A</value>
  </field>
  <field id="cod">
    <value>32456240</value>
  </field>
  <field id="price">
    <value>$30.00</value>
  </field>
</tool>
```

Ferramenta

prod Martelo M5A

cod 32456240

price $30.00

Submit Reset

O programa `tool.xsl` é dado abaixo.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html><body>
<form method="get" ACTION="http://localhost/cgi-bin/cleangi.exe">
<h2>Ferramenta</h2>
<table border="0">
<xsl:for-each select="tool/field">
<tr>
<td><xsl:value-of select="@id"/></td>
<td><input type="text">
<xsl:attribute name="id">
  <xsl:value-of select="@id" />
</xsl:attribute>
<xsl:attribute name="name">
  <xsl:value-of select="@id" />
</xsl:attribute>
<xsl:attribute name="value">
  <xsl:value-of select="value" />
</xsl:attribute>
</input> </td>
</tr>
</xsl:for-each>
</table>
<br />
<input type="submit" id="btn_sub" name="btn_sub" value="Submit" />
<input type="reset" id="btn_res" name="btn_res" value="Reset" />
</form>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

5.3 Carregando documentos

Um programa em XSLT pode carregar um ou mais documentos em formato XML, conforme mostra o exemplo que segue.

Livros Mais Vendidos

Título	Autor
Os Três Mosqueteiros	Alexandre Dumas
A Dama das Camélias	Alexandre Dumas Filho
Odisséia	Homero

Autores suecos

Astrid Lindgren
Selma Lagerlöf

Autores não-suecos

H.C. Andersen

Arquivo `if-book.xsl`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

  <html><body>
    <h2>Livros Mais Vendidos</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Título</th> <th>Autor</th> </tr>
      <xsl:for-each select="livros/bk">
        <xsl:if test="sold &gt; 500">
        <tr> <td><xsl:value-of select="title"/></td>
             <td><xsl:value-of select="author"/></td> </tr>
      </xsl:if> </xsl:for-each>
    </table>
```

```
  <p> <xsl:variable name="data"
         select="document('my-authors.xml')"/>

      <h2>Autores suecos</h2>
      <xsl:for-each
         select="$data/authors/author[@country = 'Sweden']">
          <xsl:value-of select="@name"/><br/>
      </xsl:for-each>

      <h2>Autores não-suecos</h2>
      <xsl:for-each
        select="$data/authors/author[@country != 'Sweden']">
           <xsl:value-of select="@name"/><br/>
      </xsl:for-each>
  </p>
  </body></html>
</xsl:template>
</xsl:stylesheet>
```

Arquivo `xml-if-book.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="if-books.xsl"?>
<livros>
  <bk>
    <title>Os Três Mosqueteiros</title>
    <author>Alexandre Dumas</author>
    <price>200</price>
    <sold>2000</sold>
  </bk>
  <bk>
    <title>A Dama das Camélias</title>
    <author>Alexandre Dumas Filho</author>
    <price>200</price>
    <sold>600</sold>
  </bk>
  <bk>
    <title>Odisséia</title>
    <author>Homero</author>
    <price>150</price> <sold>800</sold>
```

```
  </bk>
   <bk>
      <title>O Vermelho e o Negro</title>
      <author>Stendhal</author>
      <price>38</price>
      <sold>5</sold>
  </bk>
  <bk>
    <title>Poesias</title>
    <author>Olavo Bilac</author>
    <price>30</price>
    <sold>8</sold>
  </bk>
</livros>
```

Arquivo `my-authors.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<authors>
  <author name="Astrid Lindgren" country="Sweden"/>
  <author name="H.C. Andersen" country="Denmark"/>
  <author name="Selma Lagerlöf" country="Sweden"/>
</authors>
```

5.4 DSSSL e Scheme

Se você souber Scheme, programar para a internete ficará muito mais fácil e seguro. A linguagem DSSSL, que serve de base para XSLT, é uma forma de LISP e está presente em todas as implementações de Scheme. Veja o que diz o manual da Bigloo, que é uma poderosa implementação de Scheme:

Bigloo supports extensions for the DSSSL expression language.

Além de estar na base do XSLT, Scheme possui ferramentas poderosas, que facilitam a construção de bases de dados, a análise léxico-sintática de docu-

mentos e a utilização de bibliotecas existentes em outras linguagens (como C, Java, dotNet, etc.).

Uma linguagem rigorosa como Clean, garante programas corretos, seguros e rápidos. Infelizmente, poucas pessoas possuem a disciplina mental para programar em Clean, Lisaac ou Eiffel.

Linguagens interpretadas e sem tipos, como Python, Javascript, PHP e Prolog, são fáceis de aprender e permitem um desenvolvimento rápido e indisciplinado de aplicações. Mas tudo tem seu preço! Programas escritos nestas linguagens são lentos, cheios de erros e perigosos. Um programa em Python, por exemplo, chega a ser de 100 a 200 vezes mais lento do que o equivalente em Lisp (ver [Norvig]). Além disso, erros insidiosos estão sempre derrubando o programa Python. Seus clientes perceberão que você está trabalhando com PHP ou Python, pois a aplicação não funcionará bem. Quando entrar em uma página da internete e o serviço estiver muito lento, surgir mensagens de erro na tela de seu computador, dê uma olhada nas mensagens de erro e verifique a linguagem em que o programa foi escrito: Python, PHP ou Javascript.

Scheme é uma linguagem sem tipos, interpretada e muito amigável. Será que, por isso, ela é lenta? De fato ela tem tudo para ser lenta. A ausência de tipos impede que importantes otimizações sejam feitas no código compilado. Estruturas de dados precisam ser muito gerais e dimensionadas dinamicamente, o que provoca perda de eficiência. Além disso, a falta de tipos e disciplina leva a erros de programação, tal como no Python. Aliás, os adeptos do Python dizem que sua linguagem é inspirada no Scheme!

Scheme, entretanto, tem uma coisa que as outras linguagens sem tipo não possuem: Ela tem uma estrutura sintática perfeita, a melhor estrutura sintática que se poderia imaginar. O grande lógico e matemático polonês Jan Łukasiewicz inventou a notação prefixa utilizada no Scheme e demonstrou que ela é a mais concisa, menos ambígua e mais flexível notação lógica que existe. Em consequência disso, os construtores de compilador encantaram-se com Scheme e colocaram todo seu talento para construir compiladores rápidos e seguros. Inventaram a verificação macia de tipos, que permite localizar erros nos programas, sem perder a generalidade das estruturas de dados. Além disso, desenvolveram métodos de capturar a continuação de programas, neutralizando as consequências de erros. Assim, Scheme tornou-se a linguagem mais rápida, flexível e segura que existe.

Uma pergunta que nos vem à mente é: Seria possível usar as técnicas do Scheme para aumentar a eficiência do Python ou do Java? Não se pode dizer ao certo. Entretanto, a Sun contratou especialistas em Scheme para construir os compiladores Java, aumentando a eficiência da linguagem. Talvez coisa parecida pudesse ser feita para Python. Aumentar a eficiência do Python ou do PHP, contudo, esbarra em dois problemas.

- Estas linguagens não possuem a sintaxe amigável e rigorosa do Scheme, onde programas e dados têm a mesma forma. Isto dificulta a manipulação de programas para neutralizar erros e aumentar a eficiência.

- Não tendo a elegância e beleza formal do Scheme, Python e PHP não atraem grandes construtores de compilador, como Marc Feeley, Manuel Serrano ou Jeffrey Siskind.

Das implementações de Scheme, a mais rápida chama-se Stalin, e foi desenvolvida por um cientista de computação judeu chamado Jeffrey Mark Siskind. Stalin chega a superar compiladores C em várias vezes.

A implementação de Scheme mais poderosa é a Bigloo. Bigloo trabalha de maneira transparente com C, Java e dotNet. Consegue usar, sem dificuldades bibliotecas escritas em qualquer linguagem. Embora não seja tão rápida como a Stalin, a Bigloo é muito rápida, empatando com C facilmente. Neste material, usaremos Bigloo para aproveitar as poderosas ferramentas que esta implementação oferece.

Como sempre, vamos supor que o leitor já sabe Scheme e escrever um pequeno program para receber e processar um formulário.

```
(module cgitest)

(print "Content-Type:text/html;charset=iso-8859-1\n

           <html><body>
               <H3>Internete Super Simples</H3>")
                 (for-each (lambda(c) (print c " <br />"))
                   (string-split (getenv "QUERY_STRING") "&"))
(print "   </body></html>")
```

Como você pode ver, é realmente simples programar para a internete em Scheme. O programa acima não tem dez linhas. Bastou imprimir o cabeçalho da página HTML, obter os dados e processá-los. O processamento foi fácil:

- `(string-split (getenv "QUERY_STRING") "&")`: A função `getenv` recebe o formulário em formato texto e `string-split` quebra-o em uma lista de pares `atributo=valor`.

- `for-each` imprime cada um dos pares `atributo=valor`.

Note que, em nove linhas, conseguimos fazer um programa com praticamente o mesmo comportamento do programa em Clean, que tem quase cem linhas!

Podemos criar uma lista com listas do tipo `(atributo, valor)` com todas as variáveis enviadas pelo navegador usando formulários html no modo GET usando o seguinte trecho de código:

```
(let ((query (getenv "QUERY_STRING")))
    (map (lambda (x) (pregexp-split "=" x))
         (pregexp-split "&" query)))
```

O segredo de Bigloo são as poderosas funções de biblioteca que fazem tratamento de *strings*. Linguagens como Perl também possuem tais funções. Assim, meu programa de nove linhas talvez impressione um programador Perl pela velocidade e por não exigir a presença de um intérprete para ser executado, mas não impressionará pela concisão. Acontece que, em certas situações, as funções de biblioteca não dão conta do recado. Por exemplo, se quisermos que o programa acerte o texto para torná-lo adequado às características do português ou que corrija os erros de ortografia e sintaxe. Neste caso, Perl falhará miseravelmente, e Clean brilhará. E Scheme? Conseguiria exibir um alto quociente de inteligência artificial em comparação com Clean? Para tirar suas próprias conclusões, compile o arquivo `cgischeme.scm` e transfira-o para o diretório `cgi-bin`.

```
(module cgitest )

(print
"Content-Type:text/xml;charset=iso-8859-1

<?xml version=\"1.0\" encoding=\"ISO-8859-1\" ?>\n
<?xml-stylesheet type=\"text/xml\"
               href=\"http://localhost/tag-val.xsl\" ?>\n
```

```
          <attribs title=\"Tratamento de acentos\">" )
(for-each elem (string-split (getenv "QUERY_STRING" ) "&") )
(print "</attribs>")

(define (elem c)
   (match-case (string-split c "=")
      ( (?a ?b) (display* "  <att tag=\""  a "\" val=\"")
                (acentua b) (display "\" />") )
      ( ?- "")))

(define (acentua x)
   (string-case x
      ( #\+ (display " ") (ignore))
      ( "%C3" (ignore))
      ( "%24" (display "$") (ignore))
      ( "%A0" (display "à") (ignore))
      ( "%A1" (display "á") (ignore))
      ( "%A2" (display "â") (ignore))
      ( "%A9" (display "é") (ignore))
      ( "%A7" (display "ç") (ignore))
      ( "%A3" (display "ã") (ignore))
      ( "%AD" (display "í") (ignore))
      ( "%B3" (display "ó") (ignore))
      ( "%BA" (display "ú") (ignore))
      ( "%BC" (display "ü") (ignore))
      ( "%B4" (display "ô") (ignore))
      ( (out #\newline) (display (the-string)) (ignore))
      (else (display ""))))
```

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">

<xsl:template match="/">
<html><body>
  <H2>Tabela de fatoriais</H2>
     <xsl:for-each select="//number">
     <b><xsl:text> Fatorial de  </xsl:text> </b>
    <xsl:value-of select="."/>
    <xsl:text> = </xsl:text>
    <xsl:call-template name="factorial">
       <xsl:with-param name="number" select="." />
    </xsl:call-template>
   <br />
  </xsl:for-each>
</body></html>
</xsl:template>

<xsl:template name="factorial">
  <xsl:param name="number" />

  <xsl:choose>
    <xsl:when test="$number = 0">
      1
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="f1">
        <xsl:call-template name="factorial">
           <xsl:with-param name="number" select="$number - 1" />
        </xsl:call-template>
      </xsl:variable>
      <xsl:value-of select="$number * $f1" />
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>

</xsl:stylesheet>
```

Figura 5.2: fat.xsl

Capítulo 6

Documentos e-book

Um livro eletrônico ou (*e-book* é uma publicação com textos e imagens no formato digital, que é produzida, publicada e visualizada em dispositivos digitais e computadores. Muitas vezes é equivalente de um livro impresso, mas também pode não ter um par. *E-book*s podem ser lidos em dispositivos de hardware dedicados a essa tarefa, chamados e-Readers. Os computadores pessoais e até alguns telefones celulares podem também serem usados para ler *e-book*s.

Existem vários formatos de livros eletrônicos, alguns deles são o pdf (Portable Document Format), o prc (ou *mobipocket*) e o epub.

O pdf foi desenvolvido pela empresa Adobe Systems em 1993, o intuito era representar documentos de maneira portável, de forma que independente do sistema operacional usado para criar o arquivo pdf ele pudesse ser lido em outros sistemas operacionais. Esse tipo de arquivo pode conter texto, imagens e gráficos.

O prc ou o format de *e-book* Mobipocket é baseado no padrão Open eBook, utiliza arquivos XHTML como conteúdo e pode incluir JavaScript e frames. Ele também suporta requisições SQL nativas a serem usadas com banco de dados embarcados.

O format epub ou OEBPS, é um padrão aberto para e-books criado por *International Digital Publishing Forum* (IDPF). Ele é a combinação de três padrões abertos da IDPF:

- OPS (Open Publication Structure) 2.0, descreve o conteúdo de marcação (sendo XHTML ou Daisy DTBook).

- OPF (Open Packaging Format) 2.0, descreve a estrutura em XML de um epub.
- OCF (OEBPS Container Format) 1.0, que combina todos os arquivos em um só .epub.

Somente o format epub será trabalhado neste documento, ele é um padrão aberto e pode ser lido pela maioria dos dispositivos de leitura de livros eletrônicos.

6.1 Conteúdo epub

O conteúdo do livro eletrônico é criado como um arquivo comum XHTML, o raiz de todos os documentos OPS devem especificar o seguinte namespace:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ops=" http://www.idpf.org/2007/ops">
```

A maioria das tags XHTML para marcação de conteúdo são aceitas como descritas nas especificações XHTML. As tags de script não devem ser executadas pelos dispositivos de leitura, então é útil incluir apenas as tags que fazem sentido em uma estrutura de livro.

É possível criar ligações entre diversos OPS utilizando a tag ¡a¿ como seria usada em uma página web. Outros elementos como imagens, tabelas e folhas de estilo também podem ser incluídos.

Os dispositivos de leitura epub devem suportar os módulos apresentados na tabela abaixo, conforme suas descrições nas especificações XHTML.

6.2 Formato do pacote epub

O formato do pacote epub é um arquivo XML que processado pelo leitor de livros eletrônicos indica o caminho dos arquivos de conteúdo e mídia do livro.

Nesse arquivo XML a raiz deve ser a seguinte:

```
<package version="2.0"
         xmlns="http://www.idpf.org/2007/opf">
             ...
</package>
```

O pacote epub deve ter um identificador único (*unique-identifier*) como um atributo da tag package. As partes que compoem o pacote são os metadados da publicação (titúlo, autor, editora, etc.), um manifesto, que é uma

Tabela 6.1: Módulos XHTML

Módulo XHTML	Elementos
Structure	body, head, html, title
Text	abbr, acronym, address, blockquote, br, cite, code, dfn, div, em, h1, h2, h3, h4, h5, h6, kbd, p, pre, q, samp, span, strong, var
Hypertext	a
List	dl, dt, dd, ol, ul, li
Object	object, param
Presentation	b, big, hr, i, small, sub, sup, tt
Edit	del, ins
Bidirectional Text	bdo
Table	caption, col, colgroup, table, tbody, td, tfoot, th, thead, tr
Image	img
Client-Side Image Map	area, map
Meta-Information	meta
Style Sheet	style
Client-Side Image Map	area, map
Link	link
Base	base

lista de arquivos que constituem a publicação e também indica arquivos a serem usados se os primeiros não puderem ser trabalhados pelo dispositivo de leitura. Uma espinha que determina a ordem de leitura dos documentos. E por último um guia, que é um conjunto de referências das características estruturais fundamentais da publicação como índice, bibliografia, etc.

6.3 Formato de contêiner epub

A publicação epub é colocada em um contêiner que nada mais é que um diretório da publicação com arquivos texto e mídia, arquivo XML do formato do pacote e outros arquivos da publicação compactados em formato zip com o primeiro arquivo sendo um arquivo texto de nome mimetype contendo a linha abaixo:

```
application/epub+zip
```

O sistema de diretórios dentro do arquivo contêiner deve seguir a mesma estrutura indicada pelo arquivo de formato do pacote.

6.4 epub no e-Reader

Depois de compactado o contêiner no sistema zip ele deve ser renomeado trocando-se a extensão zip por epub e inserido no dispositivo de leitura. O dispositivo de leitura abre o arquivo mimetype e identifica o tipo de livro eletrônico definido, dessa forma o conteúdo é apresentado ao usuário de acordo com o que foi definido no arquivo de formato epub.

Capítulo 7

Conclusão e trabalhos futuros

7.1 Conclusão

Apresentamos nesta dissertação alguns dos problemas que são gerados pela tentativa de fazer uso de ferramentas gráficas como o UML para a programação de sistemas de computador. O objetivo principal é o de mostrar que existe um caminho melhor a ser percorrido por quem deseja programar eficientemente e mais próximo do problema, construindo ferramentas que vão passo a passo se aproximando da solução da questão.

Empregando-se a programação exploratória utilizando uma ferramenta poderosa como a LISP é possível se libertar das ferramentas inadequadas que continuam a ser expostas nas universidades como a solução para a Engenharia de Software quando na realidade atrasam a conclusão do serviço e não adicionam qualquer benefício a não ser o dá comunicação do gerente de projeto com o cliente.

Mais tarde expomos os documentos empregados na internet, como a visualização dos dados gerados no servidor pode ser tratada utilizando-se linguagens poderosas como a XSLT. A XSLT pode ser empregada na arquitetura de software MVC (*Model-View-Controller*) de modo a ser usada como a visão (*view*) apresentando o modelo em um formato adequado ao usuário, assim um único conjunto de dados gerados pelo controlador (*controller*) pode ser apresentado de diferentes formas utilizando-se documentos XSLT diferentes de acordo com os mais variados propósitos.

Finalmente os documentos utilizados como livros eletrônicos em leitores eletrônicos foram investigados, a nova realidade da transmissão de conhecimento foi abordada. A publicação eletrônica mudou, essa nova forma de exposição deve prevalecer sobre a antiga forma de publicação em papel agora

que as livrarias vendem mais livros nesse formato do que os tradicionais impressos. Dessa forma é importante que o profissional da computação esteja pronto para a produção de novos produtos direcionados para esse novo nicho de mercado, o *e-book*.

7.2 Trabalhos futuros

Dando continuação à publicação que foi gerada junto a essa dissertação, devemos produzir software capaz de navegar por páginas da rede mundial de computadores, colher informações úteis e gerar documentos que possam ser exibidos nos diversos dispositivos leitores de livros eletrônicos. Empregando a análise da semântica das publicações online deve-se extrair automaticamente de documentos HTML o que um usuário final procuraria manualmente.

Esses sistemas também devem colher informações e imagens que estão relacionadas ao conteúdo extraído e incluí-las de forma a enriquecer a publicação eletrônica ainda mais. Devemos empregar ferramentas de programação exploratória utilizando-nos de dialetos da LISP que interajam bem com os mais diversos sistemas operacionais existentes visando a portabilidade do sistema ou ainda contruir o sistema baseado em um servidor web permitindo a utilização do software através dos mais diversos navegadores.

Bibliografia

[Lispy] **WHY UML Fails to Add Value to the Design and Development Process**.
Disponível em: <http://lispy.wordpress.com>.
Acesso em: 29 ago. 2009.

[Dzidek et al.] WJ Dzidek, E Arisholm, LC Briand. **A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance**, IEEE Transactions on Software Engineering, Vol 34 No 3, May/June 2008

[Kelly] KELLY, Steven. **Using UML takes 15% longer than just coding**.
Disponível em:
<http://www.metacase.com/blogs/stevek/blogView?showComments=true&entry=3432385251>.
Acesso em: 14 nov. 2010.

[Spolsky] SPOLSKY, Joel. **The Perils of JavaSchools**.
Disponível em: <http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html>.
Acesso em: 22 nov. 2010.

[Gat] Erann Gat. **LISP as an alternative to Java**, Novembro 1999 Communications of the ACM.
Disponível em:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.532&rep=rep1&type=pdf
Acesso em: 11 ago. 2010.

[Avg] Paul Graham. **Beating the averages**.
Disponível em:
http://www.paulgraham.com/avg.html Acesso em: 7 out. 2010.

[Schmandt-Besserat] SCHMANDT-BESSERAT, Denise. **Tokens and Writing: the Cognitive Development**. Proceedings Of The Scripta 2008, Seoul, n. , p.8-12, out. 2008.

[Amelsvoort] AMELSVOORT, Marije Van; BREUGELMANS, Seger. **Boundary conditions for applying argumentative diagrams**. Cscl, Tilburg, n. , p.724-726, 2007.

[John Hughes] HUGHES, John. **Why functional programming matters**. The Computer Journal, Glasgow, n. , p.17-42, 1997.

[Karl Popper] POPPER, Karl Raimund. **A Lógica Da Pesquisa Cientifica**. São Paulo: Cultrix, 2000.

[onlisp] GRAHAM, Paul. **On LISP: Advanced Techniques for Common LISP**. Prentice Hall, 1993.

[Hoare] HOARE, Charles Antony Richard. **An axiomatic basis for computer programming**. Communications Of The Acm, p.576-583, out. 1969.

[acl1] GRAHAM, Paul. **ANSI Common Lisp**. Prentice Hall, 1995.

[opencart] **OpenSource Shopping Cart Solution**. Disponível em: <http://www.opencart.com/index.php?route=feature/feature> Acesso em: 24 nov. 2010.

[Norvig] NORVIG, Peter. **Python for Lisp Programmers**. Disponível em: <http://www.norvig.com/python-lisp.html>. Acesso em: 11 out. 2009.

[Lnotes] **LISP50 Notes part VI: The Future of Lisp**. Disponível em: <http://lispy.wordpress.com/2008/10/25/lisp50-notes-part-vi-the-future-of-lisp/>. Acesso em: 11 jun. 2010.