UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CURSO DE PÓS-GRADUÇÃO EM CÎENCIA DA COMPUTAÇÃO

Programação Funcional Usando Java

por

JORGE JUAN ZAVALETA GAVIDIA

Dissertação submetida à avaliação, como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Prof. Dr. Paulo Alberto de Azeredo
Orientador

Porto Alegre, julho de 1997

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Zavaleta Gavidia, Jorge Juan

Programação funcional usando Java / por Jorge Juan Zavaleta Gavidia.- Porto Alegre: CPGCC da UFRGS, 1997.

145f. : il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-graduação em Ciência da Computação, Porto alegre, BR - RS, 1997. Orientador: Azeredo, Paulo Alberto.

1. Java. 2. Orientação a Objetos. 3. Linguagem funcional. 4. Interpretador Lisp. 5. Construtor de funções Java. I. Azeredo, Paulo Alberto. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Profa. Wrana Panizzi
Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenador do CPGCC: Prof. Flavio Wagner
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Ao longo do processo de levantamento de informações, pesquisas e compilação de idéias que nortearam este trabalho, muitas amizades foram criadas e muitos contatos foram feitos, a estas pessoas meu muito obrigado pela colaboração.

Àqueles que auxiliaram na minha lapidação e desespero, afirmo que o caminho foi árduo e penoso, mas recompensador.

Aos grandes colegas e amigos do CPGCC/UFRGS pela excelente amizade consolidada neste período.

Aos funcionários da biblioteca, dos laboratórios e da secretaria pela atenção dispensada.

Aos colegas da Universidad Nacional de Trujillo, da escola de Matemáticas no Peru, pelo incentivo dado para fazer Mestrado no II da UFRGS.

Ao corpo docente do Curso de Pós-Graduação em Ciência da Computação do Instituto de Informática da Universidade Federal do Rio Grande do Sul (UFRGS), em especial a meu orientador, pelo apoio e pelas sugestões que muito me auxiliaram na elaboração deste trabalho. Agradeço também ao CNPq pela bolsa de estudos concedida ao longo do Curso.

Em especial dedico este trabalho a meus pais *Andrés* e *Alejandrina* pelos ensinamentos, dedicação incansável e incentivo constante durante toda minha vida. Aos meus irmãos, *Guillermo, Víctor, Oswaldo, Rosa, Ivan, César e Andrés Jr.*, e a minha querida sobrinha *Monique Maribelle* pelo constante apoio moral dado desde o Peru, através do telefone. A todos eles muito obrigado.

Jorge Juan

Sumário

Lista de Abreviaturas

API	Interface de Programas de Aplicação
BNF	Backus Normal Form
GUI	Interface de Uso Gráfico
HTML	HyperText Markup Language
IA	Inteligência Artificial
JDK	Java Developers Kit
JVM	Máquina Virtual Java
LAN	Redes de Área Local
LPF	Linguagem de Programação Funcional
LsPF	Linguagens de Programação Funcional
MTU	Máquina de Turing Universal
MTU's	Maquinas de Turing Universais
PC	Computador Pessoal
PFOO	Programação Funcional Orientada a Objetos
POO	Programação Orientada a Objetos
TCP/IP	Transmission Control Protocol/Internet Protocol
TIA	Teoria de Informação Algorítmica
UFRGS	Universidade Federal do Rio Grande do Sul
URL	Uniform Resource Locator
WWW	World Wide Web

Lista de Figuras

Lista de Tabelas

Resumo

Desde a introdução da World Wide Web para o mundo nos inícios de 1990, usando a Internet como uma rede para transferir dados, empregando uma forma de expressão chamada de Hipertexto, a qual liga as informações relacionadas e combinadas com multimídia, os Webs resultantes têm aberto novas possibilidades de expressão e comunicação. A quantidade de tráfego de dados na Web e o número de computadores ofertando informação vem crescendo dramaticamente, mas falta expressividade e qualidade interativa na Web; ainda assim, vem despertando um grande interesse instrutivo e útil.

O ilimitado universo de possibilidades da Web para acessar aplicações seguras, portáveis e independentes para cada plataforma em hardware e software e que possam chegar a qualquer lugar sobre a Internet, surge a linguagem Java da Sun Microsystem [DEC 95]. A habilidade de Java para executar código sobre hosts remotos de uma maneira segura é uma necessidade crítica para muitas organizações de desenvolvedores de software e provedores de Internet na atualidade [ARN 96].

A linguagem Java é realmente valiosa para redes de ambientes distribuídos como a Web. Entretanto, Java vai mais longe deste domínio ao fornecer uma linguagem de programação de propósito geral poderosa e adequada para construir uma variedade de aplicações que não dependem das características da rede [ARN 96].

O modelo imperativo tradicional é padrão e quase universal vem tendo uma profunda influência sobre a natureza das linguagens de programação e ainda continua a tendência de sempre ter uma direção para fornecer mais e mais formas abstratas de resolver problemas, tentando mudar a simplicidade na programação com rapidez na execução de programas [FIE 88].

Parece, portanto, natural e quase inevitável o desenvolvimento em tecnologia das linguagens. Os amplos esforços gastos em desenvolver métodos rigorosos para especificar, produzir, verificar software e produtos de hardware, mas os esforços foram restringidos às linguagens convencionais. A aproximação natural de Von Neumann tem contribuído a esta falha desde a noção de um estado global que pode mudar arbitrariamente em cada passo da computação e vem sendo provado ser intuitivamente e matematicamente intratável. Esta falha tem tornado ao software o componente mais caro para muitos sistemas de computação [GLA 84].

Os primeiros passos para solucionar estas falhas foram tomadas pela programação estruturada ao trabalhar nas áreas de especificação formal, verificação de programas e na semântica formal que ainda continuam em pesquisa. O crescimento numeroso de pesquisadores têm certeza de que os problemas originam-se da aproximação fundamental à filosofia de Von Neumann e estão voltando-se para uma linguagem de um novo tipo. Uma de tais aproximações é a tomada pelas linguagens de programação funcional [PLA 93].

Num programa funcional, o resultado de uma função chamada é unicamente determinado pelos valores atuais dos argumentos da função [PLA 93]. As linguagens de

programação funcional têm a vantagem que elas oferecem um uso geral das funções, o qual não esta disponível nas linguagens imperativas clássicas. Devido a ausência de efeitos colaterais, as provas de correção dos programas são mais fácies que nas linguagens imperativas. As funções podem ser avaliadas em qualquer ordem assim como a disponibilidade total das mesmas, a nova geração de linguagens funcionais também oferecem uma elegante noção de uso amigável [PLA 93]. Os padrões e a proteção que fornecem ao usuário um acesso simples a estruturas de dados complexos, basicamente não tendo a preocupação do gerenciamento da memória, como faz a linguagem Java.

O objetivo principal deste trabalho é a descrição da implementação de um Construtor de Funções Java (*LispJ*), usado para gerar funções Lisp em código Java utilizando a linguagem Java da Sun Microsystem como ambiente de desenvolvimento. A descrição compreende a codificação de um Interpretador Lisp da linguagem funcional LISP, e a codificação do Construtor de Funções Java visualizado através de um applet Java utilizado como interface entre o Construtor de Funções Java e o usuário sobre a Internet.

PALAVRAS-CHAVE: Java, orientação a objetos, linguagem funcional, interpretador Lisp, construtor de funções Java.

TITLE: "FUNCTIONAL PROGRAMMING USING JAVA"

Abstract

Since the introduction of the World Wide Web to the world in the beginning of the nineties, using the Internet as a network to transfer data, using a form of expression called Hypertext, which connects related and combined information with multimedia, the resulting Webs have opened new possibilities of expression and communication. The amount of data traffic in the Web and the number of computers offering information have been growing dramatically, but there is a lack of interactive expressivity and quality in the Web; nevertheless, its instructive and useful interest is growing wider.

From the unlimited universe of possibilities of the Web to access safe, portable and independent applications for each platform in hardware and software and that are able to get anywhere on the Internet, there is the Java Sun Mycrosystem language [DEC 95]. Java's ability to perform code on remote hosts in a safe way is a critical need for many software developing organizations and Internet providers nowadays [ARN 96].

Java language is really valuable for network environments arranged as the Web. However, Java extends further from this domain as it provides a broad programming language that is powerful and adequate to build a variety of applications which do not depend on the characteristics of the network [ARN 96].

The prevailing traditional model is a pattern and almost universal, has had a deep influence on the nature of the programming languages and there is still a trend of one direction to provide more and more abstract ways of solving problems, trying to change the simplicity in the fast programming in programs run [FIE 88].

It seems, therefore, natural and almost inevitable the development in technology of the programming languages. Wide efforts were made to develop strict methods to specify, produce, check software and hardware products, but the efforts were restricted to conventional languages. Von Neumann's natural approximation has contributed to this gap since the notion of a global state which can change arbitrarily in each step of the computer science and has proven to be intuitively and mathematically intractable. This gap has turned the software into the most expensive component for many computing systems [GLA 84].

The first steps to solve these gaps were taken by the structured programming when working on the areas of formal specification, programs checking and the formal semantics, which are still being researched. The ever growing number of researchers are sure that the problems come from the fundamental approximation to Von Neumann's philosophy and are turning to a new kind of language. One of such approximations is the one through the functional programming languages [PLA 93].

In a functional program, the result of a called function is determined only by the present values of the function arguments [PLA 93]. The functional programming languages have the advantage of offering a general use of the functions, which is not

available in the classic prevailling languages [PLA 93]. Due to absence of side effects, the correction tests in the programs are easier than in the prevailing languages. The functions may be evaluated in any order and so may their total disposal. The new generation of functional languages also offers an elegant notion of friendly use [PLA 93]. The patterns and protection offer the user a simple access to complex data structures, basically by not worrying about memory management, as occurs with the Java language.

The main objetive of this work is the description of the implementation of a Java Functions Builder (*LispJ*), used do generate Lisp functions in Java code utilizing the Java language from Sun microsystem as a developing environment. The description covers the code of the Lisp Interpreter of the LISP functional language, and the Java Functions Builder code visualized through a Java applet utilized as interface between the Java Functions Builder and the users on the Internet.

KEYWORDS: Java; orientation to objects; functional language; Lisp interpreter; builder of Java functions.

1 Introdução

Pessoas e organizações usam a World Wide Web para comunicarem-se globalmente e imediatamente. Usando a Internet como uma rede para transferir dados, a Web emprega uma forma de expressão chamada Hipertexto que liga informações relacionadas. Combinado com multimídia, os webs resultantes de hipermídia têm aberto novas possibilidades para expressão e comunicação [DEC 95].

Mas, algumas coisas desapareceram da Web, desde sua introdução para o mundo nos inícios de 1990. Embora, a quantidade de tráfego de dados na Web e o número de computadores ofertando informação sobre a Web tem crescido dramaticamente, falta conteúdo importante de expressividade e qualidade interativa nas páginas. Apesar de que freqüentemente vem despertando o interesse instrutivo e útil, a Web foi desprovida gradualmente de oferecer interatividade para muitos sistemas de multimídia e hipermídia, que rodam sobre computadores fora da Internet. Assim, apesar da Web ter estimulado as interconexões entre pessoas e informação, ela tem capacidade somente para ser observada pelas pessoas: ler texto, assistir vídeos, ouvir música, e pesquisar informação [DEC 95].

O ilimitado universo de possibilidades sobre a Web podem conduzir aos usuários para sentir que os hiperlinks justamente ficam de conduzi-lo, até que ultimamente, a Web parece uma estrada para algum lugar. Java muda todo isto; Java constrói os destinos possíveis para os usuários da Web [DEC 95].

A linguagem de Programação Java foi recebida agradavelmente pela comunidade mundial dos desenvolvedores de software e fornecedores de Internet. Os usuários da Internet e a World Wide Web beneficiam-se do acesso a aplicações seguras, independentes da plataforma e que podem chegar a qualquer lugar sobre a Internet. Os desenvolvedores de software criam aplicações em Java e beneficiam-se desenvolvendo código somente uma vez, não precisando "*portar*" suas aplicações para cada plataforma em software e hardware [ARN 96].

Para muitos, Java, é conhecido como uma ferramenta para criar applets para a World Wide Web. Applet é um termo usado em Java para aplicações pequenas que rodam dentro de uma página web [ARN 96]. Um applet pode realizar tarefas e interagir com o usuário sobre o browser de sua página sem usar recursos do servidor Web. Alguns applets podem interagir com o servidor para seus próprios propósitos, mas só para seus próprios assuntos [ARN 96].

Java é realmente valiosa para redes de ambientes distribuídos como a Web. Entretanto, Java vai mais longe deste domínio ao fornecer uma linguagem de programação de propósito geral poderosa, adequada para construir uma variedade de aplicações que não dependem das características da rede, ou precisa delas por diferentes razões. A habilidade de Java para executar código sobre hosts remotos, de uma maneira segura, é uma necessidade crítica para muitas organizações [ARN 96].

Outros grupos usam Java como uma linguagem de programação de âmbito geral, para projetos onde a independência da máquina é menos importante. A facilidade para programar em Java, e suas características de segurança ajudam a produzir código rapidamente depurado [ARN 96]. Alguns erros comuns de programação nunca acontecem devido a suas características como o coletor de lixo e referências de segurança [ARN 96]. As modernas aplicações baseadas em redes e interfaces de uso gráfico que devem estar presente simultaneamente em múltiplas tarefas, são satisfeitas pelo suporte de Java em "*multithreading*", enquanto o mecanismo de "*exception handling*" facilita a tarefa de procedimentos com condições de erro. Ao passo que estas ferramentas embutidas são poderosas, a linguagem Java é em si mesma, uma linguagem simples na qual os programadores podem vir a ser competentes rapidamente [ARN 96].

Java foi projetada para máxima portabilidade, e é projetada especificamente para ter poucas implementações dependentes no possível. Definindo tudo o possível ao respeito da linguagem e seus ambientes de execução, possibilita aos usuários rodar, compilar código em qualquer lugar e compartilhar código com qualquer pessoa que tem um ambiente Java [ARN 96].

Java compartilha muitas características das linguagem comuns com vários linguagens de programação usados na atualidade. Contudo, é diferente de C e C++, Java fornece gerenciamento automático de armazenamento e "*exception handling*", integrado com suporte para processos [ARN 96].

As linguagens funcionais possuem ótimas características para o desenvolvimento de soluções para os mais diversos problemas [GEY 86]. No entanto, a maioria das suas implementações exigem máquinas com grande capacidade de memória e processadores de alto desempenho. Na prática, esses requisitos ou inviabilizam a solução de determinados problemas nas máquinas de que o usuário dispõe ou pelo menos produzem uma relação de custo/beneficio muito pouco atrativa [GEY 86].

Apesar de algumas das linguagens funcionais, como e principalmente LISP, serem muito antigas, a pesquisa por máquinas alternativas que reduzam aqueles requisitos tem continuado até os dias atuais. Inclusive, respondendo ao crescente interesse em aplicações classificadas na área da Inteligência Artificial, para as quais LISP é uma das principais ferramentas, os centros de pesquisa tem aumentado os seus esforços na busca de soluções para os problemas acima [GEY 86].

Os caminhos tomados por inúmeros pesquisadores são diversos e variam desde conceitos de arquitetura de computadores radicalmente novos até a busca de algoritmos mais eficientes para problemas específicos [GEY 86].

Considerando que a implementação de um *Construtor de Funções Java* para ser empregado na programação funcional usando a linguagem LISP, ajudará a solucionar alguns problemas de hardware e software, se utiliza a linguagem Java da Sun Microsystem devido às excelentes características que possui. A utilização do *Construtor de Funções Java* permitirá que funções em código LISP sejam convertidas a funções em código Java, e estas serão embutidas dentro de um pacote chamado funcional para que possa ser utilizado na programação funcional sobre um ambiente orientado a objetos.

A programação funcional orientada a objetos (PFOO) e a integração de dois paradigmas de programação [NG 95], permitirão ganhar novas propriedades para ambos paradigmas num só estilo de programação, e que possam ser aproveitadas em toda sua capacidade quando modela-se um fenômeno do mundo real e cotidiano sobre um ambiente como a Internet, permitindo que os programas sejam mais rápidos e fácies durante sua codificação, leitura e modificação.

O presente trabalho tem como objetivo a descrição da implementação do *Construtor de Funções Java* usando a linguagem Java da Sun Microsystem, para o qual será necessário usar um *Interpretador Lisp* da linguagem funcional LISP implementado em Java. O *Interpretador Lisp* será ligado ao *Construtor de Funções Java* através de uma função, a qual construíra funções em código Java equivalentes às funções da linguagem LISP usadas pelo *Interpretador Lisp*. A visualização do código das funções tanto em LISP como em Java será apresentado através de um applet Java (*LispJ*) especialmente implementado para este caso.

O trabalho será apresentado da seguinte maneira:

- Capítulo 2: Contém uma breve descrição da linguagem Java, assim como da suas características fundamentais que serão usadas para implementar o *Construtor de Funções Java*.

- Capítulo 3: Contém a apresentação de conceitos básicos das linguagens funcionais.

- Capítulo 4: Contém uma breve descrição da linguagem funcional Lisp, assim como do *Interpretador Lisp*.

- Capítulo 5: Contém a descrição do *Construtor de Funções Java*.

- Capitulo 6: Apresenta as principais conclusões do trabalho

2 Introdução à linguagem Java

Nos últimos anos observa-se o surgimento de múltiplas arquiteturas de hardware incompatíveis, cada uma suportando múltiplos sistemas operativos incompatíveis, com cada plataforma operando com uma ou mais interfaces de uso gráfico (*GUI*), também incompatíveis. O crescimento da Internet, a World Wide Web e o comércio eletrônico têm introduzido novas dimensões de complexidade dentro do processo de desenvolvimento. Imagine ser um desenvolvedor de software de aplicação. Sua linguagem de eleição é C ou C++. Agora suponha deparar-se com tudo isto para fazer trabalhos de aplicação num ambiente distribuído cliente-servidor [SUN 95b].

As ferramentas para desenvolver aplicações não parecem ajudar muito. Ainda estamos fazendo frente aos mesmos problemas antigos; a nova tecnologia orientada a objetos parece agregar novos problemas sem resolver os antigos [SUN 95b].

Agora há um caminho melhor, este é o ambiente da linguagem de programação Java da Sun Microsystem. Imagine que:

- Sua linguagem de programação é orientada a objetos,
- Seu ciclo de desenvolvimento é muito mais rápido porque Java é interpretada,
- Suas aplicações são portáveis através de múltiplas plataformas. Escreve-se uma aplicação e não é necessário portá-la, ela roda sem nenhuma modificação sobre múltiplos sistemas operativos e arquiteturas de hardware,
- Suas aplicações são robustas porque Java gerência o sistema e o tempo de execução na memória,
- Suas aplicações gráficas interativas têm alto desempenho porque a atividade dos processos concorrentes múltiplos em suas aplicações são suportadas por múltiplos processos embutidos no ambiente Java,
- Suas aplicações são adaptáveis para mudança de ambiente porque pode-se baixar dinamicamente o código dos módulos de algum lugar da rede,
- Seus usuários finais podem ter confiança que suas aplicações são seguras, ainda que todo o código seja baixado da Internet, o sistema em tempo de execução de Java tem embutido proteção contra vírus e adulterações.

Não precisa sonhar com estas características. O ambiente da linguagem de programação Java já as fornece. Java foi projetada para enfrentar os desafios do desenvolvimento de aplicações no contexto heterogêneo, e ambientes distribuídos de redes. Suas maiores vantagens são: a entrega segura de aplicações que consomem o mínimo dos recursos do sistema, pode rodar sobre qualquer plataforma de hardware e software, e pode ser estendido dinamicamente [SUN 95b].

Java, foi criada como parte de um projeto de pesquisa para desenvolver software avançado para uma grande variedade de dispositivos de rede e sistemas embutidos. A finalidade foi desenvolver um ambiente pequeno, seguro, portátil, e distribuído operando em tempo real. Quando o projeto iniciou-se, C++ foi a linguagem escolhida. Após algum tempo começaram a surgir dificuldades com o uso desta linguagem de programação, sendo que os problemas encontrados foram resolvidos criando-se um novo ambiente

para a mesma. As decisões de desenho e arquitetura foram extraídas de uma variedade de linguagens tais como Eiffel, SmallTalk, Objetive C. O resultado é um ambiente ideal para desenvolvimento seguro e distribuído em aplicações para usuários finais baseadas em redes rodando em ambientes com dispositivos embutidos em redes para a World Wide Web.

As exigências projetadas por Java são gerenciadas pela natureza do ambiente de computação nas quais deve ser desenvolvido o software. O grande crescimento da Internet e a World Wide Web nos conduzem a desenvolver novas formas de distribuição de software. Para viver no mundo de distribuição e comércio eletrônico, Java deve possibilitar o desenvolvimento seguro, e de alto desempenho em aplicações altamente robustas rodando sobre múltiplas plataformas em redes heterogêneas e distribuídas.

A operação sobre múltiplas plataformas em redes heterogêneas invalida o esquema tradicional de distribuição binária, lançamento, atualização e assim por diante. Para sobreviver nesta selva, Java deve ser de arquitetura neutra, portátil e adaptável dinamicamente.

O sistema Java que surge para suprir essas necessidades é simples, assim pode ser facilmente utilizado na programação por mais desenvolvedores; de fácil aprendizagem, pois devido a sua familiaridade com desenvolvedores comuns que podem facilmente aprender Java; orientada a objetos para tomar vantagens da metodologia de desenvolvimento de softwares modernos e ajustar-se em aplicações distribuídas cliente-servidor; multiprocessos, para alto desempenho em aplicações que precisam realizar múltiplas atividades concorrentes, tais como multimídia e interpretada para máxima portabilidade e capacidades dinâmicas. Além das propriedades citadas acima, Java também inclui uma coleção de "*buzzword*". A fim de entender o potencial de Java, vai-se a examinar cada uma dessas "*buzzword*" em termos dos objetivos do projeto da Sun, que foi usado para criar a linguagem.

2.1 Características de Java

2.1.1 Simples

Java é uma linguagem simples. Um objetivo do projeto foi a criação de uma linguagem que pudesse ser aprendida facilmente por um programador. Outro objetivo do projeto foi desenvolver uma linguagem familiar para a maioria de programadores, para fácil migração. Um programador de C e C++ constatará que Java usa a mesma construção das linguagens C e C++ [FLA 96].

A fim de ficar pequena e familiar, os projetistas de Java removeram algumas características disponíveis em C e C++, principalmente as que conduziam a práticas de programação muito ruins e eram raramente usadas. Por exemplo, Java não suporta a

sentença *goto*. Em vez disso, fornece sentenças etiquetadas com *break*, *continue* e "*exception handling*" [FLA 96]. Java não usa arquivos de cabeçalho, eliminando o preprocessador C. Uma vez que Java é orientada a objetos, as construções de C como *struct* e *union* foram removidas. Java também elimina o operador overloading e características de múltipla herança do C++ [FLA 96].

Talvez a mais importante simplificação, entretanto, é que Java não usa ponteiros. Desde que Java não tem estruturas, e os arrays e strings são objetos, não há necessidade para ponteiros. Java manipula automaticamente referenciando e dereferenciando objetos. Java também implementa automaticamente uma coleção de lixo ("*garbage collection*"), liberando ao usuário da preocupação pelo gerenciamento da memória. Liberado das preocupações com ponteiros perigosos, referências inválidas, vazamento de memória, o programador se concentra mais na funcionalidade dos programas [FLA 96].

2.1.2 Orientada a objetos

Java é uma linguagem de programação orientada a objetos. Para um programador, isto significa que a focalização sobre os dados e métodos que manipulam estes dados, permite chegar estritamente a pensar em termos de procedimentos [FLA 96]. O programador acostumado à programação baseada em procedimentos como C, precisa mudar a forma de projetar seus programas quando usa Java. O potencial do novo paradigma, está na facilidade e rapidez de adaptar-se a ele [FLA 96].

Num sistema orientado a objetos, uma *classe* é uma coleção de dados e métodos que operam sobre esses dados. Tomados juntos, os dados e métodos descrevem o estado e comportamento de um *objeto*. As classes são dispostas hierarquicamente, assim, as subclasses podem herdar o comportamento de suas superclasses. A hierarquia das classes sempre tem uma classe raiz, esta é uma classe com comportamento geral [FLA 96].

Java possui um conjunto de classes organizadas em pacotes ("*packages*"), que podem ser usadas nos programas. Por exemplo, Java fornece classes para criar componentes de Unidades de Interface Gráfica (GUI: *java.awt package*), classes que manipulam a entrada e saída de dados (*java.io package*), e classes que suportam funcionalidade com redes (*java.net package*). A classe *Object* em *java.lang package* serve como raiz da hierarquia da classe Java [FLA 96].

Ao contrario de C++, Java foi projetada para ser orientada a objetos. Quase tudo em Java são objetos. Os tipos numéricos simples, caracteres e booleanos são somente exceções. Enquanto Java é projetada para parecer-se com C++, verifica-se que Java remove muitas das complexidades desta linguagem muito familiar [FLA 96].

2.1.3 Robusta

Sua origem como uma linguagem para software de projetos eletrônicos [FLA 96], indica que Java foi projetada para escrever software robusto e de alta seguridade. Java certamente não elimina a necessidade de software seguro e de qualidade, e as continuas possibilidades de escrever software duvidoso [FLA 96]. Entretanto, Java elimina certos tipos de erros de programação, o que torna consideralvelmente fácil escrever software seguro.

Java é uma linguagem fortemente tipada, que permite uma verificação extensiva em tempo de compilação para problemas potenciais "*type-mismatch*" [FLA 96]. Java é mais fortemente tipada que C++, e requer métodos declarados explicitamente, e não suporta as declarações implícitas ao estilo C. Estes requerimentos estritos asseguram que o compilador possa capturar a invocação de erro dos métodos, os quais guiam a programas mais seguros [FLA 96].

Uma das mais significativas melhorias em termos de segurança é o modelo de memória de Java. Não suporta ponteiros, os quais eliminam a possibilidade de sobreescrever na memória e corromper os dados. Similarmente, a coleção de lixo automático de Java, evita vazamento de memória e outros bugs perniciosos relacionados à alocação e desalocação dinâmica de memória. O interpretador Java também realiza um número de verificações em tempo de execução, tais como, verificar que todos os arrays e strings acessados estão dentro dos limites [FLA 96].

A "*exception handling*" é outra característica de Java que faz os programas mais robustos. Uma "*exception*" é um sinal de alguma classe que indica uma condição excepcional, como um erro ocorrido [FLA 96].

2.1.4 Segura

Segurança é uma preocupação importante, uma vez que Java é destinada para ser usada em ambientes de redes. Sem nenhuma garantia de segurança, certamente não se pode baixar um applet de uma site por acaso na Internet e rodá-lo sobre um computador pessoal. Java implementa vários mecanismos de segurança para proteger seu código. Todos os mecanismos de segurança são baseados na premissa de que nada é para ser confiado [FLA 96].

A segurança anda junto com a robustez. O modelo de alocação de memória de Java é uma das principais defessas contra um código maldoso. Java não suporta ponteiros, assim um programador não pode corromper a memória. As referências em código Java compiladas são resolvidas para endereços reais de memória em tempo de execução pelo interpretador Java. O sistema de Java em tempo de execução usa um processo de verificação dos bytecodes para assegurar que o código carregado sobre a rede não viola nenhuma restrição da linguagem Java. Parte destes mecanismos de

segurança implica em como as classes são carregadas através da rede. A rede é com clareza um ambiente perigoso [FLA 96]. Java oferece uma razoável área intermediária entre ignorar o problema de segurança e ser paralisado por ele [FLA 96].

2.1.5 Arquitetura Neutra

Os programas são compilados para uma arquitetura neutra em formato de bytecodes [FLA 96]. A principal vantagem desta aproximação é que isto permite que uma aplicação Java rode sobre qualquer sistema tão rápido quanto o sistema implemente a Máquina Virtual Java (JVM). Desde que Java foi projetada para criar aplicações baseadas em redes, é importante que as aplicações Java sejam capazes de rodar sobre diferentes tipos de sistemas encontrados sobre a Internet [FLA 96].

Como nas aplicações desenvolvidas para o mercado de software provavelmente se deseja desenvolver versões de aplicações que possam rodar sobre Pcs, Macs e estações de trabalho Unix, incrementa-se a dificuldade de produzir software para todas as possíveis plataformas. Uma aplicação escrita em Java, entretanto, pode rodar sobre todas as plataformas [FLA 96], com a aparência apropriada e o comportamento de cada plataforma.

2.1.6 Portátil

Ser de arquitetura neutra é uma grande parte de ser portátil. Mas Java vai mais adiante, assegurando que não há nenhum aspecto "dependentes da implementação" na especificação da linguagem [FLA 96]. Por exemplo, Java especifica explicitamente o tamanho de cada um dos tipos de dados primitivos, bem como seu comportamento aritmético.

O ambiente Java a si mesmo é portátil para novas plataformas de hardware e sistemas operativos. O compilador Java foi escrito em Java, enquanto o sistema de tempo de execução de Java foi escrito em ANSI C [FLA 96].

2.1.7 Alto desempenho

Java é uma linguagem interpretada, assim nunca vai ser mais rápida que uma linguagem compilada como C. De fato, Java é cerca de vinte vezes mais lenta que C. Mas esta rapidez é mais que adequada para rodar aplicações interativas, GUI e aplicações baseadas em redes, onde a aplicação está freqüentemente parada, esperando por uma decisão do usuário para fazer alguma coisa ou esperando por dados da rede [FLA 96].

Naturalmente, existem algumas situações onde o desempenho é critico. Para suportar essas situações, os projetistas de Java estão trabalhando sobre compiladores "*just in time*" que podem traduzir os bytecodes Java em código máquina para uma CPU particular em tempo de execução. O formato bytecodes de Java foi desenhado com esse compilador "*just in time*", assim o processo de gerar código máquina é razoavelmente simples e produz um bom código. De fato, a Sun afirma que o desempenho dos bytecodes convertidos para código máquina é quase tão bom como os nativos C ou C++ [FLA 96].

Enquanto, considerando-se o desempenho, é importante lembrar-se que Java desce no espectro das linguagens de programação disponíveis. Por lado, do espectro existem linguagens scripts totalmente interpretadas e de alto nível tais como Tcl e o Unix Shell. Essas linguagens são grandes protótipos e são altamente portáveis, mas também são muito lentas. Por outro lado, do espectro têm-se linguagens compiladas em baixo nível como C e C++. Estas linguagens oferecem alto desempenho, mas perdem em termos de segurança e portabilidade [FLA 96].

O desempenho dos bytecodes interpretados por Java é muito melhor que as linguagens scripts de alto nível, mas ainda oferece a simplicidade e portabilidade destas linguagens. E, enquanto Java não é rápido como uma linguagem compilada, ela fornece um ambiente de arquitetura neutra no qual é possível escrever programas seguros [FLA 96].

2.1.8 Multiprocessos

Numa aplicação de redes baseadas numa GUI tais como browsers Web, é fácil imaginar múltiplas ações ocorrendo ao mesmo tempo. Um usuário pode estar escutando um audio clipe enquanto pesquisa informação e no background do browser estar carregando uma imagem. Java é uma linguagem multiprocessos; fornecendo suporte para múltiplos processos de execução que podem manejar diferentes tarefas [FLA 96].

Um beneficio dos multiprocessos é que melhora o desempenho interativo de aplicações gráficas para o usuário. Infelizmente para o programador, escrever código em C ou C++ que lidam com múltiplos processos pode dar um pouco de dor de cabeça. A principal dificuldade está em fazer implementações de rotinas seguras, que podem ser rodadas por múltiplos processos concorrentes [FLA 96].

A rotina muda o valor de um estado variável, por exemplo, só um processo pode ser executado por uma rotina de uma só vez. Com C ou C++ escrever uma aplicação multiprocessos implica em obter travas para rodar certas rotinas e realizá-las. Escrever código que explicitamente manipula travas é problemático, e pode facilmente conduzir a situações deadlock [FLA 96].

Java faz programação com processos muito mais fácil, fornecendo suporte para processos embutido na linguagem. O *package java.lang* fornece uma classe *Thread* que suporta métodos para iniciar, rodar, parar e conferir o estado de um processo [FLA 96].

2.1.9 Dinâmica

A linguagem Java, foi projetada para adaptar-se a um ambiente em evolução, Java é uma linguagem dinâmica. Por exemplo, Java carrega classes tão rápido como sejam necessitadas, até mesmo através da rede. As classes em Java também têm uma representação em tempo de execução. Ao contrário de C ou C++, um objeto passado para um programa pode encontrar a classe a qual pertence conferindo o tipo de informação em tempo de execução. As definições das classes em tempo de execução em Java faz possível para ligar as classes dinamicamente dentro de um sistema de execução [FLA 96]. No caso do browser *HotJava* e aplicações similares, o código executável interativo pode ser carregado de qualquer lugar, o qual possibilita uma atualização transparente de aplicações. O resultado são os serviços on-line que constantemente evoluem, eles podem permanecer inovados e novos, atraindo mais consumidores, e estimular o crescimento do comércio eletrônico na Internet [SUN 95b].

2.2 O sistema base Java

O sistema completo Java inclui um número de bibliotecas de classes e métodos usados por desenvolvedores para criar aplicações multiplataforma [SUN 95b]:

- *java.lang*, coleção de tipos básicos que são sempre importados dentro de qualquer unidade de compilação. Aqui encontram-se as declarações de objetos (raiz da classe hierárquica) e classes, mais processos, exceções, tipos de dados primitivos e uma variedade de outras classes fundamentais.
- *java.io*, fluxos de acesso por acaso a arquivos. Aqui encontram-se bibliotecas padrões I/O, familiares com o sistema Unix.
- *java.util*, recipiente e classes úteis. Aqui encontram-se classes tais como dicionário, hashtable, e outras, bem como técnicas de codificação, de decodificação, datas e classes de tempo.
- *java.awt*, esta biblioteca contém classes para componentes de interface gráfica tais como cores, fontes e controles, como botões e scrollbars.
- *java.net*, fornece suporte para sockets, telnet, interfaces e URLs.

2.3 O ambiente Java

Tomando individualmente, as características discutidas anteriormente podem ser encontradas numa variedade de softwares desenvolvidos em diversos ambientes. O que é novo, é a maneira na qual Java, e seu sistema em tempo de execução combina-se para produzir um sistema de programação poderoso e flexível [SUN 95b].

Desenvolver aplicações usando Java, dará como resultado um software que é portátil através de máquinas de arquiteturas múltiplas, sistemas operativos e interfaces de uso gráfico, seguras e de alto desempenho. Com Java, o trabalho de desenvolver softwares é muito mais fácil.

2.4 Java simples e familiar

2.4.1 Principais características da linguagem Java

Java acompanha C++ em alguns pontos, os quais possuem o beneficio de ser familiares para muitos programadores [SUN 95b]. Esta seção descreve as características essenciais de Java e pontos onde a linguagem diverge de seus antecessores C e C++.

2.4.1.1 Tipos de dados primitivos

Tipos de Dados primitivos em Java são objetos. Até os mesmo tipos de dados primitivos podem ser encapsulados dentro de uma biblioteca fornecendo objetos quando precisa-se. Java acompanha C e C++ regularmente e atentamente no seu conjunto de dados de tipo básico como uma dupla de exceções menores. Existem apenas três grupos de tipos de dados primitivos, chamados tipos numéricos, booleanos e arrays [SUN 95].

2.4.1.1.1 Dados de tipo numérico

São inteiros de 8-bit byte: *short* de 16-bit byte, *int* de 32-bit byte, *long* de 64-bit byte. O dado tipo 8-bit byte em Java substitui ao antigo tipo de dado *char* de C e C++. Java emprega uma interpretação diferente sobre o dado de tipo *char*. Não existe tipo *unsigned* especificado para tipos de dados inteiros em Java. Os números reais são *float* de 32-bit byte e *double* de 64-bit byte. Os tipos numéricos reais e suas operações

aritméticas são definidas pelas especificações da IEEE 754. Um ponto flutuante de valor literal, como 23.79, é considerado double por omissão [SUN 95b].

2.4.1.1.2 Dados de tipo Caracter

Os dados de tipo caracter da linguagem Java são uma saída do C tradicional. O dado de tipo *Char* em Java define um caracter Unicode de 16-bit byte. Os caracteres Unicode são valores unsigned de 16-bit byte que define código caracter no alcance de 0 a 65,535. Escreve-se uma declaração como:

char myChar = 'Q';

Obtém-se um tipo Unicode (valor unsigned de 16-bit byte) que é inicializado para o valor Unicode do caracter Q. Adotando um conjunto padrão de caracteres Unicode para estes dados de tipo caracter, as aplicações da linguagem Java estão sujeitas à internacionalização e localização, expansão do mercado para aplicações World Wide Web [SUN 95b].

2.4.1.1.3 Dados de tipo booleano

Java tem agregado dados de tipo *booleano* como um tipo primitivo. As variáveis booleanas Java assumem o valor TRUE ou FALSE. Um booleano Java, é um dado de tipo diferente do C comum, um tipo booleano Java não pode ser convertido para nenhum tipo numérico [SUN 95b].

2.4.1.2 Operadores aritméticos e relacionais

A familiaridade dos operadores C e C++ pode aplicar-se. Uma vez que Java carece de dados de tipo unsigned, o operador >>> tem sido adicionado à linguagem para indicar um tipo unsigned "*right shift*" (lógico). Java também usa o operador + para concatenar strings [SUN 95b].

2.4.1.3 Arrays

Em contraste a C e C++, os arrays da linguagem Java são objetos da linguagem de primeira classe. Um array em Java é um objeto real com representação em tempo de

execução. Declaram-se e alocam-se arrays de quaisquer tipo, alocam-se arrays de arrays para obter arrays multidimensionais [SUN 95b].

Exemplo:

Declara-se um array de pontos como:

Ponto mPontos[];

mPontos[], é um array de pontos não inicializados. Com inicialização é da seguinte forma:

mPontos = new Ponto[10];

Para alocar um array de dez referências para *mPontos* que estão inicializados pela referência nula. Notar que esta alocação de um array não está realmente alocando nenhum objeto da classe *Ponto*, para alocar os objetos *Ponto*, faz-se:

```
int i;
for(i = 0; i<10; i++){
mPontos[i]= new Ponto();
}
```

Para acessar elementos de *mPontos* pede-se fazer via o estilo normal C, indexando, mas todos os acessos a arrays são verificados para assegurar-se que seus índices estão dentro do alcance dos arrays. É gerada uma excepção quando o índice está fora do alcance do array [SUN 95b].

2.4.1.4 Strings

Os strings são objetos da linguagem Java, não pseudos arrays de caracteres como em C. Existem realmente dois tipos de objetos string: a classe String, é implementada apenas para leitura de objetos e a classe *StringBuffer*, para objetos string que podem ser modificados. Os strings são objetos da linguagem Java, o compilador Java segue a tradição do C ao fornecer strings no estilo C, isto é, o compilador Java compreende que os string de caracteres cercados por aspas duplas são marcados para serem instanciados como um objeto String. Assim, a declaração:

String ola = "Oi, pessoal!";

instância um objeto de classe String através das vistas e inicializações com um caracter String contendo a representação caracter Unicode de *"Oi, pessoal!"* .

Java estendeu o significado do operador + para indicar a concatenação de um string. Assim, pode-se escrever:

System.out.println(" Existem" + numero + "caracteres no arquivo.");

fragmento de código que concatena o string *"Existem "* com o resultado da conversão do valor numérico *numero* para um string, e concatenar isto com o string *"caracteres no arquivo."*. Logo, imprime o resultado dessas concatenações usando a saída padrão [SUN 95b].

Justamente como um array de objetos, os objetos string fornecem o método *length()* para obter o número de caracteres do string [SUN 95b].

2.4.1.5 Interrupções multi-níveis

Java não tem a declaração *goto*. Para quebrar o continuar múltiplos laços ou construtores *switch*, pode-se empregar etiquetas sobre laços e construtores *switch* , e então romper para, ou continuar para o bloco chamado pela etiqueta. Exemplo:

```
teste: for( int i= index; i+max1 <= max2; i++){
          if(charAt(i) == c0){
            for( int k = 1; k < max1; k++){
              if(charAt(i+k)!= str.charAt(i)){
                continue teste;
                    }
             }
            }
           }
```

2.4.1.6 Gerência de memória e coletor de lixo

Os programadores de C e C++, têm problemas de gerenciar memória explicitamente: alocação de memória, liberação de memória, quando liberar memória. Explicitamente o gerenciamento de memória fornece um frutífero código de bugs, vazamento de memória, e um pobre desempenho [SUN 95b].

Java remove completamente o gerenciamento de memória carregada pelo programador. Não existem os ponteiros ao estilo C, ponteiros aritméticos e malloc. A coleta de lixo automático é uma parte integral de Java e de seu sistema em tempo de execução. Enquanto Java tem um novo operador para alocar memória para objetos, não há funções explícitas livres. Para alocar um objeto, o sistema em tempo de execução guarda o caminho do estado do objeto e automaticamente reduz a memória quando os objetos usados não são grandes, e liberando memória para futuros usos.

O modelo de gerenciamento de memória de Java está baseado em objetos e referências para objetos [SUN 95b]. Uma vez que Java não tem ponteiros, todas as referências para fixar o armazenamento, o que na pratica significa que todas as

referências para um objeto, são através de *"handles"* simbólicos. O gerenciamento da memória de Java guarda o caminho das referências para os objetos. Quando um objeto não tem referência, o objeto é um candidato para o coletor de lixo [SUN 95b].

O modelo de alocação de memória de Java e a coleta de lixo automático fazem a programação mais fácil, eliminam completamente as classes de bugs e em geral fornecem melhor desempenho que o obtido através do gerenciamento de memória explicito [SUN 95b].

Exemplo: (exemplo da "on-line Java Language programmer's guide")

```
class ReverseString{
        public Statitc String reverseIt(String source){
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);
        for( i = (len-1); i>=0; i--){
                dest.appendChar(source.charAt(i));
          }
        return dest.toString();
    }
}
```

A variável *dest* é usada como uma referência de objeto temporal durante a execução do método *reverseIt*. Quando *dest* sai do escopo, a referência a esse objeto desaparece e este é então o candidato para o coletor de lixo [SUN 95b].

2.4.1.7 O coletor de lixo de segundo plano

O coletor de lixo Java alcança alto desempenho com a vantagem da natureza do comportamento do usuário quando interage com o software de aplicação tal como o browser HotJava. O usuário típico de uma aplicação interativa tem muitas pausas naturais na contemplação de cenas ou pensando o que fazer a seguir. O sistema de tempo de execução Java aproveita os períodos inativos e roda o coletor de lixo num processo de prioridade baixa quando outros processos não estão concorrendo para ciclos CPU. O coletor de lixo junta e compacta memória não usada, incrementando a probabilidade para adequar recursos de memória disponíveis quando necessita-se durante períodos de uso interativo pesado [SUN 95b].

2.4.1.8 Características removidas de C e C++

Esta seção discute as características removidas de C e C++ na evolução de Java O primeiro passo foi eliminar a redundância de C e C++. Em muitas das formas a linguagem C evolui dentro de uma coleção de características sobrepostas, fornecendo também muitas formas para dizer a mesma coisa, enquanto em muitos casos não fornece características necessárias. C++, em uma tentativa para agregar "classes em C", simplesmente agrega mais redundância enquanto conserva muitos dos problemas inerentes de C [SUN 95b].

2.4.1.8.1 typedef, define, e preprocessadores

O código fonte escrito em Java é simples. Não tem preprocessador, *#define* e capacidades relacionadas, não tem *typedef* e não tem nenhuma necessidade de arquivos de cabeçalho. Em vez de arquivos de cabeçalho, os arquivos fonte da linguagem Java fornecem as definições de outras classes e seus métodos.

O problema maior com C e C++ é o tempo que precisa-se para entender o código de outro programador: tem-se que ler todos os arquivos do cabeçalho, todas as *#define* relacionadas a todas as *typedef* antes de iniciar o análise do programa. Em essência, a programação com *#define* e *typedef* resulta numa nova linguagem de programação, incompreensível para outras pessoas que não sejam o criador, frustrando os objetivos da boa prática da programação [SUN 95b].

Em Java, obtém-se efeitos de *#define* usando constantes. Obtém-se os efeitos de *typedef* declarando classes. Após isto efetivamente as classes declaram um novo tipo. Não precisa-se de arquivos de cabeçalho porque o compilador Java compila as definições das classes em uma forma binária que conserva todos os tipos de informações através do tempo de ligação [SUN 95b]. Removendo toda esta bagagem, Java torna-se de contexto livre. Os programadores podem ler e entender o código e, mais importante, modificar e reusar o código muito mais rápido e facilmente [SUN 95b].

2.4.1.8.2 Estruturas e uniões

Java não tem estruturas ou uniões como tipos de dados complexos. Não são necessárias as estruturas e uniões quando têm-se classes, o mesmo efeito é alcançado simplesmente declarando uma classe com as variáveis de instância apropriadas [SUN 95b].

Exemplo:

Declaração da classe *Ponto*:

```
Class Ponto extend  Object{
        double x;
        double y;
métodos para acessar as variáveis de instância
}
```

O seguinte fragmento de código declara uma classe chamada Retângulo, que usa objetos da classe *Ponto* como variáveis de instância:

```
Class Rectangulo extend Object{
        Ponto lowerLeft;
        Ponto upperRight;
        métodos para acessar as variáveis de instância
}
```

em C, necessita-se definir estas classes como estruturas. Em Java, simplesmente declaram-se as classes. Pode-se fazer as instâncias varáveis como privadas ou como públicas segundo a necessidade para a implementação de outros objetos [SUN 95b].

2.4.1.8.3 Funções

Java não tem funções. A programação orientada a objetos substitui os estilos funcional e procedural. Misturando os dois estilos, conduz a uma confusão e dilui a pureza de uma linguagem orientada a objetos. Tudo o que pode-se fazer com uma função, pode-se fazer em Java, definindo-se uma classe e criando-se métodos para essa classe. Considerando a classe *Ponto* anterior, agregando um método público ao conjunto e acessando as variáveis de instância:

```
class Ponto extend Object{
        double x;
        double y;
        public void setX(double x){
                this.x = x;
      }
public void setY(double y){
        this.y = y;
    }
public double x(){
        return x;
   }
public double y(){
        return y;
   }
} // fim da classe Ponto
```

Ainda que as variáveis de instância *x* e *y* são privadas para esta classe, o acesso às mesmas é via métodos públicos da classe [SUN 95b].

Exemplo: mostra-se como usar objetos da classe *Ponto* como objetos da classe *Rectangulo:*

```
class Rectangulo extend object{
        Ponto lowerLeft;
        Ponto upperRight;
public void setEmptyRect(){
        lowerLeft.setX(0.0);
        lowerLeft.setY(0.0);
        upperRight.setX(0.0);
        upperRight.setY(0.0);
}
}
```

Isto não quer dizer que usar funções e procedimentos é inerentemente errado. Mas eliminando as funções, o trabalho do programador é simplificado: trabalha somente com classes e seus métodos [SUN 95b].

2.4.1.8.4 Herança múltipla

A herança múltipla e todos os problemas que gera foram descartados do Java. As características desejáveis de múltipla herança são fornecidas por interfaces, conceitualmente similar aos protocolos do Objective C [SUN 95b].

Uma interface não é uma definição de um objeto. Esta é uma definição de um conjunto de métodos que implementam um ou mais objetos. Uma importante saída da interface é que elas declaram somente métodos e constantes. As variáveis não podem ser definidas nas interfaces [SUN 95b].

2.4.1.8.5 Declarações goto

Java não tem declarações *goto*. A eliminação do *goto* conduz a uma simplificação da linguagem, por exemplo, não existem regras acerca dos efeitos de uma *goto* dentro de uma sentença *for*. Estudos sobre aproximadamente 100,000 linhas de código C determinam que irregularmente 90 de 100 sentenças *goto* foram usados puramente para obter efeitos de ruptura de laços [SUN 95b].

2.4.1.8.6 Operadores overloading

Os efeitos de um operador overloading (sobrecarga) pode ser alcançado facilmente declarando uma classe, uma variável de instância apropriada e métodos apropriados para manipular essas variáveis [SUN 95b].

2.4.1.8.7 Coações automáticas

Java proíbe o estilo C e C++ de coação automática. A coação nos dados de um tipo para outro tipo de dado pode resultar em perda de precisão, isto deve-se fazer só explicitamente usando um molde [SUN 95b].

Exemplo:

```
int myInt ;
double myFloat = 3.141592;
myInt = myFloat;
```

A atribuição de *myFloat* para *myInt* gera como resultado um erro de compilação, indicando uma possível perda de precisão e deve ser usado um molde explicito. Assim:

```
int myInt;
double myFloat = 3.141592;
myInt = (int)myFloat;
```

2.4.1.8.8 Ponteiros

Vários estudos concordam que os ponteiros são uma das características primárias que possibilitam aos programadores a inserir bugs dentro do código. As estruturas são perdidas, os arrays e strings são objetos, a necessidade por ponteiros para esses construtores é excluída. Assim, Java não tem ponteiros. Qualquer tarefa que requer um array, estruturas e ponteiros em C, pode ser mais fácil e de desempenho mais seguro declarando-se objetos e arrays de objetos. Em vez de manipulação complexa de ponteiros sobre arrays de ponteiros, acessa-se arrays para seus índices aritméticos. O sistema em tempo de execução de Java confere toda a indexação para assegurar-se que os índices estão dentro dos limites de um array [SUN 95b].

2.5 Java é orientada a objetos

Para permanecer atualizado na prática no desenvolvimento de software moderno, Java é orientada a objetos. O motivo de projetar uma linguagem orientada a objetos não é simplesmente acompanhar a última moda de programação. O paradigma orientado a objetos identifica-se bem com as necessidades cliente-servidor e o software distribuído. Os benefícios da tecnologia de objetos são rapidamente realizados por mais organizações que movem suas aplicações para o modelo distribuído cliente-servidor [SUN 95b].

Infelizmente, o conceito de orientação a objetos permanece mal compreendido. A visualização da programação orientada a objetos é justamente uma nova forma de organizar seu código fonte. Embora pode ter alguns méritos esta visualização, uma vez que pode alcançar resultados com técnicas de programação orientada a objetos que não pode-se ter com técnicas procedurais [SUN 95b].

Uma importante característica que distingue os objetos dos procedimentos ordinários ou funções é que um objeto pode ter um tempo de vida maior que um objeto criado por estes [SUN 95b]. No modelo distribuído cliente-servidor isto cria um potencial para objetos serem criados num lugar, passados através da rede, e armazenados em outro lugar qualquer, possivelmente em uma base de dados, para serem recuperados para futuros trabalhos [SUN 95b].

Como uma linguagem orientada a objetos, Java utiliza os melhores conceitos e características das linguagens orientadas a objetos, principalmente do Eiffel, SmallTalk, Objetive C e C++. Java vai mais longe de C++, estendendo o modelo orientado a objetos e removendo as maiores complexidades de C++. Com as exceções de seus tipos de dados primitivos, tudo em Java é um objeto, e até mesmo os tipos primitivos podem ser encapsulados dentro de objetos, quando há necessidade [SUN 95b].

2.6 A tecnologia de objetos em Java

Para ser considerada *orientada a objetos*, uma linguagem de programação deve suportar no mínimo quatro características[SUN 95b]:

- *Encapsulação*, modularidade e ocultamento da informação implementada (abstração).
- *Polimorfismo*, a mesma mensagem enviada para diferentes objetos resulta em comportamentos dependentes da natureza do objeto que recebe a mensagem.
- *herança*, define-se novas classes e comportamentos baseado em classes existentes para obter código re-usado e código de organização.
- *binding dinâmico*, os objetos podem vir de qualquer lugar, possivelmente através da rede. Envia-se mensagens para objetos sem ter que conhecer seu tipo específico uma vez que escreve-se seu código. O binding dinâmico fornece máxima flexibilidade enquanto um programa é executado.

Java junta estes requerimentos elegantemente, e agrega suporte considerável em tempo de execução para trabalhar mais facilmente quando desenvolve-se um software.

2.7 O que são objetos?

Simplesmente, a tecnologia de objetos é um conjunto de análise, projeto e metodologia de programação que focaliza o projeto modelando as características e comportamentos de objetos no mundo real [SUN 95b].

O que são objetos? Existem softwares para programar modelos. Em nosso dia-a-dia, estamos rodeados por objetos: carros, árvores, ..., assim por diante. Os softwares de aplicação contém objetos: botões sobre interfaces do usuário, listas, menus e assim por diante. Esses objetos têm estado e comportamento. Representa-se todas essas coisas construindo software para objetos, o qual também pode ser definido por seu estado e comportamento [SUN 95b].

Em nosso dia-a-dia necessitamos de transporte, um carro pode ser modelado por um objeto. O carro tem estado (qual velocidade, em qual direção, consumo de gasolina, e assim por diante) e comportamento (partida, parada, virar) [SUN 95b].

2.8 Bases de objetos

Na implementação de objetos seu estado é definido por suas variáveis de instância. As variáveis de instância são privadas para um objeto. A menos que, explicitamente sejam feitas públicas e disponíveis para outras classes "amigáveis", as variáveis de instância de um objeto são inacessíveis fora do objeto [SUN 95b].

O comportamento de um objeto é definido por seus métodos. Os métodos manipulam as variáveis de instância para criar um novo estado; os métodos dos objetos podem também criar novos objetos [SUN 95b].

As variáveis de instância de um objeto (dados), são empacotadas ou encapsuladas dentro de um objeto. As variáveis de instância são rodeadas pelo métodos dos objetos. Com certas exceções bem definidas, os métodos dos objetos estão só significando quais outros objetos podem acessar ou alterar suas variáveis de instância. Em Java as classes podem declarar suas variáveis de instância para serem públicas, desta forma, as variáveis de instância são acessíveis globalmente por outros objetos [SUN 95b].

2.8.1 Classes

Uma *classe* é um software construtor que define as variáveis de instância e métodos de um objeto. Uma classe é um "*template*" que define como um objeto parece-se e comporta-se, enquanto que o objeto é criado ou instanciado da especificação declarada pela classe. Obtém-se objetos concretos instanciando classes definidas previamente. Pode-se instanciar muitos objetos de uma definição de uma classe [SUN 95b]. O exemplo a seguir mostra uma declaração básica de uma classe simples chamada *Ponto*:

```
class Ponto extends Object {
        public double x;   /* variável de instância */
        public double y;   /* variável de instância */
}
```

Esta definição simplesmente define um "*template*" do qual os objetos reais podem ser instanciados.

2.8.2 Instanciar um objeto de sua classe

Tendo declarado o tamanho e a forma da classe *Ponto* acima, qualquer outro objeto pode criar um objeto *Ponto*, uma instância da classe *Ponto*, com um fragmento de código como este:

```
Ponto myPonto;      // declara uma variável para referênciar a um objeto Ponto
myPonto = new Ponto(); // aloca uma instância de um objeto Ponto
```

Agora, pode-se acessar as variáveis deste objeto *Ponto* referenciando os nomes das variáveis, habilitado com o nome do objeto:

```
myPonto.x = 10.0;
myPonto.y = 25.7;
```

Este esquema de referência, trabalha de forma similar às referência da estrutura C. Uma vez que as variáveis de instância de *Ponto* foram declaradas públicas na declaração da classe. A declaração da classe *Ponto* necesitará fornecer métodos de acesso para fixar e pegar suas variáveis [SUN 95b].

2.8.3 Construtores

Quando declara-se uma classe em Java, pode-se declarar construtores opcionais que realizam inicializações quando instancia-se objetos da classe. Também pode-se declarar um finalizador opcional. Exemplo usando a classe *Ponto* anterior:

```
class Ponto extends Object {
        public double x; /* variável de instância */
        public double y; /* variável de instância */

Ponto( ) {      /* construtor para inicializar valores por omissão */
        x = 0.0;
        y = 0.0;
}

/* construtor para inicializar valores específicos */

Ponto(double x, double y) {
/* variáveis de instância para ser passadas como  parâmetros */
        this.x = x;
        this.y = y;
        }
}
```

Métodos com o mesmo nome que a classe no fragmento de código acima são chamados *Construtores*. Quando cria-se (instância) um objeto da classe *Ponto*, o método construtor é invocado para realizar qualquer inicialização que precise, neste caso para fixar as variáveis de instância em estado inicial. O seguinte exemplo é uma variação da classe *Ponto* anterior. Quando precisa-se criar e inicializar um objeto *Ponto*, pode-se obter as inicializações para seus valores por omissão, ou pode-se inicializar para valores específicos [SUN 95b]:

```
Ponto lowerLeft;
Ponto upperRight;

lowerLeft = new Ponto();  /* inicialização de valores por omissão */
upperRight = new Ponto(50.0, 100.0);  /* inicialização não nula */
```

O construtor especifico que é usado quando cria-se um novo objeto *Ponto* é determinado pelo tipo e número dos parâmetros da nova invocação [SUN 95b].

A variável ***this*** do exemplo anterior, refere-se ao objeto recebido. Usa-se *this* para clarificar a quais variáveis se está referindo. No exemplo acima, os construtores são simples conveniências para a classe *Ponto*. Existem situações, onde os construtores são uma necessidade, especialmente em casos onde o objeto sendo instanciado deve instanciar outros objetos [SUN 95b]. Ilustraremos isto declarando uma classe Retângulo que usa dois objetos *Ponto* para definir seus limites:

```
class Retangulo extends Object {
        private Ponto lowerLeft;
        pivate Ponto upperRight;

        Retangulo(){ // construtor
                lowerLeft = new Ponto();
                upperRight = new Ponto();
        }
...
}
```

Neste exemplo, o construtor *Retangulo()* é vitalmente necessário para assegurar que os dois objetos *Ponto* são instanciados quando o objeto Retângulo é instanciado, senão, o objeto Retângulo busca referências de pontos que foram alocados incorretamente [SUN 95b].

2.8.4 Métodos e mensagens

Quando um objeto precisa de outro objeto para o mesmo trabalho, na programação orientada a objetos, o primeiro objeto envia uma mensagem para o segundo objeto. Em resposta, o segundo objeto seleciona o método apropriado a invocar. A invocação dos métodos Java parecem similares a funções em C e C++ [SUN 95b].

Usando o envio das mensagens no paradigma da programação orientada a objetos, pode-se construir redes completas de objetos que trocam mensagens entre eles para mudar de estado. Esta técnica de programação é uma das melhores formas de criar modelos e simulações de sistemas complexos do mundo real [SUN 95b]. Assim, redefine-se a declaração da classe *Ponto* anterior tal que suas variáveis de instância são privadas, e fornecendo métodos de acesso para essas variáveis:

```
class Ponto extends Object {
private double x;    /* variável de instância */
private double y;  /* variável de instância */

Ponto(){  /* construtor inicializado a zero */
        x = 0.0;
        y = 0.0;
}
/* construtor inicializado para valores específicos */
Ponto(double x, double y){
        this.x = x;
        this.y = y;
}
public void setX(double x){ /* método de acesso */
        this.x = x;
```

```
}
public void setY(double y){ /* método de acesso */
        this.y = y;
}
public double getX(){ /* método de acesso */
        return x;
}
public double getY(){ /* método de acesso */
        return y;
}
}
```

As declarações deste método mostram como a classe *Ponto* fornece acesso para suas variáveis externas. Outro objeto que precise manipular as variáveis de instância dos objetos *Ponto* deve agora fazer isto via métodos de acesso:

```
Ponto myPonto;  // declara uma variável para referir-se a um objeto Ponto
myPonto = new Ponto(); // aloca uma instância de um objeto Ponto
myPonto.setX(10.0);      // fixa a variável x via método de acesso
myPonto.setY(25.7);
```

Fazendo as variáveis de instância públicas exibe-se alguns dos detalhes da implementação da classe.

2.8.5 Finalizadores

Pode-se declarar um finalizador opcional que realizará necessariamente ações de demolir quando o coletor de lixo estiver em vias de liberar um objeto [SUN 95b]. O fragmento de código ilustra um método finalizador numa classe.

```
/**
fecha a stream quando o lixo é coletado.
*/
protected void finalize(){
        try {
        file.close();
        } catch (Exception e) {
                }
}
```

O método *finalize()* pode ser invocado quando o objeto está sendo coletado pelo coletor de lixo [SUN 95b].

2.8.6 "*Subclassing*"

"*Subclassing*" é um mecanismo pelo qual os objetos novos podem ser realçados e definidos em termos de objetos existentes [SUN 95b]. Dá-se um exemplo de "*subclassing*" uma variante da classe *Ponto* dos exemplos anteriores, para criar um novo *Ponto* tridimensional chamado *PontoT*:

```
class Ponto extends Object {
        protected double x;  /* variável de instância */
        protected double y;  /* variável de instância */

Ponto(){  /* construtor inicializado a zero */
        x = 0.0;
        y = 0.0;
        }
}

class PontoT extends Ponto {
        protected double z; /* coordenada z do ponto */
PontoT(){    /* construtor por omissão */
        x = 0.0;  /* inicialização de variáveis */
        y = 0.0;
        z = 0.0;
        }
PontoT(double x, double y, double z){ /* construtor especifico */
        this.x = x;    /* inicialização das variáveis */
        this.y = y;
        this.z = z;
        }
}
```

Nota-se que *PontoT()* agrega uma nova variável de instância para a coordenada *z* do ponto. As variáveis de instância *x* e *y* são herdadas da classe *Ponto* original, assim, não precisa-se declará-las em *PontoT()* .

A "*subclassing*" permite usar código existente que já foi desenvolvido , e muito mais importante, testado, para casos genéricos. Assim, a "*subclassing*" ganha reuso de código existente. O sistema de execução Java fornece várias bibliotecas de funções úteis que são testadas e são também gravadas no processo [SUN 95b].

2.8.7 Controle de acesso

Quando declara-se uma nova classe em Java, pode-se indicar o nível de acesso permitido a suas variáveis de instância e métodos. Java fornece quatro níveis de acessos

específicos. Três dos níveis devem ser explicitamente especificados quando deseja-se usá-los. Estes são *public, protected* e *private.*

O quarto nível não tem nome, freqüentemente é chamado de *"friendly"*. O nível de acesso *"friendly"* (amigável) indica que as variáveis de instância e métodos são acessíveis para todos os objetos dentro do mesmo pacote, mas inacessível para objetos fora do pacote [SUN 95b].

O nível de acesso *"friendly"* é muito útil quando cria-se pacotes de classes que estão relacionados com cada um dos outros e pode-se acessar cada variável de instância dos outros diretamente. Por exemplo, todo o código do sistema I/O de Java é coletado dentro de um pacote simples. O beneficio primário dos pacotes está na organização de definições de muitas classes dentro de uma unidade simples. O benefício secundário do ponto de vista dos programadores é que as variáveis de instância *"friendly"* e os métodos estão disponíveis para todas as classes dentro de um mesmo pacote, mas não para as classes definidas fora do pacote [SUN 95b].

Os métodos e variáveis de instância *public* estão disponíveis para qualquer outra classe em qualquer lugar. *Protected* significa que as variáveis de instância e métodos projetados deste modo são acessíveis somente pelas "*subclassing*" da classe. Os métodos e variáveis de instância *private* são acessíveis somente dentro da classe na qual foram declarados, e não estão disponíveis para suas "*subclassing*" [SUN 95b].

2.8.8 Variáveis de classe e métodos de classe

Java segue convenções de outras linguagens orientadas a objetos fornecendo métodos de classes e variáveis de classes. Normalmente, as variáveis declaradas em uma definição de classe são variáveis de instância, existe uma dessas variáveis em cada objeto separado que é criado (instanciado) da classe. A variável de classe, por outro lado, é local para a mesma classe, existe somente uma cópia simples da variável e esta é compartilhada por cada objeto instanciado da classe [SUN 95b]. Para declarar variáveis de classe e métodos de classe, declara-se como *static*. O código do fragmento seguinte ilustra a declaração de variáveis de classe:

```
class Retangulo extends Object {
        static final int versao =7;
        static final int revisao = 0;
}
```

A classe *Retangulo* declara duas variáveis estáticas para definir a versão e o nível da revisão desta classe. Agora, cada instância do Retângulo criada desta classe compartilhará essas mesmas variáveis. Note, também que estão definidas como *final* porque deseja-se que sejam constantes.

Os métodos de classes são métodos que são comuns para uma classe completa. Quando usa-se métodos de classe? Freqüentemente, quando tem-se um comportamento

que é comum a cada objeto da classe. Os métodos de classe podem operar somente sobre as variáveis de classe. Os métodos de classe não podem acessar variáveis de instância, ou não podem invocar métodos instanciados. Como as variáveis de classe, declaram-se os métodos de classe definindo-los como estáticos (*static*) [SUN 95b].

2.8.9 Métodos abstratos

Os métodos abstratos são construções potentes no paradigma orientado a objetos [SUN 95b]. Para entender métodos abstratos, temos que verificar a noção de superclasse abstrata. Uma superclasse abstrata é uma classe na qual definem-se métodos que não estão realmente implementados por essa classe, as classes subseqüentes devem prover suas implementações atuais [SUN 95].

No mundo dos objetos, suponha que é criada uma aplicação de desenho. Esta aplicação pode projetar retângulos, linhas, círculos, polígonos, e assim por diante. Além disto, tem-se uma série de operações que podem ser realizadas sobre as formas: reformas, rotações, encher de cor, e assim por diante. Define-se cada uma dessas formas gráficas como uma classe separada; tem-se uma classe retângulo, uma classe linha, e assim por diante. Cada classe precisa de variáveis de instância para definir sua posição, tamanho, cor, rotação e assim por diante, a qual por sua vez invoca métodos para fixar e obter suas variáveis.

Neste ponto, pode-se colecionar todas as variáveis de instância dentro de uma superclasse abstrata simples chamada Gráfico, e implementar mais métodos para manipular estas variáveis. O esqueleto da superclasse abstrata deve parecer-se com o modelo apresentado abaixo:

```
abstract class Grafico extends Object {
        protected Ponto lowerLeft;
        protected Ponto upperRight;
                . . .
        mais variáveis de instância
                . . .
public void setPosition(Ponto 11, Ponto ur){
        lowerLeft = 11;
        upperRight = ur;
        }
abstract void desenhMy();   // método abstrato
}
```

Agora pode-se instanciar a classe Gráfico, porque esta foi declarada abstrata. Pode-se instanciar somente uma subclasse dela. Quando implementa-se a classe *Retângulo* ou a classe circulo, estende-se Gráfico (subclasse). Dentro de Retângulo, fornece-se uma implementação concreta do método *desenhMy*() que desenha um retângulo, uma vez que a definição de *desenhMy*() deve ser necessariamente única para

cada forma herdada da classe *Grafico*. Olhemos um pequeno fragmento da declaração da classe *Retangulo*:

```
abstract class Retangulo extends Grafico {
        void desenhMy() {
        moveTo(lowerLeft.x, lowerLeft.y);
        lineTo(upperRight.x, lowerLeft.y);
        lineTo(upperRight.x, upperRight.y);
        lineTo(lowerLeft.x, upperRight.y);
        . . .
        }
}
```

Nota-se que, na declaração da classe *Grafico*, o método *setPosition()* foi declarado como um método regular *(public void)*. Todos os métodos que podem ser implementados pelas superclasses abstratas podem ser declarados neste momento e todas suas implementações definidas. Então, cada subclasse da superclasse abstrata também pode herdar esses métodos.

2.9 Java é de arquitetura neutra, portátil e robusta

Com o grande crescimento das redes, os desenvolvedores de software devem pensar em distribuí-los. Aplicações, ou até mesmo parte de aplicações, devem ser capazes de migrar facilmente para uma ampla variedade de sistemas de computação, uma ampla variedade de arquiteturas de hardware, e uma ampla variedade de arquiteturas de sistemas operativos. Tudo isto deve operar com uma amigável interface de uso gráfico [SUN 95b].

Claramente, as aplicações devem ser capazes de executar em qualquer lugar da rede sem prévio conhecimento do hardware original e plataformas do software. Os desenvolvedores de aplicações são forçados a desenvolver para plataformas específicas, fazendo com que o problema da distribuição binária cresça rapidamente. Vários métodos tem sido empregados para derrotar o problema, tais como criar "*fat*" binários que se adaptam a arquiteturas de hardware específico, mas tais métodos além de mal feitos ainda são encaixados para um sistema de operação específica. Para resolver o problema da distribuição binária, os softwares de aplicações e fragmentos de aplicações devem ser de arquitetura neutra e portáveis [SUN 95b].

Credibilidade é também um alto prêmio no mundo da distribuição. Código de qualquer lugar da rede deve trabalhar robustamente com baixas probabilidades de criar "colisões" em aplicações que importam fragmentos de código [SUN 95b].

2.9.1 Arquitetura Neutra

A solução que o sistema Java adota para resolver o problema da distribuição binária é um *formato de código binário*, que é independente da arquitetura do hardware, das interfaces do sistema operativo e do sistema windows. O formato deste código binário, independente do sistema, é de arquitetura neutra. Se o sistema Java em tempo de execução é acessado sobre um hardware e plataforma de software dado, uma aplicação escrita em Java pode então executar-se sobre tal plataforma sem a necessidade de realizar qualquer trabalho de portabilidade para esta aplicação [SUN 95b].

2.9.2 Bytecodes

O compilador Java não gera "código de máquina" no sentido das instruções do hardware nativo, gera bytecodes: *código de máquina independente, de alto nível, e para uma máquina virtual que é implementada pelo interpretador Java e o sistema em tempo de execução* [SUN 95b].

Um dos primeiros exemplos de aproximação aos bytecodes foi o UCSD P-System, o qual foi levado para uma variedade de arquiteturas de oito-bit a mediados de 1970 e inícios de 1980, e desfrutou de grande popularidade durante o auge das máquinas de oito-bit. Até os presentes dias, as arquiteturas correntes têm o poder de suportar a aproximação dos bytecodes para software distribuído. Os bytecodes Java foram projetados para serem de fácil interpretação sobre qualquer máquina, ou para tradução dinâmica dentro do código de máquina original [SUN 95b].

A aproximação à arquitetura neutra é útil não só para aplicações baseadas em redes, mas também para distribuição de software em sistemas simples [SUN 95b]. No mercado de software atual, os desenvolvedores de aplicações têm que produzir versões de suas aplicações que sejam compatíveis com IBM PC, Apple Macintosh, e com as diferentes estações de trabalho e arquiteturas de sistemas operativos UNIX no mercado.

Com o mercado PC (através de Windows 95 e Windows NT) diversificado para muitas arquiteturas de CPU, e para PowerPc, a produção de software que rodem sobre todas estas plataformas chega a ser quase impossível. Usando Java, associada com a Abstract Window toolkit, a mesma versão das aplicações pode rodar sobre todas estas plataformas.

2.9.3 Portátil

O principal beneficio da aproximação dos bytecodes interpretados é que os programas compilados na linguagem Java são portáveis para qualquer sistema no qual o

interpretador Java e o sistema em tempo de execução tenham sido implementados [SUN 95b].

Os aspectos da arquitetura neutra discutida acima é o maior passo tomado em direção de ser portável. C e C++ tem como defeito a fato de possuírem muitos tipos de dados fundamentais como "dependentes da implementação" [SUN 95b]. A tarefa do programador é assegurar que os programas sejam portáveis através das arquiteturas por programação [SUN 95b].

Java elimina estas tarefas definindo comportamento padrão que aplica-se aos tipos de dados através de todas as plataformas, especifica os tamanhos de todos estes tipos de dados primitivos e o comportamento aritmético sobre eles. Os tipos de dados são padrões através de todas as implementações de Java [SUN 95b].

O ambiente Java assim mesmo é portável para novas arquiteturas e sistemas operativos. O compilador Java é escrito em Java. O sistema em tempo de execução Java é escrito em ANSI C. Não existem notas de "implementações dependentes" na especificação da linguagem Java [SUN 95b].

2.9.4 Robusta

Java pretende desenvolver softwares robustos, altamente confiáveis e seguros, numa variedade de formas. Existe forte ênfase sobre verificação primária para possíveis problemas, também como verificação dinâmica, para eliminar situações propensas a erro [SUN 95b].

2.9.4.1 Tempo de compilação e verificação em tempo de execução

O compilador Java emprega uma ampla e rigurosa verificação em tempo de compilação, assim os erros relacionados à sintaxe podem ser detectados inicialmente [SUN 95b].

Uma das vantagens de uma linguagem fortemente tipada (como C++) é que isto permite uma verificação ampla em tempo de compilação, assim os bugs podem ser encontrados primeiramente. Desafortunadamente, C++ e C são relativamente descuidados, mais notavelmente na área de declarações de métodos ou funções. Java impõe muitos mais requerimentos sobre o desenvolvedor: Java requer declarações explícitas e não suporta declarações implícitas ao estilo C [SUN 95b].

Muitas das rigorosa verificações em tempo de compilação, no nível do compilador Java, são realizadas em tempo de execução, para verificar consistência em tempo de execução e para fornecer grande flexibilidade. As ligações entendem os tipos

de sistemas e repetem muitas das verificações feitas pelo compilador, para guardar versões com problemas de erro [SUN 95b].

A maior diferença entre Java e C ou C++ é que o modelo de memória de Java elimina a possibilidade de sobrescrever a memória e corromper os dados. Em vez de ponteiros aritméticos, Java têm arrays e strings, isto significa que o interpretador pode verificar arrays e strings indexadas [SUN 95b].

Apesar de que Java não pretende remover integralmente o problema de segurança na qualidade do software, a remoção total de classes de erros de programação, facilita consideravelmente a tarefa de testar e assegurar qualidade [SUN 95b].

2.10 Java é interpretada e dinâmica

Os programadores usam software "tradicional" para desenvolver ferramentas, abandonando o estilo artificial de: edição, compilação, link, carga, e assim por diante, como estilo corrente de prática no desenvolvimento. Adicionalmente, guardando o caminho que deve ser recompilado quando uma declaração muda em algum lugar além de esticar as capacidades das ferramentas de desenvolvimento, até mesmo ferramentas de fantasia estilo "*make*" tais como as encontradas em sistemas UNIX. Este desenvolvimento de aplicações aproxima o código base a centenas de linhas.

Melhores métodos de rapidez e protótipos ousados são necessários no desenvolvimento. O ambiente da linguagem Java é uma das melhores formas uma vez que esta é interpretada e dinâmica [SUN 95b].

Como discutido nas primeiras seções sobre neutralidade da arquitetura, o compilador Java gera bytecodes para a Máquina Virtual Java (JVM). A noção de uma máquina virtual interpretada não é nova. Mas a linguagem Java traz os conceitos dentro do domínio de segurança, distribuição e sistemas baseados em redes [SUN 95b].

A máquina virtual da linguagem Java é estritamente definida como máquina virtual para a qual um interpretador deve estar disponível para cada arquitetura de hardware e sistema operativo sobre o qual deseja-se rodar aplicações da linguagem Java [SUN 95b]. Uma vez tendo o interpretador da linguagem Java e suporte em tempo de execução disponível sobre um hardware dado e uma plataforma de um sistema operativo, pode-se rodar qualquer aplicação da linguagem Java de qualquer lugar, sempre supondo que uma aplicação especifica da linguagem Java é escrita de maneira portável [SUN 95b].

A noção de um "*link*" separado, fase depois da compilação está ausente do ambiente Java. A ligação, o qual é realmente o processo de carregar novas classes pelo carregador de classes, é mais um processo de incremento [SUN 95b].

2.10.1 Carga dinâmica e obrigatória

A portabilidade da linguagem Java e sua natureza interpretada que cria um sistema altamente dinâmico e dinamicamente extensível. A linguagem Java foi projetada para adaptar-se e evoluir em determinados ambientes. As classes são ligadas tão pronto como sejam requeridas e podem ser baixadas através da rede. O código que está sendo enviado é verificado antes de ser passado para o interpretador para sua execução [SUN 95b].

A programação orientada a objetos, tem sido aceitada para resolver ao menos parte da "crises do software", ajudando o encapsulamento de dados e procedimentos correspondentes, incentivando o reuso de código [SUN 95b]. Muitos programadores fazem desenvolvimento orientado a objetos, adotando C++ como sua linguagem de eleição. Mas C++ sofre de sérios problemas que impedem seu uso generalizado na produção e distribuição de software. Este defeito é conhecido como o problema da superclasse frágil.

2.10.2 O problema da superclasse frágil

Este problema surge como um efeito colateral da forma como C++ é usualmente implementada. Cada vez que agrega-se novos métodos ou novas variáveis de instância para uma classe, alguma ou todas as classes que referenciam essa classe requerem uma recompilação. Guardando o caminho das dependências entre as definições de classe e seus clientes tem provado ser uma grande fonte de erros de programação em C++, até mesmo com a ajuda das ferramentas como "*make*". O resultado da superclasse frágil é algumas vezes referenciada como o "problema de recompilação constante" [SUN 95b]. Pode-se evitar esses problemas em C++, mas com extraordinária dificuldade, não fazendo uso de alguma das características da linguagem orientada a objetos diretamente [SUN 95b].

2.10.3 Resolvendo o problema da superclasse frágil

A linguagem Java resolve o problema da superclasse frágil em várias etapas. O compilador Java não compila referências para valores numéricos, em vez disso, passa informação via referências simbólicas através do verificador de bytecodes e do interpretador . O interpretador Java realiza a resolução de um nome final, quando as classes estão sendo ligadas. Uma vez que o nome é resolvido, a referência é escrita em um formato numérico, possibilitando ao interpretador Java rodar a toda velocidade [SUN 95b].

Finalmente, o arranjo de armazenamento dos objetos não é determinado pelo compilador. O arranjo dos objetos na memória é retardado para o tempo de execução e determinado pelo interpretador. A atualização de classes com novas variáveis de instância ou métodos podem ser ligados sem afetar o código existente [SUN 95b].

O baixo custo da procura de um nome permite que à primeira vez qualquer nome seja encontrado, assim a linguagem Java elimina o problema da classe frágil [SUN 95b]. Os programadores de Java podem usar técnicas de programação orientada a objetos de uma forma direta sem a constante sobrecarga da recompilação causada pelo C++ . As bibliotecas podem ser amigáveis para agregar novos métodos e variáveis de instância sem algum efeito sobre seus clientes. A vida do programador é simplificada [SUN 95b].

2.10.4 Interfaces da linguagem Java

A interface na linguagem Java é simplesmente uma especificação de métodos que implementam um objeto. O conceito de uma interface na linguagem Java foi emprestado do Objective C. Uma interface não inclui variáveis de instância ou código de implementação. Pode-se importar e usar múltiplas interfaces de uma maneira flexível, fornecendo os benefícios de múltipla herança sem as dificuldades de herança criada pela estrutura usual de herança da classe rígida [SUN 95b].

2.10.5 Representações em tempo de execução

As classe na linguagem Java têm representação em tempo de execução. Existe uma classe chamada *Class*, e instâncias que contém as definições da classe em tempo de execução [SUN 95b]. Em programas C ou C++, pode-se controlar um ponteiro para um objeto, mas se não conhece-se o tipo do objeto, não é possível encontrá-lo [SUN 95b].

É possível procurar a definição de uma classe fornecendo um string contendo seu nome. Isto significa que se pode computar os tipos de dados chamados e facilmente ligá-los dinamicamente dentro do sistema de execução [SUN 95b].

2.11 Segurança em Java

Chefiar segurança possui grande valor no uso crescente da Internet para produtos e os variados serviços de distribuição eletrônica de software e conteúdo multimidia como "dinheiro digital" [SUN 95b]. A área de segurança revisada aqui será, a mesma que do compilador Java e o sistema em tempo de execução que limitam a programação de aplicações para criar código maldoso.

O compilador da linguagem Java e o sistema em tempo de execução implementam várias camadas de defesa contra código incorreto. O ambiente inicia com a suposição de que nada é verdadeiro.

2.11.1 Alocação de memória e arranjo

Uma das primeiras linhas de defesa do compilador Java é sua alocação de memória e modelo de referência. As decisões do arranjo da memória não são feitas pelo compilador da linguagem Java, como são em C e C++. O arranjo da memória é decidida em tempo de execução, sendo potencialmente diferente dependendo das características das plataformas de hardware e software sobre as quais o sistema Java se executa [SUN 95b].

Java não têm "ponteiros" no sentido tradicional como C e C++, onde as posições de memória contem os endereços de outras posições de memória. O código compilado Java faz referência à memória via "*handles*" simbólicos que são resolvidos para endereços de memória real em tempo de execução pelo interpretador Java. Os programadores Java, não podem forjar ponteiros para memória, uma vez que a alocação de memória e o modelo de referência são completamente obscuros para os programadores e controlados totalmente pelo sistema básico em tempo de execução [SUN 95b].

A estrutura obrigatória para a memória significa que os programadores não podem inferir o arranjo da memória física de uma classe olhando sua declaração. Removendo o arranjo da memória C e C++ e modelos de ponteiros, a linguagem Java eliminou a habilidade do programador para obter visualmente e produzir ponteiros para memória. Estas características devem ser vistas como benefícios positivos antes que uma restrição sobre os programadores, uma vez que elas conduzem a aplicações mais confiáveis e seguras [SUN 95b].

2.11.2 Processo de Verificação dos bytecodes

O compilador Java assegura que o código fonte Java não viola as regras de segurança, quando uma aplicação tal como o browser HotJava importa fragmentos de código de qualquer lugar, não se conhece realmente se o fragmento de código segue as regras da linguagem Java para segurança. O sistema Java em tempo de execução não valida o código importado, mas sujeitado o mesmo à verificação dos bytecodes [SUN 95b].

O alcance dos testes para verificar se o formato de um fragmento de código é incorreto, é realizado através da submissão de cada fragmento de código a uma prova de um teorema simples [SUN 95b]. As seguintes regras são estabelecidas:

- não falsificar ponteiros,
- não violar restrições de acessos,
- os objetos são acessados tais como eles são.

Java é uma linguagem que é segura, realiza verificação em tempo de execução para gerar código, e estabelece um conjunto base para garantir que as interfaces não sejam violadas [SUN 95b].

2.11.2.1 O verificador de bytecodes

O verificador de bytecodes atravessa os bytecodes, constrói a informação do tipo de estado, e verifica os tipos dos parâmetros para todas as instruções bytecode.

O verificador de bytecodes atua como uma espécie de porteiro; isto assegura que o código passado para o interpretador Java esteja num estado apropriado para ser executado e possa rodar sem receio de pendurar o interpretador Java. O código importado não pode ser executado por nenhum motivo até depois que tenha sido submetido aos testes de verificação [SUN 95b]. Uma vez feita a verificação, importantes propriedades são conhecidas:

- Os tipos dos parâmetros de todas as instruções bytecode são sempre reconhecidas como corretas
- Os objetos acessados de um campo são sempre reconhecidos como legais, privados (*private*), públicos(*public*) ou protegidos(*protected*).

Enquanto toda esta verificação pareça insuportavelmente detalhada, após o verificador de bytecodes ter feito este trabalho o interpretador Java pode proceder, sabendo que o código pode rodar com segurança. Conhecer essas propriedades faz com que o interpretador Java seja muito mais rápido, uma vez que não tem que realizar mais verificações [SUN 95b].

2.11.3 Verificação de segurança no carregador bytecode

Enquanto um programa Java é executado, este pode requerer que uma classe particular ou um conjunto de classes sejam carregadas, possivelmente através da rede. Depois que o código é determinado limpo pelo verificador de bytecodes, a próxima linha de defesa é o carregador bytecodes Java. O ambiente consultado pelo processo de execução rodando bytecodes Java pode ser visualizado como um conjunto de classes particionadas dentro espaços separados. Existe uma identificação do espaço para as classes que chegam do sistema local de arquivos, e um espaço separado e identificado para cada fonte da rede [SUN 95b].

Quando uma classe é importada através da rede esta é colocada dentro do espaço chamado privado associado com sua origem. Quando uma classe referencia outra classe, esta é primeiro procurada no chamado espaço para o sistema local, e logo no chamado espaço das referências de classe. Não existe possibilidade de que uma classe importada possa pôr em perigo outras classes. As classes importadas de diferentes lugares são separadas umas das outras [SUN 95b].

2.11.4 Segurança de Java no pacote para redes

O pacote de redes de Java fornece as interfaces para lidar com vários protocolos de redes (FTP, HTTP, TELNET, ...). Esta é a primeira linha de defesa no nível de interface de rede. O pacote de redes pode ser iniciado com nível de configuração, como:

- Eliminar todos os acessos à rede,
- Permitir acesso à rede somente para hosts dos quais o código foi importado,
- Permitir acesso à rede somente fora do firewall se o código chega de fora,
- Permitir todos os acessos à rede.

2.12 Multiprocessos em Java

Os usuários percebem que o mundo está cheio de múltiplos eventos, todos sucedendo uma mesma vez, e surge a questão de como fazer que os computadores trabalhem da mesma forma [SUN 95b].

Desafortunadamente, escrever essa quantidade de programas com muitas coisas sucedendo de uma vez pode ser mais difícil que escrever no estilo de processo simples e convencional como C e C++. Pode-se escrever aplicações multiprocessos nas linguagens tais como C e C++, mas o nível de dificuldade sobe em magnitude [SUN 95b].

O termo processo seguro significa que dada uma biblioteca de funções implementada, esta pode ser executada por múltiplos processos concorrentes de execução[SUN 95b].

O maior problema com processos programados suportados explicitamente é que nunca pode-se estar seguro de obter as travas que se precisam e soltá-las outra vez no tempo adequado [SUN 95b].

2.12.1 Processos no nível da linguagem Java

Os processos fornecem aos programadores em Java uma ferramenta poderosa para melhorar o desempenho interativo de aplicações gráficas. Quando as aplicações precisam rodar animações e tocar música enquanto rola a página, e baixar um arquivo de texto de um servidor, os multiprocessos são a forma de obter rapidez e concorrência dentro de um espaço de processo simples [SUN 95b].

Os processos são a base fundamental de Java. As bibliotecas de Java fornecem uma classe *Thread* que suporta uma variada coleção de métodos para iniciar um processo, rodar um processo, parar um processo e verificar os estados de um processo [SUN 95b].

O suporte dos processos Java incluem um sofisticado conjunto de sincronizações primitivas baseadas no amplo uso do monitor, e a condição de paradigma variável introduzida vinte anos atrás por C.A.R. Hoare e implementado na montagem da produção no sistema Xerox PARC's Cedar/Mesa. Suportes integrados para processos dentro de uma linguagem facilitam seu uso e sua robustez. Muitos dos estilos de integração dos processos Java foram modelados antes de Cedar e Mesa [SUN 95b].

2.12.2 Sincronização de processos integrados

Java suporta multiprocessos no nível da linguagem (sintática) e via suporte de seu sistema em tempo de execução, e processos de objetos. No nível da linguagem, os métodos dentro de uma classe podem ser declarados sincronizados para não rodar concorrentemente, tais métodos rodam sob os controles do monitor para assegurar que as variáveis permanecem num estado consistente [SUN 95b].

Aqui, dois fragmentos de código para uma demonstração de classificação. Os pontos principais de interesse são os dois métodos *parar* e *iniciaClassf*, os quais compartilham uma variável comum chamada *kicker*:

```
public synchronized void parar(){
        if (kicker != null) {
        kicker.parar();
        kicker = null;
        }
}

private synchronized void iniciaClassf() {
        if (kicker == null || !kicker.isAlive()) {
        kicker = new Thread(this);
        kicker.start(); }
}
```

Os métodos *parar()* e *iniciaClassf()* são declarados para serem sincronizados, eles não podem rodar concorrentemente, a fim a manter um estado consistente na variável compartilhada *kicker*. Quando um método sincronizado é ativado, este adquire um monitor sobre o objeto corrente. O monitor impede a execução de qualquer outro método sincronizado nesse objeto. Quando um método sincronizado retorna qualquer significado, este monitor é liberado. Outros métodos sincronizados dentro do mesmo objeto são agora livres para rodar [SUN 95b].

Ao escrever aplicações Java, deve-se tomar cuidado ao implementar suas classes e métodos para garantir processos seguros, da mesma forma que as bibliotecas de Java em tempo de execução são processos seguros. Para que os objetos sejam processos seguros, qualquer método que possa mudar os valores das variáveis de instância deve ser declarado sincronizado. Isto assegura que só um método pode alterar o estado de um objeto em qualquer momento. Os monitores Java são reiniciantes: um método pode adquirir o mesmo monitor mais de uma vez, e o processamento prossegue [SUN 95b].

2.12.3 Suporte de multiprocessos

Enquanto outros sistemas tem fornecido facilidades para multiprocessos, a construção de suporte para multiprocessos dentro de uma linguagem como Java tem fornecido ao programador ferramentas mais poderosas para facilitar a criação de classes multiprocessos com processos seguros [SUN 95b].

Outros benefícios dos multiprocessos são a melhor receptividade interativa e o comportamento em tempo real. O ambiente Java em tempo de execução *"stand-alone"* mostra um bom comportamento em tempo real [SUN 95b].

2.13 Desempenho e comparação

2.13.1 Desempenho

Java tem sido portado para e rodado sobre uma variedade de plataformas de hardware, executando uma variedade de software de sistemas operativos. Os testes de medida de alguns programas Java simples sobre sistemas de computação correntes tais como estações de trabalho e computadores pessoais de alto desempenho mostram resultados como segue:

novos objetos 119,000 por segundo
new C() (classe com vários métodos) 89,000 por segundo
o.f() (método *f* invocado sobre um objeto o) 590,000 por segundo

o.sf() (método sincronizado *f* invocado sobre um objeto o) 61,500 por segundo

Assim, a criação de um novo objeto requer aproximadamente 8.4 milésimos de segundo (mseg), criar uma nova classe contendo vários métodos consume cerca de 11 mseg, e invocar um método sobre um objeto requer aproximadamente 1.7 mseg [SUN 95b].

Enquanto esses números do desempenho para interpretar bytecodes são realmente mais que adequados para rodar aplicações gráficas interativas de usuário final, podem surgir situações onde é requerido alto desempenho. Em tais casos, os bytecodes podem ser traduzidos dentro de código máquina para uma CPU particular sobre a qual a aplicação está sendo executada. Para os usuários acostumados ao projeto normal de um compilador e carregador dinâmico, isto seria como inserir o gerador de código máquina final no carregador dinâmico [SUN 95b].

O formato dos bytecodes foi projetado tendo em mente um gerador de código de máquina, sendo que o processo atual de gerar código de máquina geralmente é simples. O código produzido é razoavelmente bom: realiza alocação de registro automático e o compilador realiza alguma otimização quando são produzidos os bytecodes. O desempenho dos bytecodes convertidos a código máquina é aproximadamente o mesmo dos códigos C e C++ nativos [SUN 95b].

2.13.2 A linguagem Java comparada

São literalmente centenas de linguagens de programação disponíveis aos desenvolvedores para escrever programas que resolvam problemas em áreas específicas. As linguagens de programação cobrem um amplo campo através das linguagens interpretadas totalmente tais como UNIX shells, awk, TCL, Perl, e assim por diante.

Linguagens no nível dos Shells e TCL, por exemplo, são linguagens de alto nível interpretadas totalmente. Elas lidam com objetos a nível de sistema, onde seus objetos são arquivos e processos. Algumas dessas linguagens são convenientes para o desenvolvimento de protótipos muito rápidos. As linguagens escritas são também altamente portáveis, o inconveniente principal é o desempenho; elas são geralmente muito mais lentas que qualquer código de máquina nativo ou que os bytecodes interpretados [SUN 95b].

No nível intermediário, surgem linguagens como Perl, que compartilham muitas características em comum com Java. Na evolução de Perl, foram adotadas características de orientação a objetos e características de segurança; e ela exibe muitas características em comum com Java, tais como robustez, comportamento dinâmico, arquitetura neutra, e assim por diante [SUN 95b].

No nível mais baixo, tem-se linguagens tais como C e C++, nas quais desenvolvem-se projetos de programação em grande escala que possuem alto desempenho. O alto desempenho chega a ter um custo, que incluem o alto custo de um duvidoso sistema de gerenciamento de memória e o uso de capacidades de

multiprocessos que são difíceis de implementar e usar. E naturalmente quando usa-se C++, tem-se a perene fragilidade das superclasses. O problema da distribuição binária de código compilado é difícil de manejar no contexto de plataformas heterogêneas sobre a Internet [SUN 95b].

O ambiente da linguagem Java cria uma área extremamente atrativa entre alto nível e portabilidade. A linguagem Java ajusta-se em algum lugar deste espaço. Além disso, sendo extremamente simples para programar, de alta portabilidade e de arquitetura neutra, a linguagem Java fornece um nível de desempenho que é totalmente adequado para todas as aplicações [SUN 95b].

2.14 Os maiores benefícios de Java

A linguagem Java é muito dinâmica como Lisp, TCL e SmallTalk que são freqüentemente usadas para protótipos. Uma das razões para seu sucesso é que esta linguagem é muito robusta e pelo gerenciamento de memória [SUN 95b].

Similarmente, os programadores podem ficar relativamente despreocupados acerca do comportamento da memória quando se programa em Java. O sistema coletor de lixo faz com que o trabalho dos programadores seja bastante amplo e fácil [SUN 95b].

Outra razão comum dada às linguagens como Lisp, TCL e SmallTalk é que são boas para protótipos e semanticamente ricas [SUN 95b]

3 Programação Funcional

A operação de um computador convencional está baseada na execução seqüencial de instruções simples e recuperadas de uma maneira simples, apesar do meio de armazenamento. Este "modelo de computação" é padrão e quase universal, e tem tido uma profunda influência sobre a natureza das linguagens de programação, para entender que até mesmo na atualidade, ainda estamos ligados à idéia de programas convencionais de alto nível como seqüência de instruções. Apesar de várias linguagens terem sido desenvolvidas em anos recentes, muitos dos detalhes de baixo nível das arquiteturas das máquinas ainda estão ocultos, possibilitando ao programador concentrar-se sobre o problema em altos níveis de abstração. Por permanecerem essas linguagens convencionais ainda presentes, têm-se um estilo de programação baseado em "receitas" para explicar ao computador "como" é dado um problema para ser resolvido. Conseqüentemente, o programador deve sempre ter em mente "como" o programa é avaliado, para depois produzir a seqüência correta de operações para resolver um problema apresentado. A filosofia através dos processos de programação está portanto em dizer "como"; em outras palavras, está baseada na descrição de soluções para problemas, mais que na descrição dos problemas. Tais linguagens são chamadas freqüentemente de linguagens *imperativos,* para refletir o fato de que cada declaração em um programa é uma receita do que fazer no seguinte passo para resolver um determinado problema [FIE 88].

Apesar do estado corrente das linguagens de programação, entretanto, continua a tendência de sempre ir-se numa direção para fornecer mais e mais formas abstratas de resolver problemas, tentando mudar a simplicidade da programação com rapidez na execução dos programas, utlizando-se uma linguagem ainda longe do modelo de execução de instruções seqüenciais. Parece, portanto, natural e quase inevitável o desenvolvimento em tecnologia das linguagens, o que separa o processo de programação dos princípios fundamentais do modelo computacional [FIE 88].

Desta forma, podemos partir da idéia de um programa como uma receita para computar uma resposta, ao invés de desenvolver a idéia de um programa como uma declaração clara e concisa da resposta, ignorando a forma e a extensão do cômputo da resposta. A filosofia através do processo de programação está baseada mais na especificação abstrata de problemas do que numa descrição de seus métodos de solução [FIE 88].

Amplos esforços têm sido gastos para desenvolver métodos rigorosos para especificar, produzir e verificar software e produtos de hardware. Entretanto, os amplos esforços para alcançar esses objetivos foram restringidos a linguagens convencionais. A proposta natural de Von Neumann tem contribuído para esta falha, visto que a noção de um estado global que pode mudar arbitrariamente em cada passo da computação tem sido provado ser intuitivamente e matematicamente intratável. Esta falha tem feito do software o mais caro componente de muitos sistemas de computação [GLA 84]. O maior custo do software pode ser distribuído nas seguintes fases:

- Especificação;

- Desenho;
- Implementação;
- Testes;
- Operação e manutenção.

Enquanto a última fase é certamente a mais custosa, muito do custo pode ser atribuído a falhas nas primeiras fases do ciclo. Apesar dos métodos de especificação formal que existem, eles não são usados amplamente e há um severo "erro" entre os níveis das linguagens usadas nas fases de especificação, desenho e implementação. Assim, o problema de decidir se um programa particular atende a uma especificação, usualmente envolve testes exaustivos, e inevitavelmente conduz a falhas de software devido a erros não detectados durante a última fase do ciclo. A solução para esses problemas parece enganosa no uso da especificação formal e projeto das linguagens de alto nível, bem como nas provas de correção formal [GLA 84].

Os primeiros passos nesta direção foram tomados pela programação estruturada, e o trabalho nas áreas de especificação formal, verificação de programas e semântica formal das linguagens de programação que ainda continuam até hoje. Os pesquisadores têm concluído que muitos dos problemas originam-se da proposta fundamental à filosofia de Von Neumann, e estão voltando-se para uma linguagem de um novo tipo, assim como à arquitetura dos computadores como uma solução. Uma destas propostas é apresentada pelas linguagens de *programação funcional* e suas arquiteturas relacionadas [GLA 84]. Algumas das requisições que têm sido feitas à a programação funcional são:

- As linguagens funcionais estão mais orientadas a problemas que as linguagens convencionais: A distância da especificação para um programa funcional é muito menor e fácil [GLA 84].
- As linguagens funcionais têm uma base matemática simples, o λ-cálculo que diminui os efeitos laterais e as provas de correção de um programa se tornam mais fáceis [GLA 84].
- Os programas funcionais são geralmente muito menores que os programas convencionais, e assim é muito mais fácil aperfeiçoa-los e mantê-los.
- As linguagens funcionais parecem fornecer uma resposta para o problema de exploração do paralelismo oferecido pelos sistemas multiprocessos [GLA 84].

Cada um destes pontos é matéria de uma extensa pesquisa. Contudo, é claro que a programação funcional continuará sendo parte importante da Ciência da Computação nos anos futuros.

3.1 Porque Programação Funcional?

Imagine a possibilidade de ter sistemas de computação perfeitos: sistemas de hardware e software que são usados amigavelmente, baratos, seguros e rápidos. Imagine também que os programas podem ser especificados de uma forma que além de fácil compreensão, sua correção pode ser facilmente provada. Além disso, o hardware básico

idealmente suporta esses sistemas de software e supercondução em arquiteturas paralelas possibilitando obter uma resposta a nossos problemas de uma maneira rápida e deslumbrante [PLA 93].

Na atualidade, todas as pessoas sempre têm problemas com seus sistemas de computação. Na realidade, dificilmente encontram-se fragmentos de software ou hardware livres de "*bugs*". As pessoas têm apreendido a viver com as "crises do software", e tem concluído que muitos produtos de software são duvidosos e improváveis. Tem-se consumido muitos recursos em novos lançamentos de fragmentos de software no qual os antigos "*bugs*" foram removidos e outros novos foram introduzidos. Os sistemas de hardware são geralmente muito mais seguros que os sistemas de software, mas mais sistemas de hardware surgem projetados com precipitação, e até mesmo processos bem estabelecidos contém erros [PLA 93].

Os sistemas de hardware e software com certeza têm chegado a ser muito complexos. Um bom, projeto precisa de muitas pessoas, e anos de pesquisa e desenvolvimento, enquanto ao mesmo tempo aumenta a pressão para pôr o produto no mercado. Assim, é compreensível que esses sistemas contenham bugs. A vantagem é que o hardware é mais barato e pode ser comprado. Mas o crescimento do poder de processamento automaticamente leva a um crescimento do uso, com um aumento da complexidade do software como resultado. Assim, os computadores nunca são suficientemente rápidos, enquanto a complexidade dos sistemas está crescendo rapidamente [PLA 93]. Os dois problemas que a comunidade da Ciência da Computação tem que resolver são:

- Como podemos baixar o custo, e fazer grandes sistemas de software que continuem seguros e de uso amigável?;
- Como incrementar o poder de processamento a baixo custo?

Os pesquisadores estão procurando soluções para esses problemas: investigando "técnicas de engenharia de software", para resolver os problemas relacionados ao gerenciamento de projetos de software , construção e manutenção de software [PLA 93].

Outra proposta está baseada na idéia que os problemas acima mencionados são problemas fundamentais que não podem ser resolvidos a menos que uma proposta totalmente diferente seja usada [PLA 93].

3.2 O estilo de programação imperativa

Muitos programas de computador são escritos numa linguagem de *Programação Imperativa* na qual os algoritmos são expressados pela seqüência de comandos. Essas linguagens, tais como FORTRAN, C, Algol, COBOL, PL/1 e Pascal, são originalmente deduzidas da arquitetura dos computadores nas quais elas rodam. Essas arquiteturas de computadores, embora diferente em detalhe, são baseadas na mesma arquitetura: a

arquitetura de computador de Von Neumann (Burks et al., 1946). A arquitetura de computador de Von Neumann esta baseada no modelo matemático de computação proposto por *Turing* em 1937: a *Máquina de Turing*.

A grande vantagem deste modelo e a arquitetura correspondente é que tanto o modelo e a arquitetura são muito simples. A arquitetura de Von Neumann consiste de um fragmento de memória que contém informação que pode ser lida e modificada pela unidade central de processamento (CPU). Conceitualmente existem dois tipos de informação: *instruções* do programa na forma de código de máquina (informação que é interpretada pela CPU e que controla a computação no computador) e os *dados* (informação que é manipulada pelo programa). Este simples conceito tem feito possível fazer realizações eficientes em hardware a baixo custo [PLA 93].

No início da era da computação, as linguagens de programação imperativas de "alto nível" foram projetadas para fornecer uma notação para computações na forma de uma máquina independente. Mais tarde, foi reconhecida a importância da computação expressiva e como tais programas são compreensíveis para os humanos. É claro que, sendo possível raciocinar sobre programas, cada instrução de máquina não tem equivalente direto em linguagens de programação de alto nível. Por exemplo, é de conhecimento comum que com o uso das sentenças *GOTO*, as quais são a abstração direta das instruções, ramificações e salto que estão disponíveis sobre qualquer computador, os programas podem ser escritos de tal forma que raciocinar sobre eles é quase impossível (Dijkstra, 1968) [PLA 93].

Acredita-se fortemente que um problema similar é causado pela sentença de atribuição [PLA 93]. Um dos maiores inconvenientes do estilo de programação imperativa é que um programa imperativo consiste de uma seqüência de comandos dos quais o comportamento dinâmico deve ser conhecido para entender como esse programa trabalha. As atribuições causam problemas porque mudam o valor de uma variável. Também por causa dos efeitos colaterais, pode acontecer que as avaliações de algumas expressões em sucessão produzem diferentes respostas. Argumentar sobre as correções de um programa imperativo é portanto muito difícil. Além disso, por causa da seqüência de comandos, os algoritmos são mais seqüenciais que o necessário. Portanto, é muito difícil detectar quais partes de um algoritmo podem e não podem ser executadas concorrentemente. É lamentável, desde que a avaliação concorrente parece ser uma forma natural de incrementar a rapidez de execução [PLA 93].

A conjectura adotada por muitos pesquisadores hoje em dia é que as crises do software e a rapidez com que cresce este problema são inerentes à natureza das linguagens de programação imperativa e modelos básicos de computação. Portanto, outros estilos de programação, tais como *orientação a objetos*, *lógica* e *funcional* estão sendo pesquisados [PLA 93].

3.3 O estilo de programação orientada a objetos

Os conceitos de orientação a objetos tiveram origem com as linguagens de programação, mas só recentemente estão sendo usados como ferramenta de projetos de aplicações. O paradigma de orientação a objetos dá ênfase à modelagem de conceitos ao invés de funções ou dados. Manter estrutura estática e comportamento junto oferece maior estabilidade e fornece uma forma mais natural de modelagem e entendimento do mundo real. No desenvolvimento se software orientado a objetos, os sistemas são especificados em termos de objetos que se comunicam através da troca de mensagens. Segundo [MAR 95], "a essência do desenvolvimento orientado a objetos é a identificação e organização dos conceitos do domínio da aplicação", e "a parte difícil no desenvolvimento de software é a manipulação desta essência devido à complexidade inerente dos problemas". Em uma abordagem de desenvolvimento orientado a objetos somente quando os conceitos estiverem identificados, organizados e entendidos é que detalhes dos dados e operações devem ser tratados.

Os softwares desenvolvidos segundo este paradigma são mais facilmente reusáveis, fácies de estender e manter, e estão sendo utilizados em várias áreas da Ciência da Computação: projeto de sistemas, banco de dados, linguagens e arquitetura de computadores [CAV 94]. Apesar de vastamente utilizado, o conjunto de termos que definem os conceitos de orientação a objetos, não é de consenso entre os diversos autores, existindo diferentes interpretações sobre uma mesma característica [CAV 94].

O paradigma de programação orientada a objetos é um exemplo da busca de simplicidade de escrita e compreensão de programas, oferecendo estruturas de encapsulamento de dados e favorecendo a escrita de grandes sistemas. Já os diversos modelos de programação concorrente, visam apresentar recursos que possibilitem melhorar o desempenho de execução de aplicações, porem, por possuírem estruturas complexas, não são comumente utilizados em sistemas de grande porte [CAV 94].

Os programas escritos segundo este paradigma de orientação a objetos são montados a partir de *classes*. As classes são estruturadas em uma hierarquia de generalização/especialização, modelando conceitos de aplicação. Segundo [CAV 94], a tarefa de programação em uma linguagem orientada a objetos tem as seguintes etapas:

- *concepção do modelo*: a aplicação deve ser modelada em objetos comunicando-se através de mensagens;
- *construção de classes*: para cada grupo idêntico de objetos é abstraída uma classe para descrevê-los. Estas classes podem ser organizadas em níveis hierárquicos;
- *descrever as classes*: programar na linguagem disponível as classes que podem herdar de uma possível "*biblioteca de classes*" propriedades, utilizando o recurso de generalização/especialização em níveis hierárquicos através do mecanismo de herença;
- *composição do programa*: o inicio da execução do programa é a criação do primeiro objeto, caracterizando uma espécie de programa principal. Os demais objetos são enviados direta ou indiretamente a partir desta primeira classe.

O estilo de programação segundo este paradigma introduz características bastante marcantes. A mais imediata delas é a *modularidade* encontrada no conceito de objeto que encapsula dados e funções. Este mesmo encapsulamento induz a uma *visão integrada entre dados e funções de acesso*, garantindo um nível de segurança na manipulação de informações e de autonomia de objetos. Outras características estão fundamentalmente ligadas ao suporte a estruturação de conceitos em hierarquias de generalização/especialização, permitindo a *herença* de propriedades de uma classe a outra [CAV 94].

Segundo [AUG 94], o paradigma de Orientação a Objetos é mais que um modelo de computação, é também uma filosofia de desenvolvimento de sistemas. O desenvolvimento Orientado a Objetos é centrado na identificação de *objetos* e na construção de *classes*, uma hierarquia de *classes* que fatora as propriedades comuns de subclasses e de objetos. Assim, um programa é projetado através da classificação de objetos.

3.4 O estilo de programação funcional

John Backus (1978) destacou que a solução para os problemas do software têm que ser procurada usando uma nova disciplina de programação: o estilo de *programação funcional,* em vez de um estilo imperativo [PLA 93].

Num programa funcional, o resultado de uma função chamada é unicamente determinada pelos valores atuais dos argumentos da função. A atribuição não é usada e assim os efeitos colaterais não existem. O resultado de uma função, após todas as condições, será determinada somente pelos valores de seus argumentos [PLA 93].

3.5 Vantagem das linguagens de programação funcional

O estilo de programação funcional é importante, assim como os efeitos colaterais, portanto, as sentenças de atribuição devem ser abandonadas. Mas por que usar uma linguagem de programação funcional?. Não é possível usar uma linguagem familiar?. As linguagens de programação imperativas comuns também têm funções, assim por que não restringi-las e usar somente um subconjunto funcional de, por exemplo C, Algol ou Modula2? [PLA 93].

Podemos usar um subconjunto funcional das linguagens imperativas (isto é, somente usar funções e deixar de fora a atribuição), mas então é despojado da potência expressiva da nova geração de linguagens funcionais que tratam funções como "primeira classe". Em muitas linguagens imperativas as funções só podem ser usadas restritivamente. Uma função arbitrária não pode ser passada como um argumento para uma função nem ser produzida como resultado [PLA 93].

As linguagens de programação funcional têm a vantagem de oferecerem um *uso geral das funções*, o que não está disponível nas linguagens imperativas clássicas. Este é um fato de vida, não um problema fundamental. O tratamento restritivo das funções em linguagens imperativas é devido ao fato que, quando essas linguagens foram projetadas simplesmente não se sabia como implementar o uso geral das funções de uma forma eficiente. Não é fácil mudar as linguagens imperativas depois. Por exemplo, os sistemas dessas linguagens não são projetadas para tratar esses tipos de funções. E também os compiladores têm que ser mudados dramaticamente [PLA 93].

Outra vantagem é que nas modernas linguagens de programação funcional (LsPF) *o estilo de programação funcional* é garantido: a sentença de atribuição simples não está disponível (como GOTO está disponível nas modernas linguagens imperativas). As linguagens de programação funcional nas quais não há efeitos colaterais ou características imperativas de qualquer tipo são chamadas linguagens funcionais *puras*. Exemplos de linguagens funcionais puras são Miranda, LML(Augustsson, 1984), HOPE (Burstall et al., 1980), Haskel (Hudak et al., 1992) and Concurrent Clean (Nöcker et al., 1991b). LISP (McCarthy, 1960) e ML (Harper et al., 1986) são exemplos de linguagens funcionais bem conhecidas, que são impuras [PLA 93].

Nas LsPF puras o programador pode somente definir funções que calculam valores, unicamente determinados pelos valores de seus argumentos. A sentença de atribuição não está disponível e nem o uso pesado das noções de programação de uma variável como alguma coisa que contém um valor que é alterado de tempo em tempo por uma atribuição. As *variáveis* que existem nas linguagens puramente funcionais são usadas em matemática para chamar ou referir-se a um *valor constante desconhecido*. Este valor pode nunca ser alterado. Num estilo funcional, um cálculo desejado é expressado de maneira estática em vez de uma maneira dinâmica. Devido à ausência de efeitos colaterais, as provas de correção dos programas são mais fácies que nas linguagens imperativas. As funções podem ser avaliadas em qualquer ordem, o que faz as LsPF adequadas para avaliação paralela. Além disto, a ausência garantida de efeitos colaterais possibilita certos tipos de análises de um programa [PLA 93].

Em adição à disponibilidade total de funções, a nova geração de linguagens funcionais também oferece uma elegante noção de uso amigável. Os padrões de proteção que fornecem ao usuário um acesso simples a estruturas de dados complexos, basicamente não se tem preocupação do gerenciamento da memória. O acesso incorreto à estruturas de dados é impossível. Os programas funcionais são em geral muito mais curtos que seus correspondentes convencionais imperativos, e assim, em principio é mais fácil aperfeiçoá-los e mantê-los [PLA 93].

Aparecem as perguntas, é possível expressar qualquer possível cálculo usando somente funções puras? Felizmente, esta pergunta já tem resposta há anos. O conceito de uma função é uma noção fundamental em matemáticas. Uma das grandes vantagens das linguagens funcionais é que elas estão baseadas sobre um modelo matemático bem compreendido, o *λ-cálculo* (Church, 1932, 1933). O *λ-cálculo* foi introduzido na mesma época que o modelo de *Turing* (Turing, 1937). Na *Tese de Church* (Church, 1936), as funções que intuitivamente podem ser computadas, são iguais às classes de funções que podem ser definidas no *λ-cálculo*. Turing formalizou a computabilidade da máquina, e

mostrou que os resultados da computabilidade de *Turing* são equivalentes a "*λ-definability*". Assim, o poder dos dois modelos é o mesmo. Portanto, qualquer computação pode ser expressada usando somente o estilo funcional [PLA 93].

3.6 Desvantagens das linguagens de programação funcional

As vantagens mencionadas anteriormente são muito importantes. Mas, apesar das linguagens funcionais estarem sendo usadas mais freqüentemente, em particular como uma linguagem de protótipo rápido e como uma linguagem na qual os estudantes apreendem como programar, elas não são ainda muito comuns para sua utilização na programação de propósito geral [PLA 93]. Os dois principais inconvenientes estão nos campos da:

- eficiência e;
- conveniência para aplicações com natureza imperativa forte.

Primeiramente, os programas escritos numa linguagem funcional rodam muito devagar em comparação com seus correspondentes imperativos. O principal problema é que a arquitetura da máquina tradicional sobre a qual esses programas estão sendo executados não estão projetada para suportar linguagens funcionais. Sobre essas arquiteturas as chamadas às funções são relativamente caras, em particular quando o esquema de avaliação lazy é usado. Outro grande problema é causado pelo fato de que, em alguns algoritmos, devido à ausência de atualização destrutiva, a complexidade de espaço e tempo podem serem muito pior para um programa funcional que para seu equivalente imperativo. Em tal caso, é necessário reter a eficiência na transformação do programa. Assim, novas técnicas de compilação têm sido desenvolvidas em vários institutos de pesquisa [PLA 93].

Em segundo lugar, um importante inconveniente das linguagens funcionais é que alguns algoritmos não podem ser expressados de uma forma elegante num estilo de programação funcional. Em particular, elas parecem conter aplicações que interagem fortemente com o ambiente (programas interativos, banco de dados, sistemas operativos, processos de controle). Mas, o problema é causado, em grande parte, pelo fato de que a arte de programação funcional ainda está em desenvolvimento. Ainda temos que aprender como expressar elegantemente os diferentes tipos de aplicações num estilo funcional [PLA 93].

As vantagens de um estilo de programação funcional são muito importantes para o desenvolvimento de software seguro. As desvantagens podem ser reduzidas a um nível aceitável. Portanto, acredita-se fortemente que um dia as linguagens funcionais podem ser usadas na World Wide Web como uma linguagem de programação de propósito geral.

3.7 Funções matemáticas

Antes de discutir os conceitos básicos das linguagens funcionais, deve-se lembrar o conceito matemático de *função*. Em matemática uma **função** é uma aplicação dos objetos de um conjunto chamado de **domínio** para objetos de um conjunto chamado de **co-domínio** como mostrado na figura 3.1.

Esta aplicação não precisa ser definida para todos os objetos no domínio. Se a aplicação é definida para um objeto, este objeto no co-domínio é chamado de **imagem** do objeto correspondente no domínio. Se todos os objetos no domínio têm uma imagem, a função é chamada de **função total**, de outro modo é chamada de **função parcial**. Se x é um objeto do domínio A e f é uma função definida sobre o domínio A, a imagem de x é chamada $f(x)$ [PLA 93].

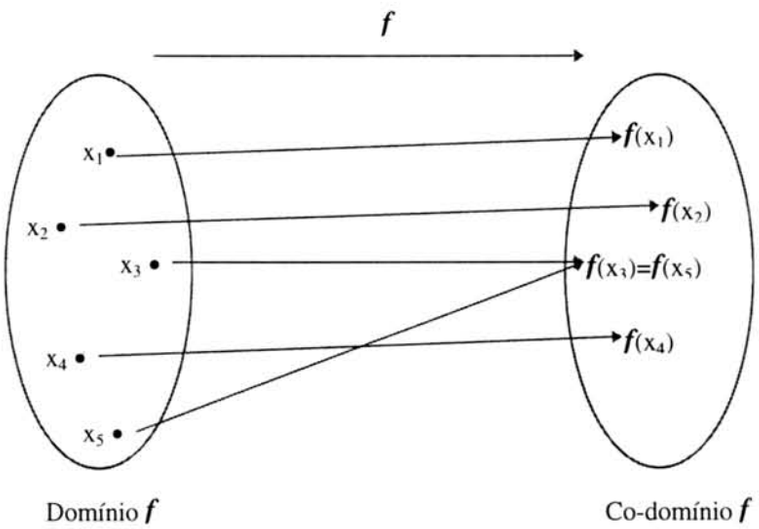

FIGURA 3.1 - Definição de função

O **tipo de uma função** é definida como segue. Se x é um objeto no domínio A, x é chamada de *tipo* de A . Se y é um objeto no co-domínio B, y é chamada de *tipo* de B. Se uma função f aplica objetos do domínio A para o co-domínio B, f é chamada de *tipo* de $A \rightarrow B$. O tipo de f é em geral especificado como:

$$f: A \rightarrow B$$

Em matemática, há várias formas de definir uma função. O tipo de uma função pode ser especificado separadamente da definição da função.

Uma forma de definir uma função é por *enumeração* explicita de todos os objetos no domínio sobre o qual a função está definida, com suas correspondentes imagens [PLA 93]. Um exemplo disto é a seguinte função parcial (domínio **Z** dos inteiros e o co-domínio **N** dos números naturais).

$abs : \mathbf{Z} \rightarrow \mathbf{N}$
$abs(-1) = 1$
$abs(0) = 0$
$abs(1) = 1$

Outra forma de definir funções é usando definições que consistem de uma ou mais *equações* (recursivas). Por exemplo, com este método a função *abs* definida acima, pode ser definida facilmente como uma função total, aplicável para todos os objetos do domínio. Naturalmente, as funções e operadores usados sobre o lado direito devem ser definidos sobre os domínios apropriados.

$abs : \mathbf{Z} \rightarrow \mathbf{N}$
$abs(n) = n, \quad n>0$
$\quad = 0, \quad n = 0$
$\quad = -n \quad n<0$

Uma função como fatorial pode ser definida como segue:

$fac : \mathbf{N} \rightarrow \mathbf{N}$
$fac(0) = 1, \quad n = 0$
$\quad = n*fac(n-1), \quad n > 0$

Outro método de definição alternativa é:

$fac : \mathbf{N} \rightarrow \mathbf{N}$
$fac(0) = 1,$
$fac(n) = n*fac(n-1), \quad n > 0$

Os matemáticos consideram as definições acima muito comuns, definições de funções ordinárias. Mas esses exemplos são também exemplos perfeitos de definições de funções numa linguagem de programação funcional [PLA 93]. A notação de uma definição de função numa linguagem funcional tem muita similaridade com a definição de uma função em matemática, entretanto, há uma importante diferença em objetivos. O objetivo numa linguagem funcional não é apenas definir uma função, mas também é definir uma *computação* que automaticamente compute a imagem (resultado) de uma função quando esta é aplicada a um objeto especifico no seu domínio (o argumento atual da função) [PLA 93].

Algumas definições de funções bem definidas desde um ponto de vista matemático, não podem ser definidas similarmente numa linguagem funcional, devido a que a imagem de algumas funções são muito difíceis de computar ou até mesmo não se podem computar em todo o domínio [PLA 93].

Considerando a seguinte definição de função:

$f : \mathbf{R} \rightarrow \mathbf{R}, \; g : \mathbf{R} \rightarrow \mathbf{R}$
$f'' - 6g' = 6\,sin(x)$
$6g'' + a^2 f' = 6\,cos(x)$

$f(0) = 0,\ f'(0) = 0,\ g(0) = 1,\ g'(0) = 1$

A equação para f, g e suas derivadas f', f'', g' e g'' têm solução, mas não é fácil computar tais funções.

O propósito especial das linguagens de programação é ser capaz de calcular funções aplicando técnicas especiais de cálculo (computação simbólica, usando técnicas de álgebra computacional ou transformação de fórmulas). Mas o propósito geral de uma linguagem funcional é usar um modelo de computação simples, baseado em substituições. Assim, quando as funções são definidas numa Linguagem de Programação Funcional (LPF), uma computação através de substituições é definida implicitamente [PLA 93].

3.8 Um programa Funcional

Um programa escrito numa linguagem funcional consiste de uma coleção de *definições de funções*, escritas numa forma de *equações recursivas*, e uma *expressão inicial* que tem que ser avaliada [PLA 93].

3.8.1 Definição de Funções

A **definição de uma função** consiste de uma ou mais **equações**. Uma **equação** consiste de um lado esquerdo, e um símbolo igual (=) e um lado direito.

lado_esquerdo = *lado_direito*

O lado esquerdo define o **nome da função** e seus **argumentos formais** (também chamados **parâmetros formais).** O lado direito especifica o **resultado da função**, também chamado **corpo da função**. Este corpo da função consiste de uma expressão. Tal expressão pode ser uma denotação de algum *valor*, ou pode ser um *argumento formal* ou uma *função aplicação* [PLA 93].

Numa *função aplicação* a função é aplicada a uma expressão ou *argumento real.* A aplicação de uma função ***f*** para uma expressão ***a*** é denotada como ***fa***. Assim a função aplicação é denotada por uma simples justaposição da função e seus argumentos. Uma importante convenção de sintaxe é que em cada expressão de uma função aplicação há sempre a prioridade mais alta (em ambos lados das equações). A definição de uma função pode ser precedida por seu *tipo de definição* (public, static, private, protected em Java). Na tabela 3.1 são apresentados alguns exemplos de definição de funções (Miranda).

TABELA 3.1- Definição de funções em Miranda

Definição de funções	Comentários
ident :: num → num *ident* x = x	/** *ident* é uma função de num para num, é a função identidade sobre números.
const:: num → num *const* x = c	/** uma função de num para num e alcança uma constante numérica sobre números.
square:: num → num *square* x = x*x	/** função quadrado de um numero.
inc :: num → num *inc* x = x +1	/** função que retorna o valor de seu argumento x +1.

Um argumento formal, tal como ***x*** no exemplo acima, é também chamado **variável.** A palavra variável é usada aqui no sentido matemático da palavra, para não ser confundida com o uso da palavra variável numa linguagem imperativa. A variável não varia, o escopo da variável é limitado à equação na qual ocorre (enquanto que os nomes das funções definidas têm o programa inteiro como escopo) [PLA 93].

As funções definidas como acima, são chamadas **funções definidas pelo usuário**. Pode-se denotar e manipular objetos de certos *tipos pré-definidos* com *operadores pré-definidos*. A tabela 3.2 contém exemplos de *tipos pré-definidos* (numéricos, booleanos, caracteres), correspondendo a *operadores pré-definidos* (funções), denotação de valores (objetos concretos) desses tipos, e o domínio real com o qual eles podem ser comparados.

TABELA 3.2- Tipos Pré-definidos

Tipos	**Operadores**	**Denotação de Valores**	**Comparável com**
numérico	+,-,*, ...	0,1,3.4,1.2E4	números reais
booleano	and, or, ...	true, false	valores de verdade
char	=, <, ...	'a','b', ...	caracter

Quando definimos uma função, é permitido incluir o tipo de informação acerca da função. Em muitas situações temos que definir o valor de uma função por análises de casos. Por exemplo, considerando a função *min*:

$$\begin{aligned} min\ x\ y &= x, \quad Se\ \ x \leq y \\ &= y, \quad Se\ x > y \end{aligned}$$

Esta definição consiste de duas expressões, cada uma das quais é distingüida pelas expressões de valor booleano, chamados de *"guards"*. A expressão de valor booleano é uma expressão que avalia os valores de verdade *"True"* ou *"False"*. A primeira alternativa da definição diz que o valor de (*min x y*) é definido para *x*,

fornecendo a expressão $x \leq y$ avaliada para *True*. A segunda alternativa diz que (*min x y*) é definido para y , fornecendo a expressão $x > y$, esgotando todas possibilidades. Assim, o valor de *min* é definido para todos os números x e y. Outra forma de definir *min* é escrever:

$$\begin{aligned} min\ x\ y &= x, \quad Se\ \ x \leq y \\ &= y, \quad \textbf{\textit{outro lugar}} \end{aligned}$$

A palavra ***"outro lugar"*** pode ser uma abreviatura conveniente para a condição a qual retorna o valor *"True"* enquanto todas as outras retornam o valor *"False"*. Na parte final da notação introduz-se a definição chamada de *local*. Em descrições matemáticas com freqüência encontra-se uma frase da forma **'*onde ...*'** , por exemplo, achar *'f(x,y) = (a +1)(a+2),* ***onde*** *a = (x + y)/2'*. O mesmo esquema pode ser usado numa definição formal:

$$\begin{aligned} f(x, y) &= (a + 1) \times (a + 2) \\ &\quad \textbf{\textit{onde}}\ \ a = (x = y)/2 \end{aligned}$$

Neste exemplo, a palavra especial '***onde***' é usada para introduzir uma definição local cujo contexto (ou escopo) é a expressão do lado direito da definição de *f*.

A definição local pode ser usada em conjunção com a definição por análises de casos, como mostrado abaixo:

$$\begin{aligned} f\ x\ y &= x + a, \quad Se\ \ x > 10 \\ &= x - a, \quad \textbf{\textit{outro lugar}} \\ &\quad \textbf{\textit{onde}}\ \ a = square(\ y + 1) \end{aligned}$$

Nesta definição, a palavra '***onde***' habilita ambas partes do lado direito [BIR 88].

Considerando a seguinte definição de *min* :

$$\begin{aligned} min\ x\ y &= x, \quad Se\ \ x \leq y \\ &= y, \quad Se\ x > y \end{aligned}$$

Note-se que os argumentos de *min* são escritos sem parênteses e sem virgula. Podemos agregar parênteses e escrever:

$$\begin{aligned} min'(\ x, y) &= x, \quad Se\ \ x \leq y \\ &= y, \quad \textbf{\textit{outro lugar}} \end{aligned}$$

As duas funções, *min* e *min'*, estão relacionadas, mas há uma sutil diferença: elas têm tipos diferentes. A função *min*' utiliza um argumento simples, que é um valor estruturado consistindo de um par de números; seus tipos são dados por:

$$min' :: (num, num) \rightarrow num$$

A função *min*, por outro lado, toma dois argumentos um de cada vez. Seu tipo é dado por:

$$min:: num \rightarrow (num \rightarrow num)$$

Em outras palavras, *min* é uma função que toma um número e retorna uma função (de número para número). Para cada valor da expressão x (*min* x) denota a função que toma um argumento y e retorna o mínimo de x e y.

O esquema simples de substituir argumentos estruturados por uma seqüência simples é conhecido como "*currying*" .

É possível definir funções usando modelos no lado esquerdo das equações. Um exemplo simples, envolvendo valores booleanos *"True"* e *"False"*, é dado pelas equações:

$$cond\ True\ x\ y = x$$
$$cond\ False\ x\ y = y$$

Estas equações podem ser escritas em qualquer ordem, desde que os dois modelos *"True"* e *"False"* sejam distintos, e cobrem todos os possíveis valores booleanos. A definição dada de *"cond"* é equivalente a:

$$cond\ p\ x\ y = x,\ \textbf{Se}\ p = True$$
$$= y,\ \textbf{Se}\ p = False$$

Os padrões ou modelos são um estilo equacional de definição, muito mais fácies de entender, assim como simplificam os processos de raciocínio formal acerca das funções.

Os tipos fontes e os tipos alvos de funções não estão restritos em qualquer forma: as funções podem tomar qualquer tipo de valores como argumentos e retornar qualquer tipo de valor como resultado. Em particular, esses valores podem ser eles mesmos funções. Uma função que toma como argumento uma função, ou a entrega como resultado, é chamada de *função de alta ordem*. A idéia é muito simples e não é tão misteriosa. O operador diferencial do cálculo, por exemplo, é uma função de alta ordem a qual toma uma função como argumento e retorna uma função, a derivada, como resultado. A função $\log_b$ para variação de b é outro exemplo. Pode-se definir automaticamente funções de alta ordem, quando se utiliza argumentos "*currying*".

3.8.2 Composição funcional

A composição de duas funções f e g é a função h, tal que $h\ x = f(g(x))$. A composição funcional é denotada pelo operador ($\circ$). Observando o exemplo a seguir:

$$(f \circ g)x = f(g\ x)$$

O tipo de (°) é dado por:

$$(°) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

A composição funcional toma uma função do tipo $(\alpha \rightarrow \beta)$ à direita, e a função de tipo $(\beta \rightarrow \gamma)$ à esquerda, e retorna uma função de tipo $(\alpha \rightarrow \gamma)$. Assim $(°)$ é outro exemplo de uma função polimórfica, que pode assumir diferentes instâncias em diferentes expressões (e dentro da mesma expressão). Somente as restrições sobre o tipo fonte do argumento do lado esquerdo devem concordar com o tipo alvo do argumento do lado direito, o que é expressado no tipo de declaração acima.

A composição funcional

$$(f \circ g) \circ h = f \circ (g \circ h)$$

é uma operação associativa para todas as funções *f*, g e *h*. Então, não é necessário pôr em parênteses quando escreve-se uma seqüência de composições.

Uma vantagem da composição funcional é que algumas definições podem ser escritas de uma forma mais concisa. Por exemplo, uma função definida pelo esquema:

$$soln\ x = função1(\ função2\ (função3\ x))$$

Pode-se escrever de uma forma mais concisa como:

$$soln = função1 \circ função2 \circ função3$$

3.8.3 A expressão inicial

A **expressão inicial** é a expressão cujo valor deve ser calculado [PLA 93]. Por exemplo, se o valor de 2+3 deve ser calculado, a expressão inicial escrita é 2+3.

3.9 Avaliação de um programa funcional

A execução de um programa funcional consiste da avaliação da expressão inicial no contexto das definições das funções no programa, chamado *ambiente*, neste caso usaremos o ambiente da linguagem Java da Sun Microsystem.

- Um programa funcional é um conjunto de funções definidas e uma expressão inicial [PLA 93].

Já que a expressão inicial sempre será de algum tipo, o universo de valores é particionado em coleções organizadas chamadas *tipos*. Os tipos podem ser divididos em dois grupos. O primeiro é o tipo *básico* cujos valores são primitivos. Por exemplo, os números constituem um tipo básico (o tipo numérico), e os valores de verdade (o tipo booleano) e os caracteres (o tipo char). Em segundo lugar, existem tipos compostos (ou derivados), cujos valores são construídos usando outros tipos *básicos*. Exemplos de tipos derivados incluem: (numérico, char), o tipo de par de valores, onde o primeiro componente do par é um número e o segundo componente é um caracter; e [char], o tipo de lista de caracteres. Cada tipo tem associado uma certa operação a qual não é significativa para outros tipos. Por exemplo, não se pode somar um número com um caracter ou multiplicar duas funções.

Um importante princípio da notação é descrever que cada expressão bem formada pode ser atribuída a um tipo, que pode ser deduzido somente de uma expressão. Em outras palavras, como os valores de uma expressão dependem somente dos valores de suas expressões componentes, este princípio é chamado de *fortemente tipado*, tal como é a linguagem Java.

A maior conseqüência da disciplina imposta pelo principio de ser fortemente tipado, é que qualquer expressão não pode ser atribuída a um tipo "sensível" porque é considerada como não bem formada (mal formada), e é rejeitada pelo computador antes de sua avaliação. Tal expressão não tem valor, ela é simplesmente considerada como ilegal [PLA 93].

Existem duas etapas de análises quando uma expressão é submetida para avaliação. A expressão é primeiramente checada para observar se a sintaxe está correta. Este estado é chamado de *análise de sintaxe*. Quando a sintaxe é incorreta o computador sinaliza um erro de sintaxe. Caso contrário a expressão é analisada para observar se possui um tipo "sensível". Esta etapa é chamada de *análise de tipo*. Quando a expressão falha nesta etapa, o computador sinaliza um tipo de erro, somente se a expressão passa em ambas as etapas pode continuar o processo de avaliação.

Ser fortemente tipada é importante para uma linguagem de programação, uma vez que sua adeção a uma disciplina pode ajudar a projetar com clareza programas bem estruturados [BIR 88].

Em programação, a *especificação* é uma descrição matemática das tarefas que um programa têm que realizar, enquanto que a *implementação* é um programa que satisfaz a especificação. As especificações e implementações são um tanto diferentes em natureza e servem diferentes propósitos. A especificação das expressões são intenções dos programadores, e seus propósitos de fazer programação breve, simples e clara; entanto que as implementações são expressões para execução por computador, e seu propósito é ser eficiente o suficiente para serem executadas no tempo e espaço disponível. A ligação entre as duas, é o requerimento que satisfaz a implementação [BIR 88].

4 Interpretador LISP

LISP, uma linguagem de programação desenvolvida nos anos de 1950 por *John McCarthy*, é uma das linguagens de programação mais antigas que se usam na atualidade, é usada principalmente na área da Inteligência Artificial (IA). LISP é uma linguagem extremadamente simples e poderosa que pode ser apreendida com facilidade, até mesmo, como a primeira linguagem de programação. Na verdade, LISP foi confinada para aplicações de Inteligência Artificial, mas agora, é muito mais amplamente usada, devido ao desenvolvimento de mais versões eficientes e flexíveis de LISP que podem ser usadas sobre uma ampla variedade de máquinas [AND 87].

A primeira implementação de LISP foi sobre uma máquina IBM 704 (a mesma, sobre a qual foi feita a primeira implementação do FORTRAN). Um protótipo interativo do sistema LISP foi demostrado em 1960 e foi um dos exemplos mais rápidos de computação interativa. O sistema LISP rapidamente espalho-se para outros computadores, e na atualidade existe sobre todas as máquinas virtuais incluindo microcomputadores. Assim, LISP é a linguagem de programação mais amplamente usada pela Inteligência Artificial (IA) e outras aplicações simbólicas. Uma das razões é a mesma que motivou seu projeto original: habilidade para representar e manipular relações complexas entre dados simbólicos [MAC 87].

Uma das características das aplicações da Inteligência Artificial é os problemas que não estão bem entendidos. Na verdade, com freqüência um objetivo da pesquisa é entender melhor o problema. Esta característica também é aplicada a pesquisas e projetos avançados de desenvolvimento em outras áreas da Inteligência Artificial; portanto, a linguagem LISP e muito apropriada para estes tipos de problemas. Uma das características que serve ao LISP para problemas mau definidos é seu sistema de tipo dinâmico e sua estrutura de dados flexíveis [MAC 87].

Nos últimos anos, as tendências em metodologia de programação e linguagens de programação têm sido a especificação de tipos de dados abstratos (*"abstract data types"*), isto é, o programador decide primeiro qual estrutura de dados e que tipos de dados precisa, e logo especifica as operações necessárias em termos de suas entradas e saídas, e finalmente, os tipos de dados abstratos são implementados. Assim, uma linguagem como LISP, com poucas restrições sobre a invocação de procedimentos e passo de parâmetros, é bem entendida e muito simples para sua experimentação [MAC 87].

A representação de programas LISP como listas LISP, é muito conveniente para manipular programas LISP usando outros programas LISP, na prática, isto significa que, os programadores em LISP têm sido incentivados a escrever muitas ferramentas de programação em LISP, isto inclui programas para transformar programas LISP em outros programas LISP, e para gerar programas LISP de outras notações, também tem simplificado a escrita de programas que processam programas LISP, tais como compiladores e otimizadores. Isto tem incentivado a escrever extensões de propósito especial como processamento de textos, edição e verificação de tipos, e assim por diante. É claro, todas essas ferramentas podem ser (em alguns casos, têm sido) fornecidas para

linguagens convencionais tais como PASCAL. Contudo, têm tendência a serem cumpridas e complicadas devido à complexidade da linguagem convencional, cuja sintaxe é orientada a caracter. Em contraste, a facilidade de um programa LISP para acessar as partes de outro programa é devida a sua sintaxe estruturada e simples. Como resultado, os programadores têm incentivo para experimentar com sofisticadas ferramentas de programação [MAC 87].

A facilidade para manipular programas LISP, tem permitido desenvolver uma grande variedade de ferramentas para programação LISP. Da idéia das bibliotecas de ferramentas foi desenvolvida a idéia de um *ambiente de programação*, este é um sistema que suporta todas as fases da programação, incluindo o projeto, a depuração, a documentação, e a manutenção. Tudo isto, permite o desenvolvimento de ambientes de programação integralmente desenvolvidos em sistemas LISP [MAC 87].

LISP é uma linguagem aplicativa de amplo uso, por isso, a experiência ganhada com LISP tem sido muito valiosa na avaliação de linguagens de programação funcional e estilos de programação funcional. Esta experiência encontra aplicação direta no desenvolvimento de linguagens de programação funcional, devido a que muitas das idéias, incluindo listas e funcionais, têm sido tomadas de LISP pela programação funcional [MAC 87].

No futuro, esperamos o desenvolvimento de novos dialetos LISP, novas linguagens funcionais, novos ambientes de programação, e novas ferramentas de programação LISP como a apresentada neste trabalho.

Portanto, para o propósito deste trabalho, vamos a descrever de uma forma muito geral a linguagem **LISP**, assim como o **Interpretador LISP**, baseado no trabalho de CHAITIN[CHA 96a].

4.1 A linguagem LISP

LISP foi a primeira linguagem funcional pura, e foi projetada por John McCarthy no início de 1960 (McCarthy et al., 1962). Apesar da linguagem LISP original ser pura e referencialmente transparente, os dialetos que foram desenvoltos nos anos seguintes incluem muitas características imperativas, mais notavelmente os construtores para realizar atribuição destrutiva, a qual destrói a simplicidade inerente e a elegância da linguagem original [FIE 88].

Encravado dentro de todos esses dialetos, todavia, existe um subconjunto 'puro', o qual se for considerado de forma isolada pode ser usado para escrever programas funcionais, assim como conhecê-los. Apesar de que as várias implementações de LISP diferem significativamente na eleição dos nomes chaves, nomes das funções primitivas e sobre toda a estrutura do programa, os princípios básicos são os mesmos. A semântica diferente entre dialetos diferentes e as implementações de LISP são mais significativas [FIE 88].

LISP é uma linguagem pouco comum, de matemática formal e de programação convencional. Como uma linguagem de matemática formal, é fundamentada sobre uma parte da lógica matemática conhecida como *Teoria das funções recursivas*. Como uma linguagem de programação, LISP está relacionada primeiramente com o processamento simbólico de dados, antes do processamento numérico [WEI 67].

LISP foi projetada para permitir expressões simbólicas de uma complexidade arbitrária para serem avaliadas por um computador. Para tentar entender seu significado, estrutura, construção e avaliação de expressões simbólicas, é necessário aprender a programar em LISP, e desenvolver gradualmente novos materiais usando materiais antigos ou estender o escopo das definições de conceitos anteriores [WEI 67].

4.2 S-expressões

A palavra **'LISP'** é dada por **'LIS**t **P**rocesing'. Uma característica que a das outras linguagens (linguagens imperativas inclusive), é que somente um tipo de dado composto é suportado, e é chamado de lista. As listas em LISP são sem tipo, assim elas podem conter componentes arbitrárias e são representadas textualmente usando as chamadas *S-expressões* (expressões simbólicas) [WEI 67].

As *S-expressões* são de longitude indefinida e têm uma ramificação da estrutura de uma árvore binário; assim as sub-expressões podem ser rapidamente isoladas [WEI 67]. As *S-expressões* são também um **átomo** ou uma seqüência de outras *S-expressões* separadas por espaços e cercadas por parênteses. O tipo de expressão final é chamada de *S-expressão* **não-atômica.** Um **átomo** pode ser *simbólico* ou *numérico.* Um átomo numérico é uma seqüência de dígitos (possivelmente precedidos por um caracter signo + ou -) e um átomo simbólico é qualquer seqüência de caracteres iniciando com um caracter alfabético. As *S-expressões* não-atômicas podem serem consideradas simplesmente como listas. Por exemplo, uma lista em HOPE:

$$[\, e_1, e_2, \ldots, e_n \,]$$

onde e_i ($1 \leq i \leq n$) são expressões, as quais podem ser expressas em LISP pela *S-expressão*

$$(\, e_1, e_2, \ldots, e_n \,)$$

diferente de HOPE, onde cada uma das e_i devem ser do mesmo tipo, em LISP elas podem ser de tipo arbitrário. Exemplos de *S-expressões* são:

42
Jorge6
(Amigos mios)
((10) (20) (1 2 3 4))
((-6 -20) and (2 mulheres numa boate) as duas são (S expressoes))

Nota-se que os átomos individuais são delimitados por um ou mais caracteres de espaço em *S-expressões* não-atômicas.

Com freqüência refere-se a *S-expressões* não-atômicas desta classe como uma lista, por razões óbvias. Todos os átomos em LISP são assumidos como não sendo decomponíveis, assim, não podemos dividir um átomo simbólico em seus caracteres componentes. Mas podemos então, decompor uma lista aplicando funções primitivas. Em HOPE decompõe-se uma lista usando modelos. Por exemplo, a seguinte função HOPE retorna o primeiro elemento da lista, usando o modelo do lado esquerdo da equação:

```
dec head: list( alpha ) → alpha;
--- head( x:: _ ) <= x;
```

Similarmente,

```
dec tail: list( alpha ) → list( alpha );
--- tail( _::l ) <= l;
```

Em LISP não há modelos, em vez disso, funções como *head* e *tail* dadas acima são fornecidas como funções primitivas da linguagem e são usadas explicitamente para extrair os elementos requeridos das *S-expressões* dadas. Há quatro primitivas associadas com a composição e decomposição de *S-expressões*:

- CAR equivalente a *head*
- CDR equivalente a *tail*
- CONS equivalente a *::* em HOPE
- ATOM confere quando um argumento é um átomo

A primitiva ATOM deve retornar alguma representação de valores booleanos *true* ou *false*, e estes são representados pelos átomos especiais *T* e *F* respectivamente (comparado com HOPE onde os valores de verdade são suportados como tipo base). A decomposição de funções têm nomes bastante curiosos, os quais não possuem relação com as operações que eles representam. Estes nomes se originam nas implementações iniciais do LISP, nas quais o primeiro elemento de uma *S-expressão* não-atômica (a *head*) foi acessada através de um registro de máquina especial chamado *'registro de endereço'* (address register), e a *tail*, através de outro registro especial chamado de *'registro de decremento'* (decrement register). Portanto, a *head* e *tail* de uma *S-expressão* podem então serem acessadas referindo-se ao seu "Conteúdo do Registro de Endereço" (CAR) e ao seu "Conteúdo do Registro de Decremento" (CDR), respectivamente [FIE 88].

Em LISP representa-se a aplicação de uma função f para um conjunto de argumentos $a_1, a_2, \ldots, a_n$ por uma S-expressão simples

$$(f\ a_1\ a_2\ \ldots\ a_n)$$

Isto significa que a aplicação como um todo, e os argumentos para a função são representados como listas (*S-expressões*). Isto deve ser notado, porque mais dialetos de

LISP são definidos para ter semântica *estrita*, significando que os argumentos das expressões $a_1, a_2, \ldots, a_n$ são avaliados antes que a função f seja chamada. Existem exceções, por exemplo, Lispikit LISP, que tem semânticas lazy [FIE 88].

A propriedade indesejável do LISP na representação de funções aplicativas, é que a *S-expressão*

(1 2 3)

não tem o mesmo formato de uma função aplicação; como uma conseqüência disto devemos ler a expressão de acima como a aplicação de *1* para os argumentos da lista (2 3), o qual não é o que se entende. Para solucionar este problema, todas as constantes em LISP devem serem "cotadas". Isto envolve a aplicação da função *QUOTE* para as *S-expressões* (constante). O efeito de QUOTE é retornar seus argumentos sem serem mudados, isto é, sem interpretar isto como uma expressão para ser avaliada. Assim, por exemplo, a *S-expressão* constante

63 (4-7) (Grande Brasil)

pode ser escrita

(QUOTE 63) (QUOTE(4-7)) (QUOTE (Grande Brasil))

respectivamente. A seguir são mostrados alguns exemplos de aplicações de primitivas dadas acima:

Expressão	**Resultado**
(CAR (QUOTE (1 2 3)))	1
(CDR (QUOTE (1 2 3)))	(1 2 3)
(CONS (QUOTE A) (QUOTE (Boa Vida)))	(A Boa Vida)
(ATOM (QUOTE 1))	T
(ATOM (QUOTE (1 2)))	F
(CAR (CDR (QUOTE (CAR CDR QUOTE))))	CDR

Agora, devemos perguntar o que acontece se pegarmos o CDR de uma lista com somente um componente. O resultado obtido é um átomo especial chamado NIL, o qual é a denotação de uma lista vazia. Neste sentido, NIL, denota a mesma lista que o construtor HOPE *nil*, como parte de outra expressão. É claro, usar QUOTE como mostrado acima [FIE 88]. Por exemplo,

Expressão	**Resultado**
(CDR QUOTE (2))	NIL

(CAR (CDR (QUOTE(- 23 NIL NIL (1 4)))))	NIL
(CONS (QUOTE 1) (QUOTE NIL))	(1)
(ATOM (QUOTE NIL))	T

Como todos os programas e dados na linguagem LISP estão na forma de expressões simbólicas usualmente referenciadas como *S-expressõe*, estas ajudam à capacidade de memória disponível num computador para ser usada para armazenar *S-expressões* na forma de uma lista de estruturas. Este tipo de organização de memória livra o programador da necessidade de alocar e armazenar diferentes seções de seu programa ou dados. Também pode-se fazer em LISP programas com dados homogêneos (programas que podem ser tratados como dados e vice-versa por outros programas), o que é uma característica única desta linguagem [WEI 67].

4.3 Condicionais e funções primitivas

Uma expressão condicional em LISP é simplesmente uma *S-expressão* com quatro componentes. O primeiro elemento é o átomo especial **IF**; o segundo é a expressão predicado (a qual retorna **T** ou **F** numa expressão condicional válida), e os dois elementos restantes são ramificações "*verdadeiro*" e "*falso*".

TABELA 4.1- Funções Primitivas

Funções primitivas LISP	
ADD	Adição aritmética.
ATOM	Confere quando um argumento é um átomo.
CAR	Retorna o cabeçalho da lista.
CDR	Retorna o resto da lista.
CONS	Junta cabeçalho e resto para fazer uma nova lista.
EQ	Confere a igualdade.
GTR	Confere maior que.
LESS	Confere menor que.
MIN	Subtração aritmética.
MUL	Multiplicação aritmética.

Por exemplo, a expressão HOPE

if *x = 0* ***then*** 0 ***else*** *x - 1*

é representada em LISP por

(IF (EQ × (QUOTE 0)) (QUOTE 0) (SUB × (QUOTE 1)))

EQ e MIN são exemplos mais distantes de funções primitivas. As aplicações de tais funções são escritas em forma de prefixo, diferente da forma de infixo utilizada em HOPE e Miranda. Isto faz com que a sintaxe da aplicação de uma função primitiva seja idêntica à aplicação de funções definidas pelo usuário. A lista de funções primitivas é dada na tabela 4.1, apesar de que muitas outras podem ser suportadas.

4.4 Definição de Funções

Em HOPE, podemos escrever diretamente uma expressão a qual denota uma função, usando uma expressão lambda . Por exemplo, a função que incrementa qualquer valor de qualquer argumento dado, pode ser escrita como:

lambda $x \Rightarrow x + 1$

Em LISP existe um mecanismo idêntico, sendo que a função é introduzida por um átomo especial LAMBDA além da palavra chave (**lambda**), como em Hope. Uma expressão lambda em LISP é uma *S-expressão* com três componentes: a primeira é o átomo LAMBDA; a segunda é a lista de parâmetros formais chamados pela função; a terceira é o corpo da função. Por exemplo, a função a qual incrementa um valor x dado é escrito como segue:

(LAMBDA (x) (ADD x (QUOTE 1))).

A função máximo é escrita como;

*- - max(m, n) <= **if** m > n **then** m **else** n ;*

em HOPE .Em Lisp é escrita como segue:

(LAMBDA (m n) (IF (GTR m n) m n))

Note-se que as referências para os parâmetros formais da função são escritos usando as variáveis m e n respectivamente, diferentemente dos átomos m e n que têm que ser usados com *QUOTE*. Por exemplo, a função que adiciona o átomo *A* ao inicio de uma lista dada, pode ser escrita

(LAMBDA (A) (CONS (QUOTE A) A))

aqui, observa-se que usar QUOTE tem outro efeito importante, a saber, distingue átomos de variáveis.

Diferente de Hope, em Lisp não há forma na qual as expressões lambda possam ser recursivas, e não há nome para referenciá-las no corpo da função. No entanto, podemos chamar a expressão cercando a expressão lambda dentro de uma expressão chamada *LETREC* (RECusrive LET). Em Hope há dois métodos para escrever uma

função recursiva, ou um conjunto de funções mutuamente recursivas, eles podem declarar uma nova função usando ***dec*** e subseqüentemente, fornecendo uma definição para essa função; ou pode-se usar a expressão recursiva ***let*** ou ***where***. Em LISP todas as funções recursivas são introduzidas usando *LETREC*. Por exemplo, a expressão Hope

let *f* == *E1* ***in*** *E2*

onde *E1* é uma expressão contendo uma referência para *f*, podendo ser escrita em LISP como segue:

(LETREC E2' (f E1'))

onde, *E1'* e *E2'* são as equivalentes em LISP de *E1* e *E2*. Como um exemplo, aqui está a função fatorial em LISP:

```
( LETREC fac
        ( fac ( LAMBDA ( x )
                ( IF ( EQ x ( QUOTE 0 ) )
                        ( QUOTE 1 )
                        ( MUL x ( fac ( MIN x ( QUOTE 1 ) ) ) ) ) ) )
        )
)
```

Esta é realmente a versão LISP da expressão Hope

let *fac* == ***lambda*** *x* ⇒ ***if*** *x = 0* ***then*** *1* ***else*** *x*fac(x - 1)*
in *fac*

pode-se usar *LETREC* para definir um conjunto de funções mutuamente recursivas incluindo as definições dessas funções dentro da mesma *LETREC*. Por exemplo, a expressão, a qual calcula o fatorial de 3, pode ser escrita como:

```
( LETREC ( f ( QUOTE 3 ) )
        ( f ( LAMBDA ( x )
                ( IF ( EQ x ( QUOTE 0 ) )
                        ( QUOTE 1 )
                        ( g x ) ) ) )
                ( g ( LAMBDA ( x )
                        ( MUL x ( f ( MIN x ( QUOTE 1 ) ) ) ) ) ) )
)
```

a qual é análoga a expressão Hope:

let *(f , g)* == *(* ***lambda*** *x* ⇒ ***if*** *x = 0* ***then*** *1* ***else*** *g(x),*
lambda *x* ⇒ *x*f(x - 1))*
in *f(3)*

Pode-se expressar um conjunto de definições não recursivas usando a expressão *LET*, a qual tem o mesmo formato que a expressão *LETREC*. Por exemplo, a expressão Hope

let *(a, b) == (h(x), h(y))*
in *f(a, b) + g(b, a)*

pode ser escrita em LISP como:

```
( LET ( ADD ( f a b ) ( g b a ) )
        ( a ( h x ) )
        ( b ( h y ) )
)
```

4.5 Funções de alta ordem em LISP

Usando *QUOTE* e *LAMBDA* juntas pode-se passar funções (expressões lambda) como parâmetros para outras funções. Tem-se como exemplo a função *map* a qual aplica uma função *f* dada para cada elemento da lista *L*. Um exemplo de aplicação na qual os incrementos da função são passados como parâmetros para *map*, usando QUOTE e definindo expressões lambda, é:

```
( LETREC ( map ( QUOTE ( LAMBDA ( x ) ( ADD x ( QUOTE 1 ) ) )
                        ( QUOTE ( 1 2 3 ) ) )
                ( map ( LAMBDA ( fL )
                        ( IF ( EQ L ( QUOTE NIL ) )
                                ( QUOTE NIL )
                                ( CONS ( f ( CAR L ))(MAP f  ( CDR L )))
                        ) )
                )
)
```

Note-se que, usar QUOTE para as expressões lambda, de outro modo, seria interpretado como uma aplicação da função *LAMBDA* à lista de argumentos *((x)(ADD x (QUOTE 1)))*. O maior problema com o uso de *QUOTE* neste contexto,entretanto, existe quando há referência de variáveis dentro da expressão quotada, a qual não esta "delimitada" pelos parâmetros formais em algumas expressões lambda, então o conflito pode ocorrer quando a expressão lambda é aplicada. Por exemplo, quando escrevemos

```
( LET ( LAMBDA ( x ) ( ADD x n ) )
        ( n ( QUOTE 1 ) )
)
```

é bem claro que *n* na expressão lambda refere-se ao valor *1*. Entretanto, quando quotada esta expressão é passada para outra função, então sua ligação de *n* para *1* pode ser perdida. Por exemplo:

```
( LET ( f ( QUOTE ( LAMBDA ( x ) ( ADD x n ) ) ) )
```

(n (QUOTE 1))
(f (LAMBDA (g) (LET (g n)
(n (QUOTE 4)))))
)

Existem, agora duas possíveis interpretações do resultado, dependendo de como são tratadas as ocorrências de *n* dentro do corpo da expressão quotada lambda, isto é, dependendo de quando *n* é limitada estaticamente ou dinamicamente. Isto está na estratégia de ligações, onde as várias implementações de LISP diferem significativamente. As primeiras implementações de LISP usaram ligações dinâmicas, mas estas tinham erros, e assim, os mais recentes dialetos de LISP (Lispikit LISP) suportam ligações estáticas [FIE 88].

Talvez a mais atrativa característica de LISP seja a simplicidade de sua sintaxe. Um programa LISP é construído por somente oito tipos de expressões: variáveis, constantes, primitivas, condicionais, aplicações, lambdas, lets e letrecs. Valores 'especiais' como *true*, *false* e *nil* estão todos representados como átomos. Além, um programa LISP é justamente uma *S-expressão* [FIE 88]. Assim, LISP quase não tem regras sintáticas, só as que formam as *S-expressões*. Neste sentido LISP é radicalmente diferente de uma linguagem como Hope, cujo BNF cobre várias páginas de texto [FIE 88].

As funções recursivas definidas em LISP usam expressões lambda, formas compostas e expressões condicionais. Portanto, uma grande variedade de classes de funções podem serem definidas usando estes métodos e o método *recursivo* [WEI 67].

Pensar de forma recursiva toma tempo e exije prática, particularmente quando se tem experiência em programação com fluxo linear de controle comum com linguagens algébricas. Uma grande ajuda para pensar de forma recursiva pode ser a heurística, através de exemplos adequados. As funções recursivas podem serem definidas de uma maneira similar a outras funções usando formas de composição. Podemos construir uma forma, tal como;

(CONS X Y)

aqui estamos fazendo uma chamada explícita sobre a função CONS. A função CONS, neste caso, é uma função existente no LISP.

Na definição de uma função recursiva, por exemplo, a função ***f***, estamos fazendo chamadas explícitas sobre funções, mas uma ou mais chamadas são feitas sobre a mesma função. A diferença entre chamadas da função CONS e chamadas da função ***f***, é que ***f*** está sendo descrita. Mas o LISP não entende. Em muitas linguagens algébricas, o programador toma cuidado para não escrever subrotinas que chamam a si mesmas, já que muitas linguagens algébricas não podem manipular a recursividade. Em LISP tudo isso é possível, ao mesmo tempo [FIE 88].

```
(EXEMPLO ( LAMBDA (L) (COND ((NULL) NIL)
                (T  (CONS  (CAR L) ( EXEMPLO (CDR L)))))
```

4.6 Descrição do Interpretador LISP

Este interpretador está baseado na construção de novas definições para uma Máquina de Turing Universal (MTU) semi-limitada, que é fácil de programar e executa rapidamente. Ele fornece novos princípios para a Teoria da Informação Algorítmica (TIA), a qual é a teoria do tamanho em bits dos programas, para Máquinas de Turing Universais (MTU's). Previamente, TIA tem uma qualidade da matemática abstrata. Agora é possível escrever programas executáveis que incorporam as construções nas provas dos teoremas [CHA 96].

Esta nova MTU semi-limitada é implementada via software, escrita numa nova versão do LISP que foi criada especialmente para este propósito. Este LISP foi projetado para escrever um **Interpretador de LISP** em Matemática [CHA 96a], que logo depois será traduzido para a linguagem Java, da Sun Microsystem.

A seguir, vai-se descreve o **Interpretador LISP** para a linguagem Java:

- *ATOM*: pode ser uma palavra, um conjunto de palavras ou inteiros sem signo.
- *// comentário* : Para descrever comentários.

Cada função primitiva de LISP tem um número fixo de argumentos. As funções abaixo são funções fornecidas com o significado usual da linguagem LISP:

- *' is Quote = is EQ*
- *atom :* Um argumento; retorna *t* ou *nil*, quando o argumento avaliado é ou não é um átomo.
- *car :* Um argumento; o qual deve ser avaliado para uma lista; retorna o primeiro elemento desta lista.
- *cdr :* Um argumento; o qual deve ser avaliado numa lista com ao menos um elemento; retorna a lista obtida, removendo o primeiro elemento dessa lista.
- *cadr :* função composta das funções CAR e CDR [MAU 72].
- *caddr:* função composta das funções CAR e CDR [MAU 72].
- *cons :* dois argumentos, deve avaliar uma expressão e uma lista, retorna a lista que consiste de elementos da lista dada, precedida pela expressão como primeiro elemento.

Também temos:

- *lambda :* introduz uma expressão lambda.
- *define :* introduz a definição de uma função.
- *let :* a função *let* permite efetuar cálculos dentro de um ambiente local.
- *if :* condicional if.
- *display :* função que mostra o resultado de uma avaliação.
- *eval :* um argumento; retorna o valor de seu argumento.

Os programadores de LISP escrevem *M-expressões*, nas quais os parênteses para cada função primitiva são implícitos, não sendo assim, para *S-expressões* com parênteses explícitos.

- *nil :* denota uma lista vazia (), e
- *True e False :* são os valores lógicos de verdade.

Para o comportamento com inteiros decimais sem signo temos:

- + *:* número indefinido de argumentos numéricos; corresponde à adição.
- - *:* dois argumentos numéricos; corresponde à subtração.
- * *:* número indefinido de argumentos numéricos; corresponde à multiplicação.
- ^ *:* um argumento numérico; corresponde à potenciação.
- < *:* dois argumentos numéricos, retorna *t* ou *nil*, de acordo com o primeiro elemento.
- > *:* dois argumentos numéricos, retorna *t* ou *nil*, de acordo com o primeiro elemento.
- <= *:* dois argumentos numéricos, retorna *t* ou *nil*, de acordo com o primeiro elemento.
- >= *:* dois argumentos numéricos, retorna *t* ou *nil*, de acordo com o primeiro elemento.

Usando as funções acima nomeadas como padrão.

A nova idéia, é definir uma Máquina de Turing Universal (MTU) semi-limitada padrão como segue:

- Seus programas estão em binário, e aparecem sobre uma fita na seguinte forma:

Primeiro, é recebida uma expressão LISP, escrita em ASCII com 8 bits por caracter, que termina com um caracter fim de linha '\n'. A MT lê estas expressões LISP, depois avalia cada expressão lida. Fazendo isto, duas novas funções primitivas *read-bit* e *read-exp*, sem argumentos, podem ser usadas para ler o restante das expressões da fita MT. As duas funções abortam se a fita está esgotada, destruindo a computação. A função *read-bit* lê um simples bit da fita. A função *read-exp* lê uma expressão LISP inteira, em caracteres de 8 bits , até alcançar um caracter de fim de linha '\n' [CHA 96a].

Esta é a única forma em que a informação sobre a fita MT pode ser acessada, e que força a usar uma forma semi-limitada. Isto se dá porque nenhum algorítmo pode procurar pelo final da fita e logo usar a longitude da fita como dado na computação. Se um algorítmo tenta ler um bit que não está sobre a fita, o algorítmo é abortado [CHA 96a].

Como pôr a informação na fita MT em primeiro lugar?. Iniciando o ambiente a fita está vazia e qualquer esforço por ler, gera uma mensagem de erro. Para pôr informação sobre a fita, deve ser usada a função primitiva ***try***, a qual comprova se uma expressão pode ser avaliada [CHA 96a].

Os significados dos três argumentos *a, b* e *c* da função ***try*** são como segue:

- O primeiro argumento é ***a***: Se *a* não é limitado no tempo, então não há nenhum limite profundo. De outro modo, *a* deve ser um inteiro decimal *unsigned*, e gera o limite profundo (limite sobre *a*, grau da função chamada e reavaliada).
- O segundo argumento *b*, da função ***try***, é a expressão a ser avaliada. O cumprimento da expressão não deve exceder o grau do limite *a*.
- O terceiro argumento *c* da função ***try*** é uma lista de bits para ser usadas como a fita MT.

O valor *v* retornado pela função primitiva ***try*** é uma terna:

1. O primeiro elemento de *v* tem sucesso se a avaliação de *b* foi completada com êxito, ou falha se este não for o caso [CHA 96a].

2. O segundo elemento de *v* está fora dos dados se a avaliação de *b* é abortada uma vez que foi feita uma tentativa para ler um bit não existente na fita MT. O segundo elemento de *v* está fora do tempo se a avaliação de *b* é abortada uma vez que o grau do limite de *a* foi excedido. Apenas estes erros são assinalados, uma vez que este LISP foi projetado com máxima tolerância de semântica. Se a computação de *b* termina normalmente em vez de abortar, o segundo elemento de *v* pode ser o resultado desta computação, isto é, seus valores. Este é o segundo elemento da lista *v* produzida pela função primitiva ***try*** [CHA 96a].

3. O terceiro elemento do valor ***v*** é uma lista de todos os argumentos para a função primitiva *display*, que foi encontrada durante a avaliação de *b*. Mais precisamente, se *display* foi chamada N vezes durante a avaliação de *b*, então *v* pode ser uma lista de N elementos. Os N argumentos de *display* aparecem em *v* em ordem cronológica. Desta forma ***try*** pode não só ser usada para determinar se a computação de *b* é longa demais (para graus maior que *a*), mas também pode ser usada para capturar todas as saídas de *b* mostradas ao longo da avaliação, se a computação de *b* for abortada ou não [CHA 96a].

Para simular a Máquina de Turing Universal semi-limitada U(*p*) rodando sobre um programa binário *p* é escrita:

try *no-time-limit* ' ***eval*** *read-exp p*

Esta é uma *M-espressão* com parênteses omitidos das funções primitivas. (rechamadas todas as funções primitivas que têm um número fixo de argumentos). Com o fornecimento dos parênteses, a *M-espressão* transforma-se na *S-expressão*:

*(**try** no-time-limit (' (**eval** (read-exp))) p)*

Isto quer dizer, que para ler uma *S-expressão* LISP *p* inteira da fita MT, e logo avaliá-la sem qualquer limite de tempo e usar tudo aquilo que está à esquerda sobre a fita *p* [CHA 96a].

5 O construtor de Funções Java

A construção de compiladores para uma linguagem de programação de propósito geral demanda um esforço significativo. Por razões práticas é necessário encontrar uma linguagem que não precisa de uma excessiva quantidade de esforço para compilar, dado que a tarefa de tradução é complexa demais para ter uma avaliação realista [SLO 95].

Neste sentido, o processo de comunicação em um ambiente distribuído requer soluções para um número importante de problemas, e potencialmente difíceis de resolver. Estes resultados são relativos à interação de tarefas paralelas e distribuídas com computação local e seqüencial, a migração eficiente de processos através dos nodos, o gerenciamento de armazenamento efetivo para aplicações distribuídas de longa vida que criam dados sobre diferentes nodos, e a implementação de protocolos para estes ambientes. O problema é antigo, e muitas soluções tem sido propostas [CEJ 95].

Quase todas essas soluções têm definido uma nova linguagem de programação ou agregado primitivas especiais para uma linguagem existente para lidar com concorrência, distribuição e comunicação. Em geral, as últimas propostas têm usado como base linguagens que tipicamente fornecem pequeno suporte para definir ou compor novas abstrações. Como resultado, a semântica de novas primitivas para lidar com programas distribuídos não podem ser facilmente expressas em termos de operações já fornecidas pela linguagem ou combinadas com facilidade para fornecer uma nova funcionalidade [CEJ 95].

A característica chave das aproximações mais populares para programação orientada a objetos é ligar cada método de um programa explicitamente para uma classe específica [PAL 95]. Como resultado, quando a estrutura de uma classe muda, os métodos freqüentemente necessitam ter modificações [PAL 95].

Sendo um objetivo da programação orientada a objetos, obter software flexível, através de mecanismos como herança [PAL 95]. Por exemplo, flexibilidade foi um dos objetivos no projeto conduzido por Booch [PAL 95], onde um pacote Ada foi convertido para formar parte de uma biblioteca de C++. Aqui foram usados "*templates*" para parametrizar certos componentes para substituições locais possíveis. Mas o grau de variação de tais componentes é limitado. Segundo Booch, " ... construir estruturas é duro. As classes das bibliotecas em geral devem equilibrar as necessidades de funcionalidade, flexibilidade e simplicidade. Esforça-se muito para construir bibliotecas flexíveis, porque nunca se conhece exatamente como um programador usará suas abstrações. Ademais, é aconselhável construir bibliotecas que fazem poucas suposições acerca de seus ambientes no possível, de tal forma que os programadores possam conhecê-las com outras classes de bibliotecas." [BOO 94].

O crescente interesse na entrega eletrônica segura de componentes de software é permitir que o conteúdo executável possa ser enviado em mensagens sobre LANs. A World Wide Web, por exemplo, é um grande sistema distribuído construído sobre o topo de uma LAN. O poder expressivo pode ser significativamente realçado se os Web

browser têm a capacidade de receber com facilidade e executar programas instalados em outros sites[CEJ 95].

Num sistema distribuído convencional rodar, localmente programas de aplicação disponíveis sobre alguns host remotos, requer baixá-los integralmente. Isto é potencialmente prejudicial se só uma pequena quantidade da funcionalidade global é requerida; por exemplo, bibliotecas ligadas a aplicações devem ser baixadas somente quando sejam realmente usadas [CEJ 95].

Considerando a implementação de programas remotos ou bibliotecas para manipular listas e para serem usadas sobre a Web, devemos ter em conta que na construção de bibliotecas para manipular listas é necessário saber que qualquer tipo de dado precisa de operações adequadas para manipular seus objetos. Assim, a linguagem LISP foi projetada para processar listas [HAS 84], além de ser uma linguagem funcional, e uma linguagem simbólica [MAU 72].

Uma alternativa para a implementação de programas remotos para serem usados sobre a Web e que permitam manipular listas é dada pela linguagem Java da Sun Microsystem [SUN 95]. Pelas características de sua construção, pode trabalhar com listas que permitem manipular dados com facilidade e clareza usando um "*string*". Os string Java são objetos padrão embutidos na linguagem [ARN 96]. Elas são instâncias da classe *java.lang.String* [FLA 96], e são tratadas como tipos primitivos, e a linguagem Java define o operador + para concatenar os string. Uma importante característica dos objetos string é que não têm métodos definidos que permitam alterar o conteúdo de um string . Para modificar o conteúdo de um string, deve-se criar um objeto *StringBuffer* do objeto string, o qual permitirá manipular o conteúdo do string.

Neste capítulo abordam-se as principais etapas envolvendo a implementação do **Construtor de funções Java** descrito no seção 4.6 do capítulo anterior. Essas questões englobam aspectos da plataforma de *hardware*, as ferramentas de *software* envolvidas e uma breve descrição da estrutura do construtor. Exemplos ilustrativos de aplicações utilizando o construtor são apresentados ao final do capítulo.

5.1 Plataforma de *hardware*

O projeto inicial deste trabalho previa a utilização dos Laboratórios do Instituto de Informática, da Universidade Federal do Rio Grande do Sul (UFRGS), e a utilização das estações de trabalho SUN como *hardware* para a implementação do **Construtor de Funções Java**. Por razões de disponibilidade, a maior parte do trabalho foi desenvolvido num Computador Pessoal (PC), com as seguintes características:

- Memória RAM de 32 MB
- Winchester de 1.28 GB
- Monitor SVGA dp 28 Ne
- Placa mãe de 256 Kb Pentium 100

5.2 Plataforma de *Software*

5.2.1 Java

A linguagem utilizada para desenvolver o trabalho é a linguagem Java da Sun Microsystem (**JDK**, versão 1.0) que executa-se nas estações de trabalho SUN, algumas com o Sistema Operativo UNIX , outras com SOLARIS e no PC anteriormente descrito com WINDOWS 95. As características da linguagem Java estão descritas no capítulo 2. A implementação compõe-se de classes escritas na linguagem Java, cujas funções e métodos utilizam a mesma linguagem para implementarem o suporte do **Construtor de Funções Java**.

5.2.2 O JDK (*Java Developer's Kit*)

Para desenvolver o trabalho se utilizaram as ferramentas para desenvolver software contidas no *JDK* versão 1.0. O propósito do JDK é fornecer um conjunto completo de ferramentas para desenvolver, testar, documentar, e executar programas Java e também Applets. Na tabela 5.1 e na figura 5.1 apresentam-se essas ferramentas.

TABELA 5.1- Ferramentas do JDK

Nome do Programa	Descrição
javac	Compilador Java
java	Interpretador
jdb	Depurador
javap	Desmontador (Disassembler)
appletviewer	Visualizador de Applets
javadoc	Gerador de documentação
javah	Gerador de arquivos de cabeçalho

Usualmente, os programas Java foram escritos usando os editores de texto do Open Windows das estações SUN, e sob WINDOWS 95 foram usados o Edit do DOS, o Java Station e parcialmente o Café (ferramenta que contem um editor Java, compilador e depurador), para escrever os arquivos fonte Java. Estes arquivos têm como base pacotes de código fonte que declaram as classes Java. Os arquivos fonte usam a extensão *.java*.

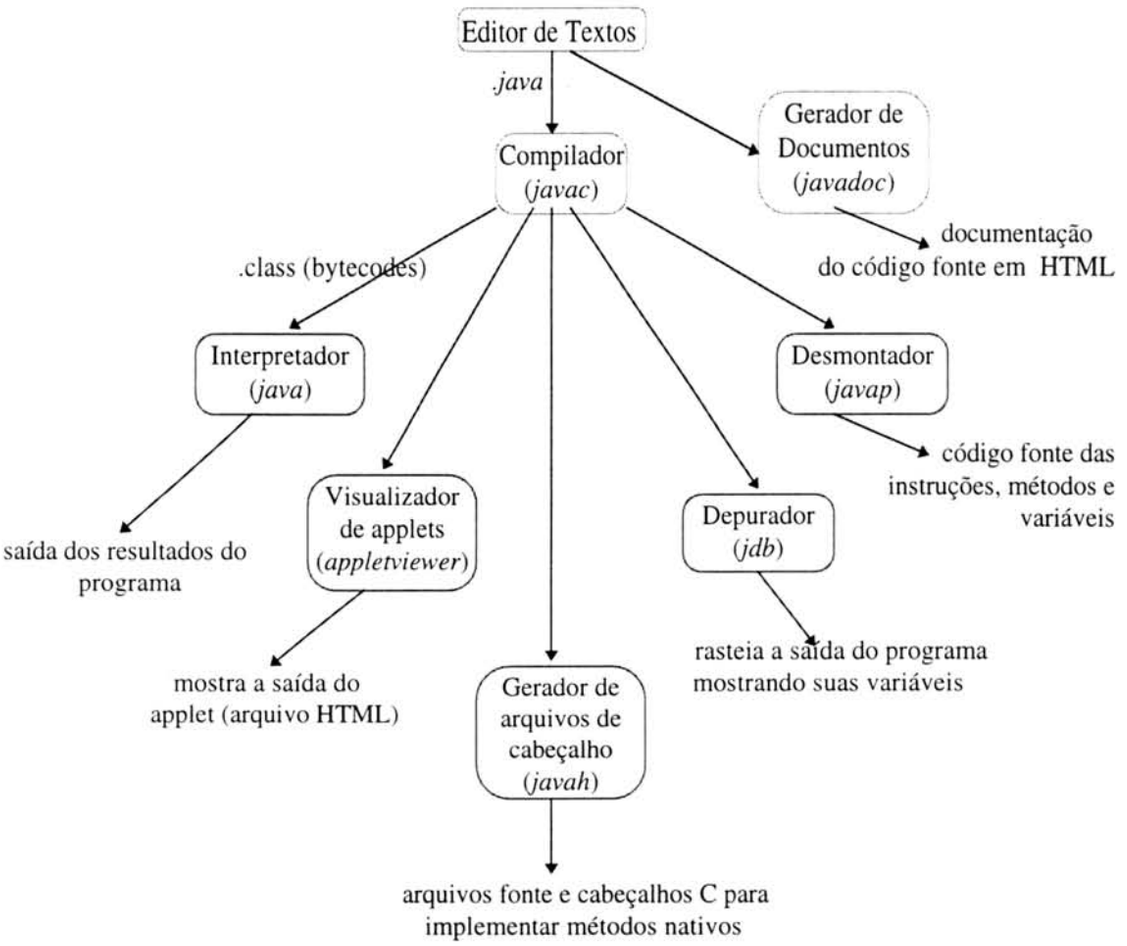

FIGURA 5.1 - Ferramentas do JDK

5.2.2.1 O Compilador (*javac*)

O compilador Java é usado para traduzir arquivos de código fonte Java em arquivos bytecode para serem executados pelo Interpretador Java (*java*). Os arquivos de código fonte devem terminar com a extensão *.java*. Estes são traduzidos em arquivos com o mesmo nome, mas com a extensão *.class*.

Por exemplo, para compilar o programa *LispJ.java* localizado no diretório de código fonte c:\java\java10\jprog, usar o comando *javac LispJ.java*. Se a compilação é correta, *javac* cria um arquivo chamado *LispJ.class* que contem o bytecode compilado de *LispJ.java*. Se a compilação indica erros, recebe-se as mensagens de erro que ajudam a corrigir os mesmos. Em geral, o compilador Java é executado como a seguir:

javac arquivo_fonte_java.*java*

A figura 5.2 ilustra a operação do Compilador Java (*javac*).

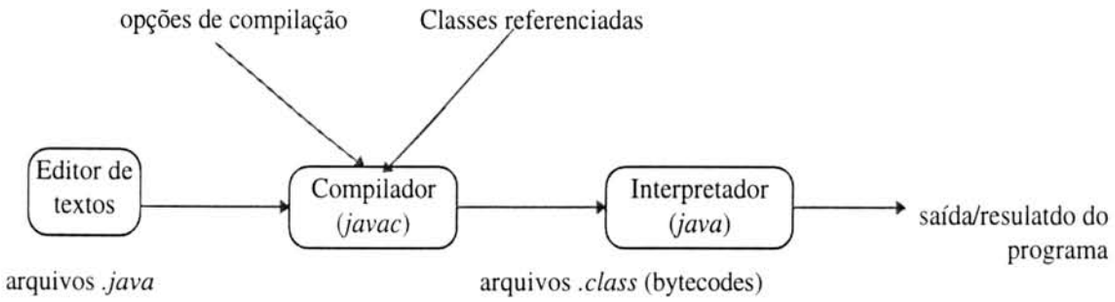

FIGURA 5.2- Operação do Compilador Java

Cada programa Java usa classes que estão definidas fora do arquivo de código fonte, as *classes externas* (*"external class"*) estão contidas no Java API. O compilador Java deve ser capaz de localizar essas classes externas para uma compilação correta do código fonte que as referencia.

Todas as declarações Java, tais como as classes, interfaces e exceções estão organizadas dentro de unidades lógicas chamadas pacotes (*"packages"*). A classe deve ser identificada como pública (*public*), para ser usada fora de seu pacote. Só as classes públicas são permitidas num arquivo fonte Java.

Uma Classe Java compilada é identificada pelo nome de seu pacote, seguida por um ponto (.), seguida pelo nome de sua classe. A figura 5.3 ilustra como os nomes dos pacotes são combinados com os nomes das classes para produzir os nomes completos das classes.

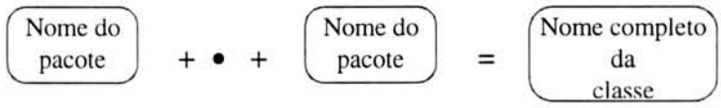

FIGURA 5.3 - Formação do nome de uma classe

O nome completo de uma classe é usado para localizar a classe com respeito à CLASSPATH. Sendo que a CLASSPATH é uma variável de ambiente do sistema usado que contém a lista de diretórios onde se encontram os pacotes Java. Ressaltando que existem diferenças no Windows 95 e no UNIX com respeito à CLASSPATH.

A CLASSPATH chama os programas do JDK onde se encontram as classes Java. Deve-se fixar a CLASSPATH para identificar a localização das classes. A fixação da CLASSPATH difere, dependendo do sistema operativo usado:

- Para Windows 95, fazer o seguinte para fixar a CLASSPATH:

  set CLASSPATH = path

  A CLASSPATH comum é .;c:\java\java10;c:\java\java10\lib\classes.zip. Isto chama ao compilador Java, e outras ferramentas do JDK para serem usadas no diretório corrente (java10), o diretório c:\java\java10 e o arquivo

c:\java\java10\lib\classes.zip são usados como uma base para encontrar os arquivos das classes Java. Para fixar a CLASSPATH, pode-se pôr esta declaração no arquivo AUTOEXEC.BAT, para ser fixada automaticamente cada vez que se inicia o DOS shell.

- Para o sistema UNIX que usa o C shell, fixar a CLASSPATH usando o comando *setenv*. Java é tipicamente instalado baixo *usr/local* em sistemas UNIX. Para fixar a CLASSPATH no diretório corrente, no diretório pessoal, e a localização das classes do JDK, usar diretamente a linha de comandos do shell ou o arquivo *.login*:

setenv CLASSPATH = . :~:/usr/local/java/lib/classes.zip

A CLASSPATH também pode ser fixada da linha de comandos do *javac* usando a opção *-classpath*. Por exemplo, para compilar o arquivo *LispJ.java* com o caminho .;c:\java\jprog, deve usar-se a seguinte linha de comando:

javac LispJ.java *-classpath* .;c:\java\jprog

A CLASSPATH fixada pela opção *-classpath* é temporária e somente aplicada para o arquivo corrente que esta sendo compilado.

5.2.2.2 O Interpretador (*java*)

O Interpretador Java executa os arquivos de *bytecodes* produzidos pelo compilador Java. Este é chamado usando o comando *java*, como a seguir:

java classe argumentos

onde classe é o nome completo gerado pela classe Java compilada usando o compilador Java. A fim de que o compilador Java localize e execute a classe, a classe deve apresentar os seguintes requisitos:

- deve ter um método *main()* válido. O método *main* é análogo à função *main* em programas C e C++.
- deve estar contida num arquivo de *bytecodes* com o mesmo nome que a classe fonte, seguida pela extensão *.class*.
- a localização da classe deve ser determinada usando a CLASSPATH e o nome completo da classe.

Os argumentos do programa são parâmetros opcionais que são passados para o método *main()* da classe que esta sendo executada. As diferentes opções do interpretador são usadas para controlar os diferentes aspectos da operação do interpretador.

5.2.2.3 O Depurador (*debugger*)

O Depurador Java é usado para monitorar a execução dos programas Java efetuando a depuração e testes das atividades. O depurador suporta depuração local e remota, executando programas Java.

O depurador pode ser iniciado em duas formas. Na primeira forma, o depurador é iniciado com o nome completo da classe. O depurador então chama o interpretador com o nome da classe a ser depurada. A segunda forma de iniciar o depurador é ligá-lo a um programa Java que já tenha sido iniciado e ainda está rodando. O programa Java, para ser depurado, deve ter sido iniciado usando a opção *-debug*.

5.2.2.4 O Desmontador (*javap*)

O Desmontador Java (*javap*), é usado para reproduzir o código fonte da classe Java compilada. Este pega o nome completo da classe como entrada e identifica as variáveis, métodos e funções que têm sido compilados dentro dos bytecodes da classe. Também identifica as instruções bytecode fonte que implementam os métodos classes.

5.2.2.5 O Visualizador de Applets (*appletviewer*)

O visualizador de applets (*appletviewer*) é usado para rodar e testar applets desenvolvidos em Java. O *appletviewer* cria uma janela na qual o applet pode ser visualizado. Este fornece suporte completo para todas as funções applet, incluindo redes e capacidades de multimídia. E é usado como a seguir:

appletviewer URL

onde a URL é a *"Universal Resource Locator"* de um documento HTML contendo o applet para ser visualizado. Este pode ser localizado numa máquina local ou sobre quaisquer website acessível na Internet.

5.2.2.6 O Gerador de Documentação (*javadoc*)

O gerador de documentos Java (*javadoc*), é a ferramenta que cria a documentação do Java API. Esta documentação pode ser obtida da website Java da Sun.

O gerador de documentação é executado usando o nome do arquivo de código fonte Java, o nome completo da classe ou um nome de pacote. Se este é executado com o nome da classe ou com o nome do pacote, este automaticamente carregará o código fonte associado com a classe ou com as classes no pacote. Se este for executado como um arquivo de código fonte, então gerará documentação para todas as classes e interfaces definidas no arquivo fonte.

O *javadoc* se usa como a seguir:

javadoc pacote
javadoc classe
javadoc LispJ.java

5.2.2.7 O Gerador de Arquivos de Cabeçalho (*javah*)

A ferramenta para gerar arquivos de cabeçalho (*javah*), é usada para produzir arquivos C, necessários para desenvolver métodos nativos. Esta produz cabeçalhos e arquivos fonte. Os arquivos de cabeçalho contêm as pontas das funções C que aplicam-se para métodos classes. O *javah* é usado como a seguir:

javah classe

O objetivo deste estudo preliminar foi conhecer as ferramentas do JDK e a forma de acessar as classes, seus métodos, e suas funções de modo a permitir uma adequação das classes ofertadas pelo JDK às especificações requeridas para implementar o **Construtor de Funções Java**.

5.3 Estratégia de Implementação

Como o objetivo deste trabalho é a simplicidade e a flexibilidade no uso da linguagem, considera-se que o esquema baseado em bibliotecas e/ou pacotes é mais flexível, pois é implementado a nível de usuário e pode ser alterado valendo-se das facilidades do mecanismo de herança da linguagem Java. Este é também mais simples, uma vez que o uso de classes com abstrações é um estilo natural da Programação Orientada a Objetos (POO). Este é o motivo que conduz à adoção de pacotes de classes Java como bibliotecas para a implementação do Construtor de Funções Java.

5.4 Estrutura do *Construtor de Funções Java*

Esta seção apresenta os módulos principais do *Construtor de Funções Java* e descreve cada uma das suas funções.

5.4.1 Os comentários

A linguagem Java permite três tipos de comentários. Pode-se usar qualquer destes três estilos para documentar os programas Java. Alguns destes estilos foram usados no código fonte do programa **LispJ.java**:

- O comentário estilo C: Começa com /* e termina com */. Exemplo:

  ```
  /* programa LispJ.java */
  ```

- O comentário estilo C++: Começa com // e continua até o final da linha. Exemplo:

  ```
  // fim do primeiro método de criação
  ```

- O comentário suportado pela documentação automática de um programa Java, começa com /** e termina com */. Este comentário é achado imediatamente antes ou depois de uma declaração Java. Exemplo:

  ```
  /** Este programa foi desenvolvido para a Dissertação de
      Mestrado de: Jorge Juan Zavaleta Gavidia.
      Instituto de Informática, Universidade Federal do Rio
      Grande do Sul, Curso de Pós-Graduação.
      Programa: LispJ.java
      Versão: 1.0
      Nota: O interpretador LISP incluído como parte deste
          programa foi desenvolvido originalmente para
          Matemática por G. J. Chaitin, IBM Research.  */
  ```

5.4.2 Estrutura do Programa *LispJ*

Os programas Java estão construídos de classes e interfaces. Uma **classe** define uma estrutura para um conjunto de dados chamados de **variáveis**, e as operações referidas pelos **métodos** que são permitidas sob as variáveis[JAW 96]. Uma **Interface** define uma coleção de métodos que devem ser implementados por uma classe[JAW 96].

O programa **LispJ.java** foi construído da classe *LispJ* o qual utiliza duas classes: *Applet* e *Stack*.

As classes e interfaces são organizadas dentro do arquivo *.java* as quais são compiladas separadamente. Os arquivos fonte *.java* são chamados de *Unidades de Compilação*. O arquivo *LispJ.java* que foi criado com um editor de textos (*JavaS*) e compilado usando o compilador Java (*javac*), é um exemplo de uma *Unidade de Compilação*.

As classes e interfaces contidas dentro das unidades de compilação estão organizadas dentro de *pacotes(packages)*. Os pacotes são usados em grupos ligados às classes, interfaces, e para evitar conflitos com os nomes dos pacotes[JAW 96]. As classes referenciadas pelo programa *LispJ* estão nos pacotes: *java.applet, java.util e java.awt*, como ilustra a figura 5.4.

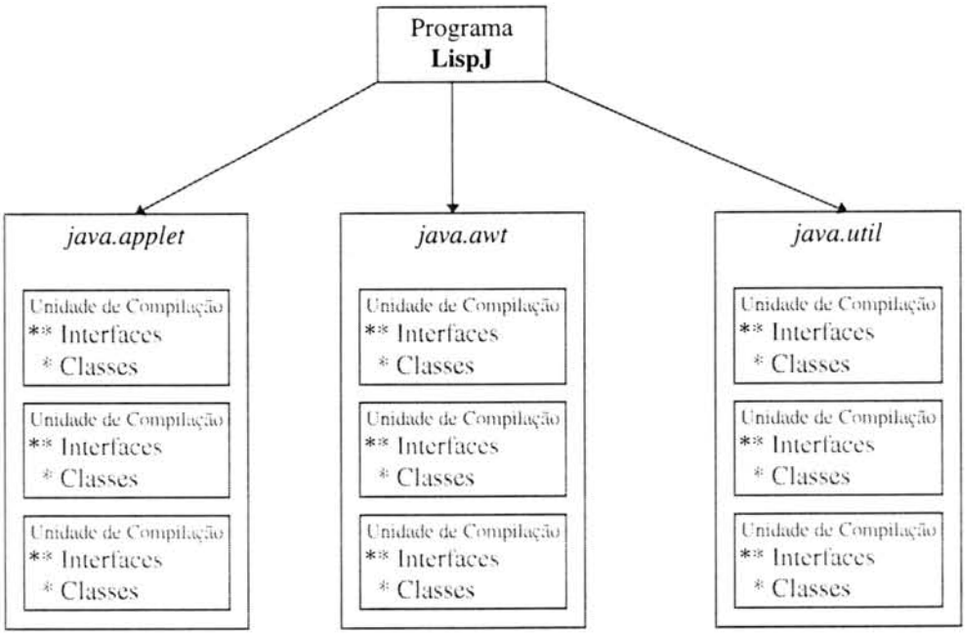

FIGURA 5.4- Estrutura de um programa Java (*LispJ*)

5.4.3 A declaração *package*

A declaração *package* (pacote), identifica em qual pacote se encontra a unidade de compilação. A declaração de *package*, deve ser a primeira declaração numa unidade de compilação. A sintaxe é:

package nome_do_pacote;

Por exemplo, a declaração *package*:

package java.jprog;

que foi usada para identificar o pacote que contém o programa *LispJ* como

java.jprog

Se a unidade de compilação não contém a declaração de *package*, as classes e interfaces contidas nas unidades de compilação são colocadas dentro de um *package* por omissão. Neste caso, o pacote não tem nome. Este pacote por omissão é o mesmo para todas as classes dentro de um diretório particular.

5.4.4 A declaração *import*

A classe *java.applet.Applet* é usada para criar e/ou implementar o applet do programa *LispJ*. Esta classes têm métodos chamados por omissão através de um web browser ou de um visualizador de applets. Exemplo: o método,

getAppletInfo()
: Este método retorna texto explicando, quem é o autor do programa, ou do applet, sua versão, direitos do autor, e assim por diante.

A classe *Applet* está no pacote *java.applet*. Para chamar o compilador *javac* para usar a classe *Applet* do pacote *java.applet*, o compilador importa a classe *Applet* usando a declaração *import*. A classe importada pelo compilador *javac* é usada quando o *javac* compila o arquivo de código fonte *LispJ*.

Da mesma forma, o pacote *java.awt*, é a caixa de ferramentas de janelas. As classes deste pacote podem serem divididas em três categorias:

- *Graphics*: Estas classes definem cores, fontes, imagens, polígonos, e outros.
- *Components*: Estas classes são componentes GUI (interfaces de Uso Gráfico), tais como botões, menus, listas, e caixas de diálogo.
- *Layout managers*: Estas classes controlam o arranjo dos componentes dentro de seus objetos recipientes.

Também é usado o pacote *java.util* que define um determinado número de classes úteis. Assim, a classe *java.util.Stack* é usada para implementar os objetos de uma pilha *"last-in-first-out"* .

A sintaxe da declaração *import* é:

import NomeDaClasse;

O nome da classe fornecido com a declaração *import* deve ser o nome de uma classe totalmente restrita à sintaxe da linguagem Java. Por exemplo, a seguinte declaração *import* importa a classe *Applet* do pacote *java.applet*:

import java.applet.Applet;

O caracter * pode ser usado em lugar do nome da classe na declaração *import*. Isto indica que todas as classes dentro do pacote devem ser importadas. Por exemplo, a seguinte declaração *import* importa todas as classes do pacote *java.awt*:

import java.awt.*;

Uma alternativa para utilizar a declaração *import* é prefixar o nome de uma classe com o nome de seu pacote. Por exemplo, a declaração:

java.lang.System.out.println(" Interpretador Lisp");

pode ter sido usada no lugar de

import java.lang.System;
.
.
.
System.out.println(" Interpretador Lisp");

A última declaração, pode ser substituída pela declaração prefixada mostrada acima.

O programa *LispJ* foi construído sob as classes *Lisp_exp* e *LispJ* cujas classes são declaradas como a seguir:

class ***Lisp_exp*** *{ // classe das expressões Lisp de tipo S-expressões*
.
.
.
}// fim da classe Lisp_exp

class ***LispJ*** *extends Applet{ // Inicio da classe LispJ*
.
.
.
public String getAppletInfo() {
return "Programação Funcional Usando Java, por: Jorge J. Zavaleta G.";
}// fim de getAppletInfo()
} // fim da classe LispJ

O arquivo *LispJ.java* é muito longo e por essa razão vamos examinar o programa *LispJ* classe por classe, método por método e função por função a fim de explicar seu funcionamento.

5.4.5 A classe *Lisp_exp*

A classe *Lisp_exp* não está definida como parte de Java API. Esta classe é declarada para manipular as expressões Lisp, assim como as expressões simbólicas chamadas de *S-expressões*. A classe *Lisp_exp* e outras classes podem ter sido definidas e compiladas separadamente, mas ela foi combinada dentro de uma simples *Unidade de Compilação* para fazer o programa *LispJ* mais compacto.

A classe *Lisp_exp* é a classe que contém os construtores das expressões Lisp com o mesmo nome da classe *Lisp_exp*, e também declara variáveis, métodos e funções descritas nas seguintes seções.

5.4.5.1 Variáveis

As variáveis são estruturas de dados que representam o estado de uma *Lisp_exp*:

```
class Lisp_exp{ // classe das expressões Lisp de tipo S-expressões
        public Lisp_exp hd = null, tl = null;
        public boolean at, nmb, err = false;
        public String pname = null;
        public long nval = 0;
        public Stack vstk = null;
        private StringBuffer str = null;
        .
        .
        .
} // fim da classe Lisp_exp
```

As variáveis definidas acima como públicas, privadas, de tipo *Lisp_exp*, *boolean*, *String*, *long*, *Stack* e *StringBuffer*, identificam o estado inicial de cada uma delas, ao momento de criar a classe *Lisp_exp*.

5.4.5.2 Métodos

Os construtores *Lisp_exp* são métodos especiais usados para inicializar um novo objeto de tipo *Lisp_exp*. Os campos podem ser iniciados com um valor quando eles são

declarados. Isto algumas vezes é necessário para assegurar um estado inicial correto [ARN 96]. Mas freqüentemente precisamos mais de um dado de inicialização para criar um estado inicial; o código criado pode precisar fornecer os dados iniciais ou realizar operações que não podem ser expressadas como uma simples atribuição, Assim :

```
public Lisp_exp(Lisp_exp h, Lisp_exp t) { // primeiro método de criação
    at = false;
    nmb = false;
    hd = h; tl = t;
    pname = new String("");
    nval = 0;
    vstk = null;
} // fim do primeiro método de criação de uma Lisp_exp
```

É o construtor (método), que permite inicializar uma expressão *Lisp_exp* de tipo *Lisp_exp*.

```
public Lisp_exp(String s) { // segundo método de criação
        at = true;
        nmb = false;
        pname = new String(s);
        nval = 0;
        hd = this; //átomos hd & tl
        tl = this;
        vstk = new Stack();
        vstk.push(this); // átomo
} // fim de criação do átomo
```

É o construtor (método) que permite inicializar um *átomo* de tipo *Lisp_exp*.

```
public Lisp_exp(long n) { // terceiro método de criação
        at = true;
        nmb = true;
        nval = n;
        Long nn = new Long(n);
        pname = nn.toString();
        hd = this; // átomos hd & tl
        tl = this;
        vstk = new Stack();
        vstk.push(this); // nunca muda
} // fim do método de criação de um número
```

É o construtor (método) que permite inicializar um *número* de tipo *Lisp_exp*.

Como Java suporta *overloading*, uma classe pode ter qualquer número de métodos construtores, todos com o mesmo nome [ARN 96][SUN95]. Baseado no número e tipos dos argumentos que passa-se dentro de um método construtor, o compilador Java pode determinar qual método construtor será usado [ARN 96].

5.4.5.3 Funções

As funções de tipo *Lisp_exp* declaradas a seguir:

```
public Lisp_exp two() { return this.tl.hd; }
public Lisp_exp three() { return this.tl.tl.hd; }
public Lisp_exp four() { return this.tl.tl.tl.hd; }
```

retornam o segundo, terceiro e quarto argumento de uma expressão *Lisp_exp*.

A função *bad()* de tipo *boolean* faz os testes correspondentes para saber quando uma expressão de tipo *Lisp_exp* é uma fórmula bem formada.

```
public boolean bad() { // teste de uma  fórmula bem formada
      if (at) return pname.equals(")");
      return hd.bad() || tl.bad();
} //fim do teste da fórmula bem formada
```

A função *toS()* de tipo *String* converte todas as expressões de tipo *Lisp_exp* para expressões de tipo *String*:

```
public String toS() { // Converte Lisp_exp para  String
      str = new StringBuffer();
      toS2(this);
      String str2 = str.toString();
      str = null;
      return str2;
} // fim da função  toS()
```

A função *toS2()* de tipo *void* imprime as expressões de tipo *Lisp_exp*:

```
private void toS2(Lisp_exp x) { // imprime expressões Lisp_exp
      if (x.at && !x.pname.equals("")) { // pname "" é nil !
      str.append(x.pname);
      return;
      }
      str.append('(');
      while (!x.at) {
      toS2(x.hd);
      x = x.tl;
      if (!x.at)
      str.append(' ');
      }
      str.append(')');
} //fim da função  toS2()
```

Todas as variáveis, os métodos e as funções referenciadas acima formam parte da classe *Lisp_exp* que por sua vez, forma parte do programa *LispJ*.

5.4.6 A classe *LispJ*

A classe *LispJ* é a classe principal do programa *LispJ*. A classe *LispJ* é uma extensão da classe *Applet*, a qual é a superclasse de todos os applets. A classe *Applet* permite implementar um applet qualquer ou criar um novo applet apropriado para cada caso, devendo criar uma subclasse da classe *Applet*, e não levando em conta alguns ou todos os métodos da classe[JAW 96]. Salientando que, nunca é preciso chamar esses métodos, estes são chamados apropriadamente por um Web browser tipo *Netscape* 2.x, *Internet Explorer* 3.x ou HotJava, ou quaisquer outro visualizador de applets tipo *Appletviewer* do JDK. A classe *LispJ* declara variáveis, métodos e funções descritas nas seguintes seções.

5.4.6.1 Variáveis

As variáveis definidas nesta classe são usadas para referenciar dados de tipos pré-definidos no Java, dados definidos pelo usuário de tipo *Lisp_exp* usando para tal efeito a função *mk_atom()*, a qual permitirá definir como átomos todas funções do Lisp, assim como, suas palavras reservadas para serem usadas pelo *Interpretador Lisp* e pelo *Construtor de Funções Java*.

```
public class LispJ extends Applet { //Classe LispJ
        /** Declaração das variáveis a serem usadas pela classe LispJ */
    long
    infinity = 999999999999999999L;
    int
    n_token = 0;
    Lisp_exp
    obj_lst = null,
    nil = mk_atom(""), // criação impossível para um átomo de uma lista vazia()
    nil2 = mk_atom("nil"), //nome alternativo para uma lista vazia()
    true2 = mk_atom("true"), // cria true
    false2 = mk_atom("false"), // cria false
    one = new Lisp_exp(1),
    zero = new Lisp_exp(0),
    quote = mk_atom("'"),
    if_then_else = mk_atom("if"),
    lambda = mk_atom("lambda"),
    rparen = mk_atom(")"), // cria parêntesis direito
    lparen = mk_atom("("), // cria parêntesis izquerdo
    time_err = mk_atom("\b"), // erro de tempo na criação do átomo
    data_err = mk_atom("\n"), // erro de dados na criação do átomo
    out_of_time = mk_atom("fora do tempo"),
    out_of_data = mk_atom("fora dos dados"),
    let = mk_atom("let"),
    car = mk_atom("car"),
```

```
cdr = mk_atom("cdr"),
cadr = mk_atom("cadr"),
caddr = mk_atom("caddr"),
atom = mk_atom("atom"),
cons = mk_atom("cons"),
equal = mk_atom("="),
fappend = mk_atom("append"),
feval = mk_atom("eval"),
ftry = mk_atom("try"),
debug = mk_atom("debug"),
size = mk_atom("size"),
length = mk_atom("length"),
display = mk_atom("display"),
read_bit = mk_atom("read-bit"),
read_exp = mk_atom("read-exp"),
bits = mk_atom("bits"),
plus = mk_atom("+"),
times = mk_atom("*"),
minus = mk_atom("-"),
to_the_power = mk_atom("^"),
leq = mk_atom("<="),
geq = mk_atom(">="),
lt = mk_atom("<"),
gt = mk_atom(">"),
success = mk_atom("sucesso"),
failure = mk_atom("falha"),
no_time_limit = mk_atom("sim-limite-tempo"),
 define = mk_atom("define");
Stack
suppress_display = new Stack(),
suppressed_displays = new Stack(),
binary_data = new Stack();

//** Mostra os botões da janela "display" */
Button reset_button = new Button("Reset");
Button run_button = new Button("Run");
Button save_button = new Button("Save");
Button restore_button = new Button("Restore");
String save = "";
Checkbox debug_button = new Checkbox("Debug",null,true);
Checkbox stdout_button = new Checkbox("Stdout",null,false);
TextArea mexp = new TextArea("Entrada de Dados",6,45);
TextArea sexp = new TextArea("Funcoes Lisp",2,45);
TextArea val = new TextArea("Valores",2,45);
TextArea f_java = new TextArea("Função Java: > ",6,45);
.
.
.
} // Fim da classe LispJ
```

5.4.6.2 Métodos

O método *init()* é um dos métodos incluídos dentro da classe *Applet*, e permite executar quaisquer inicializações de um applet. Em nosso caso, inicializará o applet *LispJ* do programa *LispJ*.

```
public void init() {
    time_err.err = true; // erro na criação do átomo
    data_err.err = true; // erro na criação do átomo

    suppress_display.push(false2); // initializa stacks
    suppressed_displays.push(nil);
    binary_data.push(nil);
    // une nil para ()
    nil2.vstk.pop();
    nil2.vstk.push(nil);
      // novas fontes
    setFont(new Font("Courier",Font.PLAIN,24));
    setForeground(Color.blue);
    add(reset_button);
    add(run_button);
    add(save_button);
    add(restore_button);
    add(mexp);
    add(sexp);
    add(val);
   add(f_java);
} // fim do método init()
```

O método booleano *action()*, é invocado pelo sistema quando o usuário está usando o applet *LispJ* e faz um clique sob quaisquer dos botões do applet.

```
public boolean action(Event evt, Object arg) {
    if (evt.target == save_button)
       save = mexp.getText();
    if (evt.target == restore_button)
       mexp.setText(save);
    if (evt.target == reset_button) {
        mexp.setText("");
        sexp.setText("");
        val.setText("");
        f_java.setText("");
    } // fim do botão reset_button
    if (evt.target == run_button) {
        sexp.setText("");
        val.setText("");
        f_java.setText("");
        buffer = stdout(mexp.getText());
```

```
                pos = 0;
                while (true) { // remove comentários
                        int i = buffer.indexOf("//");
                        if (i == -1) break;
                        int j = buffer.indexOf('\n',i);
                        if (j == -1)
                        buffer = buffer.substring(0,i);
                        else
                        buffer = buffer.substring(0,i) + buffer.substring(j);
                }// fim do lazo while
                run(); // converte m-exp para s-exp e run(roda) elas
        } // fim do botão run_button
        return true;
} // fim do método action()
```

O método *run()* de tipo *void* permite rodar o *Interpretador Lisp* e o *Construtor de Funções Java* dentro do applet *LispJ*:

```
void run() { // roda as M-expressões
        Lisp_exp s = get(); // pega as  m-exp expressões
        String st = cria_f(s,infinity); // cria a função equivalente a "s" em Java

        if (s.bad()) { //  error de sintaxes, mostra erros na janela do applet
                sexp.setText(stdout("S-exp:> "+s.toS()+" "));
                val.setText(stdout(" Error! de Sintaxes  !!! "));
                f_java.setText(stdout("Error ... na S-exp !!!"));
            return;
        } // fim de erro de sintaxe
        if (s.hd == define) {
                // define x para ter valor v
                // Si x é (fxyz) definir f para ter valor lambda(xyz)v
                Lisp_exp x = s.two();
                Lisp_exp v = s.three();
                if (!x.at) {v = jn(lambda,jn(x.tl,jn(v,nil))); x = x.hd;}
                sexp.setText(stdout("Define:> "+x.toS()+" "));
                val.setText(stdout(v.toS()+"."));
                f_java.setText(stdout("função:> "+st));
                if (!x.at || x.nmb) return;
                x.vstk.pop();
                x.vstk.push(v);
                return;
        } // fim do laço if define

        // avalia as expressões s e st
        sexp.setText(stdout("S-exp:> "+s.toS()+"  "));
        f_java.setText("Função:> "+st); //cria uma função!!
        Lisp_exp v = eval(s,infinity); // grau "infinito"
        if (v == data_err)
            val.setText(stdout("Error nos dados!"));
```

```
        else {
            val.setText(stdout("Valor:> "+v.toS()+" "));
            f_java.setText(stdout("Função:> "+st.toString()+" "));
            } //fim do laço else
} // fim do método run()
```

O método *push_env()* de tipo *void* permite ligar os átomos asi mesmos e permite pôr as expressões *Lisp_exp* sob o topo da pilha *Stack*:

```
void push_env() {// pusha novos laços
        Lisp_exp o = obj_lst;
        while (o != null) {
          o.hd.vstk.push(o.hd); // liga átomos asi mesmos
          o = o.tl;
        }
        // liga nil para ()
        nil2.vstk.pop();
        nil2.vstk.push(nil);
}// fim da função push_env()
```

O método *pop_env()* de tipo *void* permite restaurar os laços antigos e permite remover e retornar o topo das expressões *Lisp_exp* da pilha *Stack*:

```
void pop_env() {// restaura laços antigos
        Lisp_exp o = obj_lst;
        while (o != null) {
          o.hd.vstk.pop();
          if (o.hd.vstk.empty())
          o.hd.vstk.push(o.hd); // liga átomos asi mesmos
          o = o.tl;
        }//fim do laço while
}// fim do método pop_env()
```

O método *bind()* de tipo *void* liga os argumentos às variáveis, pondo os argumentos como primeiro objeto da pilha:

```
void bind(Lisp_exp vars, Lisp_exp args) {
        if (vars.at) return;
        bind(vars.tl, args.tl);
        if (vars.hd.at && !vars.hd.nmb)
        vars.hd.vstk.push(args.hd);
}// fim do método bind()
```

O método *unbind()* de tipo *void* desliga as variáveis removendo-as e retornando as expressões Lisp sob o topo da pilha *Stack*:

```
void unbind(Lisp_exp vars) {
        if (vars.at) return;
         if (vars.hd.at && !vars.hd.nmb)
```

```
        vars.hd.vstk.pop();
        unbind(vars.tl);
}// fim do método unbind()
```

5.4.6.3 Funções

A função *stdout()*,permite imprimir os *String* na tela ou na janela de saída do applet *LispJ*.

```
String stdout(String x) {
        System.out.println(x);
        return x;
}// fim da função stdout()
```

A função *jn()* de tipo *Lisp_exp*, permite retornar funções cujos argumentos podem ser, por sua vez, outras funções de tipo *Lisp_exp*:

```
Lisp_exp jn(Lisp_exp x, Lisp_exp y) {
    if (y.at && y != nil) return x;
    return new Lisp_exp(x,y);
}// fim da função jn()
```

A função *mk_atom*() de tipo *Lisp_exp* permite criar átomos para serem usados pelo Interpretador Lisp e pelo Construtor de Funções Java, para o qual é preciso passar como parâmetros de tipo String as palavras reservada, assim como as funções elementares da linguagem funcional Lisp, para que estas sejam consideradas como átomos e possam serem usadas pelo programa *LispJ*:

```
Lisp_exp mk_atom(String x) {// cria átomos
    Lisp_exp o = obj_lst;
    while (o != null) {
       if (o.hd.pname.equals(x)) return o.hd;
       o = o.tl;
    }// fim do lazo while
    Lisp_exp z = new Lisp_exp(x);
    obj_lst = new Lisp_exp(z,obj_lst);
    return z;
}// fim da função mk_atom()
```

A função *append()* de tipo *Lisp_exp* permite concatenar duas listas de tipo *Lisp_exp*:

```
Lisp_exp append(Lisp_exp x, Lisp_exp y) {// concatena duas listas
    if (x.at) return y;
    if (y.at) return x;
    return jn(x.hd,append(x.tl,y));
```

```
}// fim da função append()
```

A função *evalst()* de tipo *Lisp_exp*, permite avaliar uma lista de expressões fazendo uso da função *eval()*, a qual permite avaliar uma expressão *x* associada a uma lista *y*:

```
Lisp_exp evalst(Lisp_exp x, long d) {// avalia uma lista de expressões x
        if (x.at) return nil;
        Lisp_exp v = eval(x.hd,d); //usa a função eval()
        if (v.err) return v; // propaga erros back up
        Lisp_exp w = evalst(x.tl,d);
        if (w.err) return w; // propaga erros back up
        return jn(v,w);
}// fim da função evalst()
```

A função *eval()* retorna uma expressão Lisp (funções e palavras reservadas definidas como átomos pela função *mk_atom()*) de tipo *Lisp_exp*, a qual é avaliada e depois comparada com cada uma das expressões Lisp definidas como átomos, fazendo uso da função *evalst()* para avaliar os argumentos de cada expressão.

```
                    // avalia expressões e com listas associadas a & grau d
Lisp_exp eval(Lisp_exp e, long d) {
        if (e.at) return (Lisp_exp) e.vstk.peek(); // procura laços
        Lisp_exp f = eval(e.hd,d); // avalia a função
        if (f.err) return f; // propaga erros back up
        if (f == quote) return e.two(); // quote
        if (f == if_then_else) { // if then else
                Lisp_exp p = eval(e.two(),d); // avalia o predicado
                if (p.err) return p; // propaga erros back up
                if (p == false2) return eval(e.four(),d);
                return eval(e.three(),d); // não falso é  true
        }

       // a função evalst() permite avaliar os argumentos de uma expressão
        Lisp_exp args = evalst(e.tl,d);
        if (args.err) return args; // propaga erros back up
        Lisp_exp x = args.hd, y = args.two(); // pega o 1ro e 2do. arg
        Lisp_exp z = args.three(); // pega o terceiro argumento
        Lisp_exp v;

        if (f == size) return new Lisp_exp(x.toS().length());
        if (f == length) return new Lisp_exp(count(x));
        if (f == display) {//mostra as saídas na janela do applet
                if (suppress_display.peek() == false2) {
                f_java.appendText( stdout("Display :> "+x.toS()+" ")+ "\n" );
                }
               else {
                    Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
                    sd_ds = append(sd_ds,jn(x,nil));
```

```
            suppressed_displays.push(sd_ds);
            }
        return x;
 } // fim da função display()
if (f == read_bit) return get_bit(); //função booleana
if (f == read_exp) {// lê os fins de linha (\n) das expressões Lisp
          // lê \n dos dados binários
        if (new_line2()) return data_err; // fora dos dados ?
        v = get_exp();
        if (v == rparen) v = nil; // confere uma f.b.f
       return v;
} // fim da função read-exp()

if (f == bits) return to_bits(x);
if (f == atom) if (x.at) return true2; else return false2;
if (f == car)  return x.hd;
if (f == cdr)  return x.tl;
if (f == cons) return jn(x,y);
if (f == equal) if (eq(x,y)) return true2; else return false2;
if (f == fappend) return append(x,y);
if (f == plus) return new Lisp_exp(x.nval + y.nval);
if (f == minus) if (x.nval <= y.nval) return zero;
            else return new Lisp_exp(x.nval - y.nval);
if (f == times) return new Lisp_exp(x.nval * y.nval);
if (f == leq) if (x.nval <= y.nval) return true2; else return false2;
if (f == lt) if (x.nval < y.nval) return true2; else return false2;
if (f == geq) if (x.nval >= y.nval) return true2; else return false2;
if (f == gt) if (x.nval > y.nval) return true2; else return false2;
if (f == to_the_power) return new Lisp_exp(power(x.nval,y.nval));
if (f == base10_to_2) return to_base2(x.nval);
if (f == base2_to_10) return new Lisp_exp(to_base10(x));

if (d == 0) return time_err;  // erro fora de tempo
d = d - 1L; // decremento de grau

if (f == feval) {
          push_env();
          v = eval(x,d);
          pop_env();
          return v;
} // fim da função feval()

if (f == ftry) {
    // try novos limites de expressões de dados binários
    binary_data.push(z);
    suppress_display.push(true2);
    suppressed_displays.push(nil);
    // xx é o novo grau do limite
    long xx = x.nval;
```

```
                if (x == no_time_limit) xx = infinity;
                push_env();
                if (xx < d) // novo grau limite
                    v = eval(y,xx);
                else      // ganha antigo grau limite
                   v = eval(y,d);
                pop_env();
                Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
                suppress_display.pop();
                binary_data.pop();
                if (v == data_err) return
                  jn(failure,jn(out_of_data,jn(sd_ds,nil))); // fora dados parar aqui
                if (v != time_err) return
                     jn(success,jn(v,jn(sd_ds,nil))); // sucesso com os dados
                // dados fora de tempo
                if (xx < d) return // novo grau limite
                        jn(failure,jn(out_of_time,jn(sd_ds,nil)));
                        //não propaga erros back up
                else      // ganha antigo grau limite
                       return time_err; // propaga erro para try anterior
            } // fim da função try

            // quaisquer outro caso deve ser uma  definição de uma função
            Lisp_exp vars = f.two(), body = f.three();
                // laço
                bind(vars,args);
                v = eval(body,d);
                // sem laço
                unbind(vars);
                return v;
    } // fim da função  eval()
```

A função *power()* de tipo *long*, permite expressar a potência de um determinado número com uma *base* e com um exponente *exp*:

```
long power(long base, long exp) {
        if (exp == 0) return 1L;
        return base * power(base,exp-1L);
}// fim da função power()
```

A função *count()* de tipo *long* permite contar o limite de uma expressão Lisp:

```
long count(Lisp_exp x) {
        if (x.at) return 0;
       return 1L + count(x.tl);
}// fim da função count()
```

A função *eq()* de tipo booleano, permite conferir se as expressões Lisp são iguales retornando a igualdade, em quaisquer outro caso retorna falso:

```
boolean eq(Lisp_exp x, Lisp_exp y) {
        if (x.nmb && y.nmb) return x.nval == y.nval;
         if (x.nmb || y.nmb) return false;
         if (x.at && y.at) return x == y;
         if (x.at || y.at) return false;
         return eq(x.hd,y.hd) && eq(x.tl,y.tl);
}// fim da função eq()
```

A função *get_lst()* de tipo *Lisp_exp* permite pegar as listas de S-expressões das *M-expressões* que são introduzidas na janela de E*ntrada de Dados* do applet *LispJ*, usando a função *get()* para pegar as S-expressões simples:

```
Lisp_exp get_lst() { // pega lista de s-exps das m-exp
         Lisp_exp v = get();
         if (v == rparen) return nil;
         Lisp_exp w = get_lst();
         return jn(v,w);
}// fim da função get_lst()
```

A função *get()* de tipo *Lisp_exp* permite pegar as *S-expressões* simples das *M-expressões* introduzidas através da janela *Entrada de Dados* do applet *LispJ*, usando a função *next_token()* para pegar as *S-expressões* (funções e variáveis Lisp) :

```
Lisp_exp get() { // pega as S-exp simples das  M-exp
         String t = next_token();
         long n = nval(t);
         Lisp_exp a = null;
         if (n == -1L)
         a = mk_atom(t); // cria token dentro de um átomo
         else
         a = new Lisp_exp(n); // cria número
         if (a == lparen) return get_lst(); // lista explicita

         // funções primitivas sem argumentos
         if (a == read_bit ||
            a == read_exp) return jn(a,nil);

         // funções primitivas com um argumento
         if (a == quote ||
            a == atom ||
            a == car ||
            a == cdr ||
            a == display ||
            a == debug ||
            a == size ||
            a == length ||
            a == base10_to_2 ||
            a == base2_to_10 ||
```

```
    a == feval ||
    a == bits) return jn(a,jn(get(),nil));

// funções primitivas com dois argumentos
if (a == cons ||
    a == equal ||
    a == plus ||
    a == minus ||
    a == times ||
    a == to_the_power ||
    a == leq ||
    a == geq ||
    a == lt ||
    a == gt ||
    a == define ||
    a == fappend ||
    a == lambda) return jn(a,jn(get(),jn(get(),nil)));

// funções primitivas com três argumentos
if (a == if_then_else ||
    a == ftry) return jn(a,jn(get(),jn(get(),jn(get(),nil))));
if (a == cadr) { // car de cdr
            Lisp_exp v = get();
            v = jn(cdr,jn(v,nil));
            v = jn(car,jn(v,nil));
            return v;
}// fim de laço if cadr

if (a == caddr) { // car de cdr de cdr
            Lisp_exp v = get();
            v = jn(cdr,jn(v,nil));
            v = jn(cdr,jn(v,nil));
            v = jn(car,jn(v,nil));
            return v;
}//fim do laço if caddr

if (a == let) { // seja x para ser v na expressão e
        Lisp_exp x = get(), v = get(), e = get();
        if (!x.at) {
                v = jn(quote,
                jn(jn(lambda, jn(x.tl,jn(v,nil))), nil));
        x = x.hd;
        }// fim do laço if let
        // seja (fxyz) para ser v na expressão e
        return    jn(jn(quote,
                  jn(jn(lambda,
                  jn(jn(x,nil),
                  jn(e,nil))), nil)),
                  jn(v,nil));
```

```
        } // fim do laço let

        return a; // retorna o átomo do token t

} // fim da função get()
```

A função *nval()* de tipo *long* permite calcular o valor de um String:

```
long nval(String s) {
        long n = 0; int i = 0;
        while (i < s.length()) {
            char d = s.charAt(i);
            if (d < '0' || d > '9') return -1L;
            n = 10L*n + ((long)d) - ((long)'0');
            i = i + 1;
      }// fim do laço while
     return n;
} // fim da função nval()
```

A função *to_base10()* de tipo *long* permite expressar uma *Lisp_exp* em forma decimal:

```
long to_base10(Lisp_exp s) {
        long n = 0;
        while (!s.at) {
            n = n + n;
            if (!s.hd.pname.equals("0")) n = n + 1L;
            s = s.tl;
        }// fim do laço while
        return n;
}// fim da função to_base 10()
```

A função *to_base2()* de tipo *Lisp_exp* permite expressar qualquer número em base 2(binário):

```
Lisp_exp to_base2(long n) {
        Lisp_exp s = nil;
        while (n > 0) {
            if ((n%2L) != 0)
                s = jn(one,s);
            else
                s = jn(zero,s);
            n = n >>> 1;
        }// fim do laço while
        return s;
}// fim da função to_base2()
```

A função booleana *new_line2()* permite ler os caracteres de fim de linha (\n) dos dados binários e cria uma nova linha:

```
boolean new_line2() { // le a \n dos dados binários
        StringBuffer str = new StringBuffer();
        char ch;
        while (true) {
                ch = get_char();
                if (ch == '\b') return true;
               str.append(ch);
              if (ch == '\n') break;
        } // fim do laço while
        buffer = str.toString();
        pos = 0;
        return false;
}// fim da função new_line2()
```

Define-se variáveis para serem usadas dentro da função *next_token()*:

```
// buffer & cursor para os token
String buffer = null;
int pos;
```

A função *next_token()* de tipo *String* permite pegar os tokens dos dados de entrada para serem usados pelo Interpretador e Construtor de Funções:

```
String next_token() { // token de entrada ou dado binário
        StringBuffer token = new StringBuffer();
        while (true) {
              char ch;
              while (true) { // elimina caracteres falhados
                      if (pos == buffer.length()) ch = ')';
                      else ch = buffer.charAt(pos++);
                     // guarda só \n ou codigo ascii imprimivel
                     if (ch == 10 || (ch >= 32 && ch < 127)) break;
              }// fim do laço while
              if (token.length() == 0) { // buffer de token vazio
                  if (ch == ' ' || ch == '\n') continue;
                  token.append(ch);
                 if (ch == '(' || ch == ')' || ch == '') break;
              }
             else { // buffer de token nao vazio
                 if (ch != ' ' && ch != '\n' && ch != '(' && ch != ')' && ch != '')
                 token.append(ch);
                  if (ch == '(' || ch == ')' || ch == '')
                  pos = pos - 1;
                 if (ch == ' ' || ch == '\n' || ch == '(' || ch == ')' || ch == '')
                 break;
                } fim do laço else
        } // fim do laço while
       return token.toString();
}// fim da função next_token()
```

A função *get_bit()* de tipo *Lisp_exp* permite ler os bits dos dados binários de uma expressão Lisp.

```
Lisp_exp get_bit() { // le bit de dados binarios
        Lisp_exp data = (Lisp_exp) binary_data.peek();
        if (data.at) return data_err; // fora de dados
        binary_data.pop();
        Lisp_exp v = data.hd;
        binary_data.push(data.tl);
        // não zero é considerado um
        if (!v.pname.equals("0")) v = one;
        return v;
}// fim da função get_bit()
```

A função *to_bits()* permite converter *S-expressões* para um string de bits:

```
Lisp_exp to_bits(Lisp_exp x) { // converte S-exp para bit string
        String str = x.toS().concat("\n");
        Lisp_exp v = nil;
        int i = str.length();
        while (i > 0) {
                i = i - 1;
                int j = ((int) str.charAt(i)) % 256;
                int k = 0;
                while (k < 8) {
                        if ((j % 2) != 0) v = jn(one,v);
                        else              v = jn(zero,v);
                        j = j >>> 1;
                        k = k + 1;
                }// fim do laço while
            }// fim do laço while
        return v;
}// fim da função to_bit()
```

A função *get_char()* permite ler os caracteres dos dados binários fazendo uso da função *get_bit()* para ler os bits:

```
char get_char() { // lê caracteres de dados binários
 int k, v;
 do {
    k = 0; v = 0;
     while (k < 8) {
            Lisp_exp b = get_bit();
            if (b.err) return '\b'; // fora dos dados
            v = v << 1;
            if (b == one) v = v + 1;
            k = k + 1;
      }
```

```
} while (v != 10 && (v < 32 || v >= 127));
// guarda só \n ou o código ascii imprimivel
return (char) v;
}// fim da função get_char()
```

A função *get_list()* permite pegar as listas das *S-expressões* dos dados binários fazendo uso da função *get_exp()*:

```
Lisp_exp get_list() { // pega lista de s-exps de dados binários
        String[] lisp_t;
        Lisp_exp v = get_exp();
        if (v == rparen) return nil;
        Lisp_exp w = get_list();
        return jn(v,w);
} // fim da função get_list()
```

A função *get_exp()* permite pegar as *S-expressões* simples dos dados binários fazendo uso da função *next_token()*:

```
Lisp_exp get_exp() { // pega simples s-exp de dados binários
        String t = next_token();
        n_token=n_token+1;
        long n = nval(t);
        Lisp_exp a = null;
        if (n == -1L)
          a = mk_atom(t); // introduze um token dentro de um átomo
        else
          a = new Lisp_exp(n); // cria um número
        if (a == lparen) return get_list(); // lista explicita
        return a;
} // fim da função get_exp()
```

A função *cria_f()*, do tipo String, permite criar funções Java correspondentes às funções Lisp usadas pelo *Interpretador Lisp*, e que foram definidas como átomos pela função *mk_atom()*. As mesmas funções serão usadas pela função *eval()* para avaliar as funções Lisp e seus argumentos para depois serem convertidas para código fonte Java.

Usa-se a classe *StringBuffer* na implementação do *Construtor de Funções Java*, a qual permite a manipulação de strings mais complexos. O compilador de Java automaticamente declara e manipula objetos desta classe para implementar operações strings comuns [JAW 96]. Esta classe fornece três construtores: um construtor vazio, um construtor vazio com um buffer inicial de longitude específica, e um construtor que cria um objeto *StringBuffer* de um objeto string [JAW 96], os quais serão usados na implementação.

Na implementação do *Construtor de Funções Java* usa-se a classe *StringBuffer* da linguagem Java para manipular os strings e as funções *eval()* e *evalst()* do *Interpretador Lisp* detalhadas anteriormente, salientando que a função *eval()* avalia as funções ingressadas através da janela de *Entrada de Dados*, com as funções feitas

átomos usando a função *mk_atom()*. A avaliação é realizada através da relação de igualdade da função ingressada com a função armazenada como átomo. A função *evalst()* avalia uma lista de expressões Lisp de tipo *Lisp_exp*, assim como os argumentos das funções. Estas funções serão usadas para criar uma função de tipo *String* chamada *cria_f()* ou *"Construtor de Funções Java"*, a qual criará uma função em código fonte Java, segundo seja o caso e de acordo à função *eval()* usando os argumentos adequados segundo a função *evalst()*, o resultado da avaliação da função como de seus argumentos serão convertido para string usando a função *toString()*:

```
/** Modulo do Construtor de Funções Java
    Função construtor Cria_f() */

public String cria_f(Lisp_exp e,long d){// função contrutor
      StringBuffer imp_f = new StringBuffer();
       Lisp_exp f = eval(e.hd,d); // avalia as funções
       if (f == quote){ String nome_f = "QUOTE";
         imp_f.append("public Lisp_exp ").append(nome_f).append("() { \n");
         imp_f.append("\t return e.two();\n");
         imp_f.append("\t } \n");
         return imp_f.toString();}
      if (f == if_then_else) { // if then else
         String nome_f = "IF_THEN_ELSE";
         imp_f.append("public Lisp_exp ").append(nome_f).append("(){ \n");
         imp_f.append("\t Lisp_exp p = eval(e.two(),d);\n");
         imp_f.append("\t if (p.err) return p;\n");
         imp_f.append("\t if (p == false2) return eval(e.four(),d);\n");
         imp_f.append("\t return eval(e.three(),d);\n");
         imp_f.append("\t } \n");
         return imp_f.toString();
         }// fim de if_then_else

  // avalia os argumentos
  Lisp_exp args = evalst(e.tl,d);
     //if (args.err) return args.toString(); // propaga erros back up
  Lisp_exp x = args.hd, y = args.two(); //pega o primeiro e segundo args
  Lisp_exp z = args.three(); // pega o terceiro argumento
  Lisp_exp v;

  if (f == atom){String nome_f = "ATOM";
        imp_f.append("public boolean ").append(nome_f).append("() {\n");
        imp_f.append("\t if (x.at) return true2;\n");
        imp_f.append("\t else return false2;\n");
       imp_f.append("\t } \n");
        return imp_f.toString();}
    // fim de atom

  if (f == car){String nome_f = "CAR";
```

```
    imp_f.append("public   Lisp_exp   ").append(nome_f).append("(Lisp_exp   x)
{\n");
    imp_f.append("\t return x.hd;\n");
    imp_f.append("\t } ");
         return imp_f.toString(); }

  if (f == cdr){String nome_f = "CDR";
    imp_f.append("public   Lisp_exp   ").append(nome_f).append("(Lisp_exp   x)
{\n");
    imp_f.append("\t return x.tl;\n");
    imp_f.append("\t } ");
            return imp_f.toString(); }

  if (f == cons){String nome_f = "CONS";
    imp_f.append("public   Lisp_exp   ").append(nome_f).append("(Lisp_exp   x,
Lisp_exp y) {\n");
    imp_f.append("\t return jn(x,y);\n");
    imp_f.append("\t  } ");
           return imp_f.toString();}

  if (f == equal){String nome_f = "EQUAL";
    imp_f.append("public   boolean   ").append(nome_f).append("(Lisp_exp   x,
Lisp_exp y) {\n");
    imp_f.append("\t if (eq(x,y)) return true2;\n");
    imp_f.append("\t else return false2;\n");
    imp_f.append("\t  } ");
           return imp_f.toString();}

  if (f == fappend){ String nome_f = "FAPPEND";
    imp_f.append("public   Lisp_exp   ").append(nome_f).append("(Lisp_exp   x,
Lisp_exp y) {\n");
    imp_f.append("\t return append(x,y);\n");
    imp_f.append("\t  } ");
            return imp_f.toString(); }

  if (f == plus){String nome_f = "PLUS";
  imp_f.append("public   Lisp_exp   ").append(nome_f).append("(Lisp_exp   x,
Lisp_exp y) {\n");
  imp_f.append("\t return new Lisp_exp(x.nval + y.nval);\n");
  imp_f.append("\t  } ");
            return imp_f.toString(); }

  if (f == minus){String nome_f = "MINUS";
  imp_f.append("public   Lisp_exp   ").append(nome_f).append("(Lisp_exp   x,
Lisp_exp y) {\n");
  imp_f.append("\t if (x.nval <= y.nval) return zero;\n");
  imp_f.append("\t else return new Lisp_exp(x.nval - y.nval);\n");
  imp_f.append("\t  }  ");
            return imp_f.toString();}
```

```
  if (f == times){String nome_f = "TIMES";
  imp_f.append("public    Lisp_exp    ").append(nome_f).append("(Lisp_exp    x,
Lisp_exp y) {\n");
  imp_f.append("\t return new Lisp_exp(x.nval * y.nval);\n");
  imp_f.append("\t } ");
            return imp_f.toString();}

  if (f == leq){String nome_f ="LEQ";
  imp_f.append("public    boolean    ").append(nome_f).append("(Lisp_exp    x,
Lisp_exp y) {\n");
  imp_f.append("\t if (x.nval <= y.nval) return true2;\n");
  imp_f.append("\t else return false2;\n");
  imp_f.append("\t  } ");
           return imp_f.toString(); }

  if (f == lt){String nome_f ="LT";
  imp_f.append("public    boolean    ").append(nome_f).append("(Lisp_exp    x,
Lisp_exp y) {\n");
  imp_f.append("\t if (x.nval < y.nval) return true2;\n");
  imp_f.append("\t else return false2;\n");
  imp_f.append("\t  }  ");
          return imp_f.toString(); }

  if (f == geq){String nome_f = "GEQ";
  imp_f.append("public    boolean    ").append(nome_f).append("(Lisp_exp    x,
Lisp_exp y) {\n");
  imp_f.append("\t if (x.nval >= y.nval) return true2;\n");
  imp_f.append("\t else return false2;\n");
  imp_f.append("\t  } ");
           return imp_f.toString(); }

  if (f == gt){String nome_f = "GT";
  imp_f.append("public    boolean    ").append(nome_f).append("(Lisp_exp    x,
Lisp_exp y) {\n");
  imp_f.append("\t if (x.nval > y.nval) return true2;\n");
  imp_f.append("\t else return false2;\n");
  imp_f.append("\t  } ");
          return imp_f.toString(); }

  if (f == to_the_power){String nome_f = "TO_POWER";
  imp_f.append("public    Lisp_exp    ").append(nome_f).append("(Lisp_exp    x,
Lisp_exp y) {\n");
  imp_f.append("\t return new Lisp_exp(power(x.nval,y.nval));\n");
  imp_f.append("\t  }  ");
          return imp_f.toString(); }

  if (d == 0) return time_err.toString();//erro fora de tempo
  d = d - 1L; // decremento de grau
```

```
if (f == feval) {
    push_env();
    v = eval(x,d);
    pop_env();
    return v.toString();
 } // fim de eval

if (f == ftry) {

    binary_data.push(z);
    suppress_display.push(true2);
    suppressed_displays.push(nil);
    // xx e novo grau limite
    long xx = x.nval;
    if (x == no_time_limit) xx = infinity;
    push_env();
    if (xx < d)
      v = eval(y,xx);
    else      // ganha antigo grau limite
      v = eval(y,d);
    pop_env();
    Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
    suppress_display.pop();
    binary_data.pop();
    if (v == data_err) return
      jn(falha,jn(out_of_data,jn(sd_ds,nil))).toString(); //fora dos dados
    if (v != time_err) return
      jn(sucesso,jn(v,jn(sd_ds,nil))).toString();
    // fora de tempo
    if (xx < d) return
      jn(falha,jn(out_of_time,jn(sd_ds,nil))).toString();
    else      // ganha antigo grau limite
      return time_err.toString();
  } // fim de try

  //de outra forma deve ser a definicao de uma funcao
    Lisp_exp vars = f.two(), body = f.three();
    // lazo
    bind(vars,args);
    v = eval(body,d);
    // sim lazo
    unbind(vars);
    return v.toString();

} //fim da função cria_f()
```

5.5 Operação do *Construtor de Funções Java*

Esta seção descreve a operação do *Construtor de Funções Java (LispJ)*, usando o applet *LispJ*. A descrição toma como base a versão final do *Construtor de Funções Java (LispJ)*, isto é, a que será executada num Web Browser (Netscape 2.x ou superior e no IE 3.x ou superior) ou num visualizador de applets (*appletviewer* do JDK).

A operação do *Construtor de Funções Java*, pode ser dividida em fases, as quais estão listadas a seguir:

- ativação do *LispJ* usando um browser ou visualizador de applets
- funcionamento do applet *LispJ* do *Construtor de Funções Java*
- conversão de uma função Lisp em Java
- encerramento do applet *LispJ*

Cada uma das fases acima está descrita em uma seção particular. Além disso, optou-se por adicionar uma seção de exemplos para uma melhor comprensão da forma como opera o *Construtor de Funções Java*.

5.5.1 Ativação do *LispJ* usando um browser ou visualizador de applets

A ativação do *LispJ* pode ser realizada a partir de qualquer plataforma de sistema operacional que suporte Java. No entanto, a descrição abaixo toma por base o browser Netscape Gold 3.0 versão beta, assim como, o Internet Explorer 3.0 sobre windows 95 e SOLARIS sobre uma estação de trabalho SUN do Instituto de Informática da UFRGS.

A ativação do Construtor de Funções Java *(LispJ)* pode ser feito usando quaisquer browser mencionado anteriormente, no seguinte endereço na Internet:

◊ http://www.inf.ufrgs.br/~jorgezg/LispJ.html

◊ http://thecity.sfsu.edu/~condor/LispJ.html

e pode ser conferido nas figuras capturadas e mostradas abaixo, feitas usando o Print Shop Pro 4.1 no Windows 95 e o SpanaShot nas estações SUN.

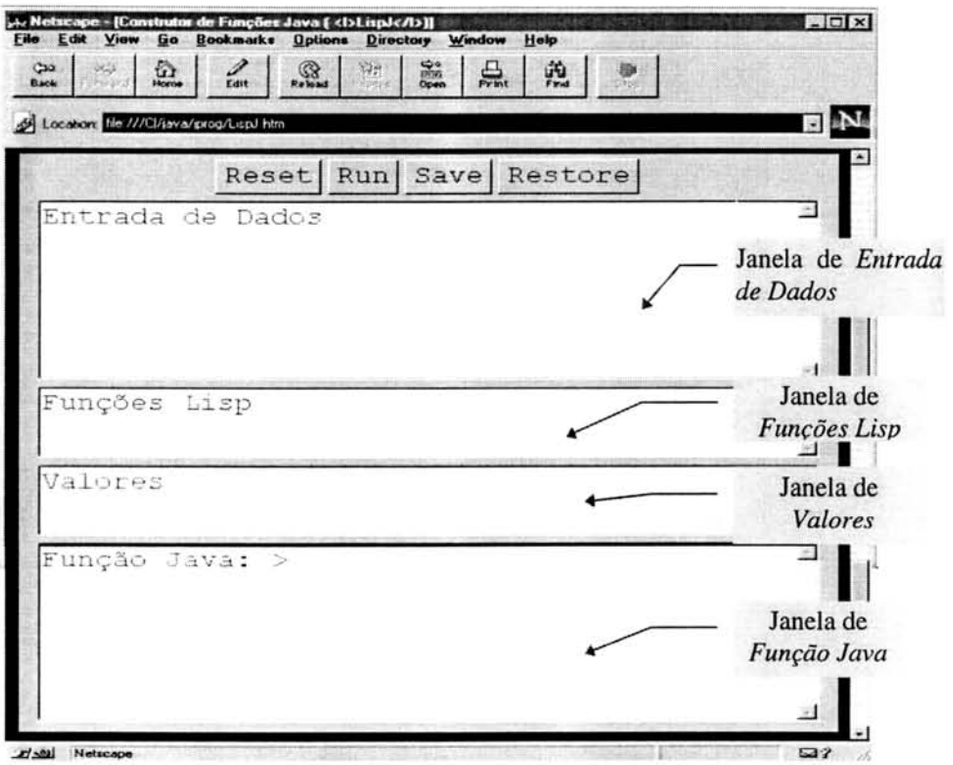

FIGURA 5.5 O *LispJ* no Netscape Gold 3.0

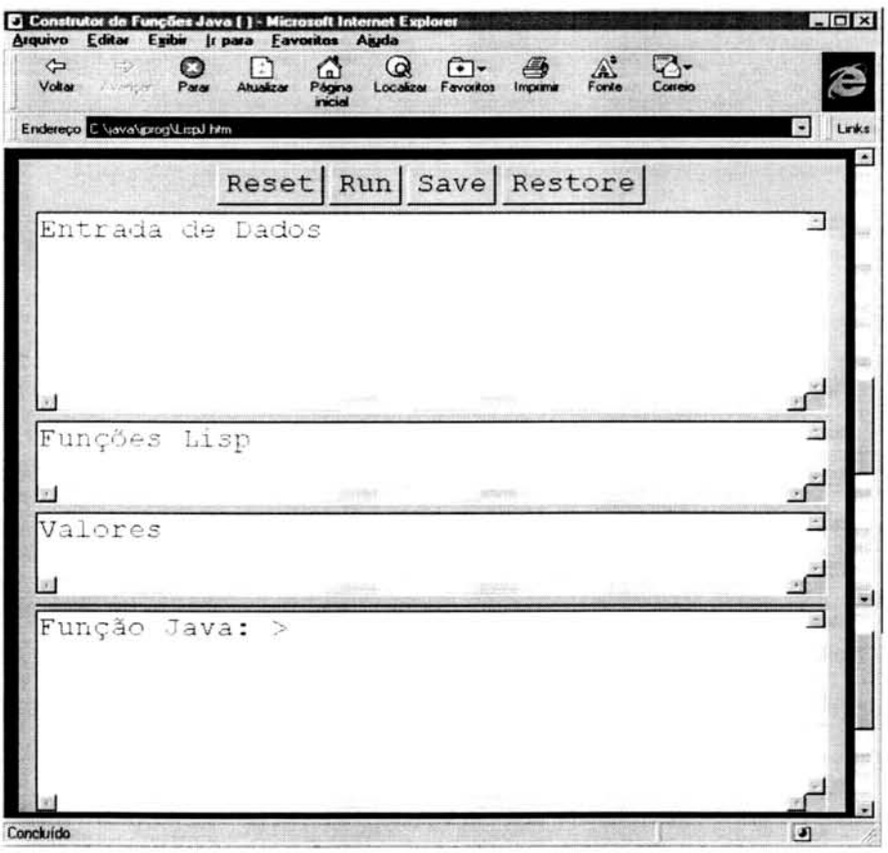

FIGURA 5.6 O *LispJ* no Internet Explorer 3.0

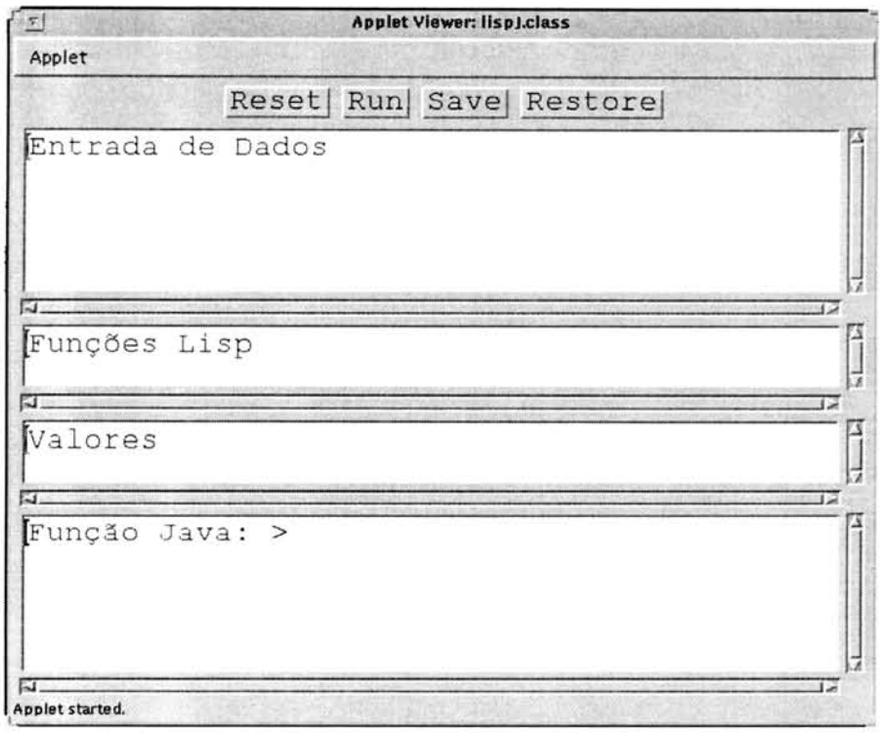

FIGURA 5.7 O *LispJ* no appletviewer do JDK 1.0

5.5.2 Funcionamento do applet *LispJ* do *Construtor de Funções Java*

O applet Java *LispJ*, permite a execução das partes que compõem o *Construtor de Funções Java*, bem como a visualização das funções de entrada e saída. O applet *LispJ* possui quatro botões na parte superior (*Reset, Run, Save, Restore)* e quatro janelas (*Entrada de Dados, Funções Lisp, Valores, Função Java*). Os botões superiores têm as seguintes funções:

- ***Reset:*** Permite reinicializar e deixar as janelas do applet *LispJ* prontas para receber informação.

- ***Run:*** Permite acionar o método *RUN()* da classe *LispJ* e executa os dados ingressados na janela de *Entrada de Dados*.

- ***Save:*** *Permite* gravar os dados num arquivo (não implementado)

- ***Restore:*** Restaura os dados ingressados anteriormente.

As janelas têm as seguintes funções:

- ***Entrada de Dados:*** Permite a manipulação e entrada do código fonte das funções Lisp.

- ***Funções Lisp:*** Permite visualizar a saída do código fonte das funções Lisp (*S-expressões*) ingressadas na janela de *Entrada de Dados*.

- ***Valores:*** Permite visualizar os valores de saída das funções Lisp ingressadas ou visualiza mensagens de erro.

- ***Função Java:*** Permite visualizar a saída do código fonte em Java das funções Lisp (*M-expressões*) ingressadas através da janela de *Entrada de Dados*.

5.5.3 Conversão de uma função Lisp em Java

Para ativar a operação de conversão de funções Lisp em funções Java usando o applet *LispJ* usa-se o **mouse** (botão esquerdo) da seguinte maneira:

- Pressionar o botão *Reset* para limpar e deixar pronta a janela de *Entrada de Dados*.
- Pressionar a botão esquerdo do **mouse** sob a janela de *Entrada de Dados*, para iniciar a entrada do código fonte das funções Lisp a serem convertidas a código Java.
- Ingressar o código fonte das funções Lisp.
- Pressionar o botão *Run* com o **mouse** para executar a operação de conversão de Lisp para Java.

5.5.4 Encerramento do applet *LispJ*

O encerramento da execução do applet *LispJ* é ordenada através de uma URL diferente à do applet *LispJ* no caso de estar usado um Web browser, ou no caso de executar o *LispJ* usando um visualizador de applets (*appletviewer)*, pressionar o botão *applet* da parte superior do visualizador e escolher a opção *quit* do menu mostrado para deixar o visualizador de applets.

5.5.5 Exemplos de Resultados

Os exemplos incluídos nesta seção têm por objetivo ilustrar o funcionamento do applet *LispJ* do *Construtor de Funções Java* para uma melhor compreensão do mesmo. Para maiores detalhes conferir no Anexo 1.

Exemplo nro. 1: (função *car*, pega o primeiro elemento da lista)

Dados: *car '(aa bb cc)*
S-exp:> *(car (' (aa bb cc)))*
Valor:>> *aa*
Função:> *public Lisp_exp CAR(Lisp_exp x) {*
return x.hd;
}

Exemplo nro. 2: (função *cdr*, pega o resto de uma lista)

Dados: *cdr '(aa bb cc)*
S-exp:> *(cdr (' (aa bb cc)))*
Valor:>> *(bb cc)*
Função:> *public Lisp_exp CDR(Lisp_exp x) {*
return x.tl;
}

Exemplo nro. 3: (função *atom*, confere os átomos)

Dados: *atom '(aa)*
S-exp:> *(atom (' (aa)))*
Valor:>> *false*
Função:> *public boolean ATOM() {*
if (x.at) return true2;
else return false2;
}

Exemplo nro. 4: (função *iqual*, confere igualdade das *S-expressões*)

Dados: = *aa bb*
S-exp:> *(= aa bb)*
Valor:>> *false*
Função:> *public boolean EQUAL(Lisp_exp x, Lisp_exp y) {*
if(eq(x,y)) return true2;
else return flase2;
}

Exemplo nro. 5: (função *iqual*, confere igualdade das *S-expressões*)

Dados: = *aa aa*
S-exp:> *(= aa bb)*
Valor:>> *true*
Função:> *public boolean EQUAL(Lisp_exp x, Lisp_exp y) {*
if(eq(x,y)) return true2;
else return flase2;
}

Exemplo nro. 6: (função *if ... then ... else*)

Dados: *if false x y*
S-exp:> *(if false x y)*
Valor:>> *y*
Função:> *public Lisp_exp IF_THEN_ELSE() {*
Lisp_exp p = eval(e.two(), d);
if(p.err) return p;
if(p = false2) return eval(e.four(),d);
return eval(e.three(),d);
}

Exemplo nro. 7: (função *plus*)

Dados: *+ abc 50*
S-exp:> *(+ abc 50)*
Valor:>> *50*
Função:> *public Lisp_exp PLUS(Lisp_exp x, Lisp_exp y) {*
return new Lisp_exp(x.nval + y.nval);
}

Exemplo nro. 8: (função *plus*)

Dados: *+ 10 50*
S-exp:> *(+ 10 50)*
Valor:>> *60*
Função:> *public Lisp_exp PLUS(Lisp_exp x, Lisp_exp y) {*
return new Lisp_exp(x.nval + y.nval);
}

Exemplo nro. 9: (função *times*)

Dados: ** 10 15*
S-exp:> *(* 10 15)*
Valor:>> *150*
Função:> *public Lisp_exp TIMES(Lisp_exp x, Lisp_exp y) {*
*return new Lisp_exp(x.nval * y.nval);*
}

Exemplo nro. 10: (função *gt* (maior que))

Dados: *> 60 50*
S-exp:> *(> 60 50)*
Valor:>> *true*
Função:> *public boolean GT(Lisp_exp x, Lisp_exp y) {*
if(x.nval > y.nval) return true2;
else return false2;
}

5.6 Aplicações

A principal dificuldade que se tem com as funções de ordem superior, é que não é fácil determinar a forma na qual a função está escrevendo os tipos de argumentos os quais pode tomar. Desta forma, devemos ser explícitos acerca dos tipos desses argumentos, e para este propósito devemos desenvolver uma notação adequada [HER 80]. O tipo de um argumento para uma função de ordem superior é denotada por um *"tipo-expressão"*. A propriedade crucial destas *"tipo-expressão"* é que, quando o argumento cujo tipo esta sendo especificado é em si mesmo uma função, então contém ao *"tipo-expressão"*, retornando uma especificação dos tipos de argumentos da função. Quando têm-se funções cujos argumentos são em si mesmos funções de ordem superior, então necessitamos ser explícitos acerca dos tipos de argumentos [HER 80], tal como faz a linguagem Java.

Na declaração de uma função, dá-se um nome para um objeto dado. Pode-se introduzir novos tipos dado com novos objetos dado, em especial declarações de tipos dado. Um tipo dado é um conjunto de objetos dado e uma coleção de operações disponíveis para manipular esses objetos dado. Por exemplo, o tipo *booleano* Java contém dois objetos dado *true* e *false*.

Um programa junto com a entrada de dados usualmente constitui um modelo de alguma área de aplicação. Para cada nova área, os objetos ou fenômenos no mundo real têm que ser descritos por objetos dado. Numa linguagem com um número fixo de tipos dado e objetos dado, têm-se que criar de uma forma engenhosa quando codifica-se objetos. Na extensão de uma linguagem, fornece-se com facilidade a adaptação da linguagem a cada nova área. Definindo os novos tipos dado, pode-se codificar e desta forma incrementar a legibilidade dos programas.

A possibilidade de introduzir novos tipos e objetos dado incrementam grande e consideravelmente o poder da modelação de uma linguagem. Portanto, O modelo apresentado nesta seção, dirige-se ao problema de uma função com múltiplos resultados, uma vez que estes têm uma importante e elegante solução, a linguagem de programação Java. Assim, uma função ***f*** é definida por uma *"função construtor"* a qual toma diferentes valores retornados para diferentes padrões na entrada dos parâmetros [NG 95].

A forma geral da *função construtor* é:

$$
\begin{aligned}
& E_1 = f\ P_{11} \ldots P_{1n} \\
\| \ & E_2 = f\ P_{21} \ldots P_{2n} \\
& \vdots \\
\| \ & E_m = f\ P_{m1} \ldots P_{mn}
\end{aligned}
$$

onde; *f* é o nome da função declarada. P_{ij} são os padrões e os E_i são as expressões retornadas para os diferentes casos. O símbolo "||" significa "or" e é usado para separar os diferentes padrões dos parâmetros de entrada [NG 95].

A forma geral da *função construtor* acima pode ser transformada a:

$$
\begin{aligned}
& E_1 = f(P_{11} \ldots P_{1n}) \\
\| \; & E_2 = f(P_{21} \ldots P_{2n}) \\
& \vdots \\
\| \; & E_m = f(P_{m1} \ldots P_{mn})
\end{aligned}
$$

A *função construtor* será embutida dentro de uma classe Java, da seguinte forma:

class Nome_classe_Java{

[Declaração de variáveis;]

.

.

// Declaração da função construtor*

$$
\begin{aligned}
& E_1 = f(P_{11} \ldots P_{1n}) \\
\| \; & E_2 = f(P_{21} \ldots P_{2n}) \\
& \vdots \\
\| \; & E_m = f(P_{m1} \ldots P_{mn});
\end{aligned}
$$

} *//fim da classe Java*

Exemplo **a**:

Definir a função *eval*() que calcula o valor de uma expressão dando como resultado o valor de uma variável simples(Standard ML):

fun *eval value(Const i)* = *i*
| *eval value(Var)* = *value*
| *eval value(Plus(e_1 , e_2))* = *eval value* e_1 + *eval value* e_2
| *eval value (Times(e_1 , e_2))* = *eval value* e_1 * *eval value* e_2 ;

usando o modelo da *função construtor* o exemplo acima pode ser transformado a:

i = *eval value (Const i)*
|| *value* = *eval value (Var)*
|| *eval value* e_1 + *eval value* e_2 = *eval value (Plus* (e_1 , e_2))
|| *eval value* e_1 * *eval value* e_2 = *eval value(Times*(e_1 , e_2));

Exemplo:

Definir uma função para achar a derivada de uma expressão com respeito a uma variável simples (Standard ML):

fun *diff (Const i)* = *Const 0*
| *diff (Var)* = *Const 1*
| *diff (Plus(e_1 , e_2))* = *Plus (diff(e_1), diff(e_2))*
| *diff (Times(e_1 , e_2))* = *Plus(Times(e_1 , diff(e_2)) , Times(diff(e_1) , e_2));*

usando o modelo da *função construtor* o exemplo acima pode ser transformado a:

0 = *diff (Const i)*
|| *1* = *diff (Var)*
|| *Plus (diff(e_1), diff(e_2))* = *diff (Plus(e_1 , e_2))*
|| *Plus(Times(e_1 , diff(e_2)) , Times(diff(e_1) , e_2))* = *diff (Times(e_1 , e_2));*

Devido a que Java suporta overloading, uma classe pode ter qualquer número de funções, todas com o mesmo nome. Baseado no número e tipos dos argumentos que passa-se dentro de uma função, o compilador Java pode determinar qual função será usada, os argumentos de cada função devem diferir em número ou em tipo uns dos outros [ARN 96].

Criar-se-á um pacote chamado "*funcional*" segundo as regras de criação de um "*package*" em Java [SUN 95][SUN 95a]. Os pacotes estão ligados às classes, interfaces e subpacotes [ARN 96], por diversas características.

Exemplo de criação da *classe funcional*, que agrega funções criadas pelo *Construtor de funções Java*:

```
/** Nome do arquivo: funcional.java
   Objetivo: concatena funções criadas pelo Construtor de Funções Java
   Versão: 1.0
   Copyright © 1997, Jorge Juan Zavaleta Gavidia
*/
import java.lang.System;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;

public class funcional{
        public static void main{String args[]) throws IOException{
        SequenceInputStream inStream;
        FileInputStream f1 = new FileInputStream("funcional.java");
        FileInputStream f2 = new FileInputStream("nome_funcao.java");
```

```
        inStream = new SequenceInputStream(f1 , f2);
        boolean eof = false;
        while(!eof) {
                int c = inStream.read();
                if( c ==-1) eof = true;
                else {
                    System.out.print( (char) c);
                    }
        }
        System.out.println("Concatenação de funções ... Ok! ");
        inStream.close();
        f1.close();
        f2.close();
        }// fim do método main()
}// fim da classe funcional
```

Cada arquivo fonte cujas classes e interfaces pertencem ao pacote *funcional* são incorporadas com a declaração *package* da seguinte maneira:

```
package funcional;
```

O pacote *funcional* é importado para usarem suas funções em outro arquivo fonte fornecendo uma declaração *package* no topo do novo arquivo:

```
package funcional;

class Nome_da Clase_Funcional {
     //Declaração de variáveis;
      ...
      // funções de instância pública, que usam funções do pacote funcional;
     ...
     // implementações de novas funções
     ...
} // fim da classe;
```

6 Conclusões

Dos muitos fenômenos no mundo da Internet, a linguagem de programação Java, da Sun Microsystem, é provavelmente a maior promessa a longo prazo. Java é um ambiente razoavelmente seguro agora, e está se tornando mais ainda; embora nenhum ambiente seja totalmente seguro. Mesmo assim, a linguagem oferece um conjunto excelente de recursos para o desenvolvimento de aplicações seguras. Os primeiros indícios são de que Java é um ambiente altamente produtivo para profissionais que entendem de programação orientada a objeto.

Os ganhos ocorrem principalmente porque os ponteiros e o alocamento de memória ficam bastante escondidos do programador. A linguagem é poderosa e elegantemente simples. A produtividade é uma medida da rapidez e facilidade com que ela permite que se desenvolva e mantenha um código, e o crescimento das aplicações Java compiladas que podem ser executadas de forma mais rápida que em alguns ambientes compilados nativamente.

O problema da segurança, para um grande número de empresas esta sendo crucial. Em um determinado nível, para muitas empresas de software e outras de alta tecnologia. Elas estão apostando alto em Java, e isso dará um impulso considerável aos ganhos gerados pelos próprios méritos da linguagem em si. As empresas estão criando sistemas de informação distribuídos além da simples recuperação de documentos em HTML via Internet.

Nesse sentido, a implementação de um *Construtor de Funções Java* na linguagem Java é um exemplo das boas características da linguagem, e comprova as qualidades de concisão e de legibilidade da programação usando Java. A utilização do *Construtor de Funções Java* como ferramenta auxiliar para converter funções Lisp em funções Java será de grande utilidade quando se implementam bibliotecas de funções para serem usadas em programação funcional. O modelo da *função construtor* e o *Construtor de Funções Java* como ferramenta auxiliar permitirão agregar novas características como expressividade, claridade e rapidez, permitindo fazer programação funcional num ambiente orientado a objetos como Java. A integração de dois paradigmas de programação num só seria uma ferramenta muito útil na hora de modelar fenômenos do mundo real, para uma melhor interpretação dos mesmos.

As vantagens de um estilo de programação funcional são muito importantes para o desenvolvimento de software seguro. Portanto, acreditamos fortemente que um dia as linguagens funcionais poderão ser usadas na World Wide Web como uma linguagem de programação de propósito geral.

Por último é conveniente reafirmar os caminhos abertos pelo trabalho. Pesquisas futuras em implementações de linguagens funcionais em Java. A implementação de construtores de funções e interpretadores em outras linguagens funcionais para Java. A utilização dos construtores de funções como ferramentas e como interfaces para Sistemas de Tutores Inteligentes para Ensino à Distância via Internet.

Anexo 1 Exemplos do *LispJ*

Os exemplos mostrados a seguir foram gerados pelo *Construtor de Funções Java* e são mostrados aqui com a finalidade de ajudar a entender seu funcionamento.

nil = lista vazia

```
nil
S-exp:> nil
Valor:>> ()
Função:> () ... nao e uma S_expressao
```

quote

```
'aa
S-exp:> (' aa)
Valor:>> aa
Função:>
        public Lisp_exp QUOTE() {
          return e.two();
          }

'(aa bb cc)
S-exp:> (' (aa bb cc))
Valor:>> (aa bb cc)
Função:> public Lisp_exp QUOTE() {
           return e.two();
           }
```

car e quote

```
'car '(aa bb cc)
S-exp:> (' (car (' (aa bb cc))))
Valor:>> (car (' (aa bb cc)))
Função:> public Lisp_exp QUOTE() {
      return e.two();
      }

car '(aa bb cc)
S-exp:> (car (' (aa bb cc)))
Valor:>> aa
Função:> public Lisp_exp CAR(Lisp_exp x) {
      return x.hd;
      }

car '((a b)c d)
S-exp:> (car (' ((a b) c d)))
Valor:>> (a b)
Função:> public Lisp_exp CAR(Lisp_exp x) {
      return x.hd;
      }

car '(aa)
S-exp:> (car (' (aa)))
Valor:>> aa
```

```
Função:> public Lisp_exp CAR(Lisp_exp x) {
        return x.hd;
        }

car aa
S-exp:> (car aa)
Valor:>> aa
Função:> public Lisp_exp CAR(Lisp_exp x) {
        return x.hd;
        }
```

cdr

```
cdr '(aa bb cc)
S-exp:> (cdr (' (aa bb cc)))
Valor:>> (bb cc)
Função:> public Lisp_exp CDR(Lisp_exp x) {
        return x.tl;
        }

cdr '((a b)c d)
S-exp:> (cdr (' ((a b) c d)))
Valor:>> (c d)
Função:> public Lisp_exp CDR(Lisp_exp x) {
        return x.tl;
        }

cdr '(aa)
S-exp:> (cdr (' (aa)))
Valor:>> ()
Função:> public Lisp_exp CDR(Lisp_exp x) {
        return x.tl;
        }

cdr aa
S-exp:> (cdr aa)
Valor:>> aa
Função:> public Lisp_exp CDR(Lisp_exp x) {
        return x.tl;
        }
```

cadr

```
cadr '(aa bb cc)
S-exp:> (car (cdr (' (aa bb cc))))
Valor:>> bb
Função:> public Lisp_exp CAR(Lisp_exp x) {
        return x.hd;
        }
```

caddr

```
caddr '(jorge carlos julio)
S-exp:> (car (cdr (cdr (' (jorge carlos julio)))))
```

```
Valor:>> julio
Função:> public Lisp_exp CAR(Lisp_exp x) {
        return x.hd;
        }
```

cons

```
cons 'jorge '( vitor andres)
S-exp:> (cons (' jorge) (' (vitor andres)))
Valor:>> (jorge vitor andres)
Função:> public Lisp_exp CONS(Lisp_exp x, Lisp_exp y) {
        return jn(x,y);
         }

cons '(jorge jj)'(andres z)
S-exp:> (cons (' (jorge jj)) (' (andres z)))
Valor:>> ((jorge jj) andres z)
Função:> public Lisp_exp CONS(Lisp_exp x, Lisp_exp y) {
        return jn(x,y);
         }

cons jorge nil
S-exp:> (cons jorge nil)
Valor:>> (jorge)
Função:> public Lisp_exp CONS(Lisp_exp x, Lisp_exp y) {
        return jn(x,y);
         }

cons jorge ()
S-exp:> (cons jorge ())
Valor:>> (jorge)
Função:> public Lisp_exp CONS(Lisp_exp x, Lisp_exp y) {
        return jn(x,y);
         }

cons Jorge Juan
S-exp:> (cons Jorge Juan)
Valor:>> Jorge
Função:> public Lisp_exp CONS(Lisp_exp x, Lisp_exp y) {
        return jn(x,y);
         }
```

atom

```
atom 'Jorge
S-exp:> (atom (' Jorge))
Valor:>> true
Função:> public boolean ATOMO() {
        if (x.at) return true2;
        else return false2;
        }

atom '(Jorge)
S-exp:> (atom (' (Jorge)))
Valor:>> false
Função:> public boolean ATOMO() {
        if (x.at) return true2;
```

```
        else return false2;
        }

atom '( )
S-exp:> (atom (' ()))
Valor:>> true
Função:> public boolean ATOMO() {
        if (x.at) return true2;
        else return false2;
        }
```

Igualdade de S-expressões

```
= jorge juan
S-exp:> (= jorge juan)
Valor:>> false
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }

= jorge jorge
S-exp:> (= jorge jorge)
Valor:>> true
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }

= '(a b)'(a b)
S-exp:> (= (' (a b)) (' (a b)))
Valor:>> true
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }

= '(Jorge Juan)'(Jorge juan)
S-exp:> (= (' (Jorge Juan)) (' (Jorge juan)))
Valor:>> false
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }
```

If ... then ... else

```
if true Jorge Juan
S-exp:> (if true Jorge Juan)
Valor:>> Jorge
Função:> public Lisp_exp IF_THEN_ELSE(){
        Lisp_exp p = eval(e.two(),d);
        if (p.err) return p;
        if (p == false2) return eval(e.four(),d);
        return eval(e.three(),d);
        }
```

```
if false Jorge Juan
S-exp:> (if false Jorge Juan)
Valor:>> Juan
Função:> public Lisp_exp IF_THEN_ELSE(){
        Lisp_exp p = eval(e.two(),d);
        if (p.err) return p;
        if (p == false2) return eval(e.four(),d);
        return eval(e.three(),d);
        }

if xxx jorge Justicia
S-exp:> (if xxx jorge Justicia)
Valor:>> jorge
Função:> public Lisp_exp IF_THEN_ELSE(){
        Lisp_exp p = eval(e.two(),d);
        if (p.err) return p;
        if (p == false2) return eval(e.four(),d);
        return eval(e.three(),d);
        }
```

cdr

```
cdr display cdr display cdr display '(a b c d e)
Display :> (a b c d e).
Display :> (b c d e).
Display :> (c d e).
S-exp:> (cdr (display (cdr (display (cdr (display (' (a b c d e))))))))
Display :> (a b c d e).
Display :> (b c d e).
Display :> (c d e).
Valor:>> (d e)
Função:> public Lisp_exp CDR(Lisp_exp x) {
        return x.tl;
        }
```

lambda

```
('lambda(x y)x 1 2)
S-exp:> ((' (lambda (x y) x)) 1 2)
Valor:>> 1
Função:> 1 ... nao e uma S_expressao

('lambda(x y)y 1 2)
S-exp:> ((' (lambda (x y) y)) 1 2)
Valor:>> 2
Função:> 2 ... nao e uma S_expressao

('lamda(x y)y 1)
S-exp:> ((' lamda) (x y) y 1)
Valor:>> lamda
Função:> lamda ... nao e uma S_expressao

('lambda(x y)y 1)
S-exp:> ((' (lambda (x y) y)) 1)
Valor:>> ()
Função:> () ... nao e uma S_expressao
```

```
('lambda(x y)y 1 2 3)
S-exp:> ((' (lambda (x y) y)) 1 2 3)
Valor:>> 2
Função:> 2 ... nao e uma S_expressao
('lambda(x y)cons y cons x nil 1 2)
S-exp:> ((' (lambda (x y) (cons y (cons x nil)))) 1 2)
Valor:>> (2 1)
Função:> (2 1) ... nao e uma S_expressao
```

let

```
let x a cons x cons x nil
S-exp:> ((' (lambda (x) (cons x (cons x nil)))) a)
Valor:>> (a a)
Função:> (a a) ... nao e uma S_expressao
```

Expressão *x*

```
x
S-exp:> x
Valor:>> x
Função:> x ... nao e uma S_expressao

let (f x) if atom display x x (f car x)(f '(((a)b)c))
Display :> (((a) b) c).
Display :> ((a) b).
Display :> (a).
Display :> a.
S-exp:> ((' (lambda (f) (f (' (((a) b) c)))))  (' (lambda (x) (if (atom (display x)) x (f (car x))))))
Display :> (((a) b) c).
Display :> ((a) b).
Display :> (a).
Display :> a.
Valor:>> a
Função:> a ... nao e uma S_expressao


f
S-exp:> f
Valor:>> f
Função:> f ... nao e uma S_expressao
```

append()

```
append '(jorhge jorge julio)'(d e f)
S-exp:> (append (' (jorhge jorge julio)) (' (d e f)))
Valor:>> (jorhge jorge julio d e f)
Função:> public Lisp_exp FAPPEND(Lisp_exp x, Lisp_exp y) {
        return append(x,y);
         }

append '(Jorge Juan Zavaleta)'(Gavidia II UFRGS)
S-exp:> (append (' (Jorge Juan Zavaleta)) (' (Gavidia II UFRGS)))
Valor:>> (Jorge Juan Zavaleta Gavidia II UFRGS)
Função:> public Lisp_exp FAPPEND(Lisp_exp x, Lisp_exp y) {
```

```
        return append(x,y);
         }

let(cat Jorge Juan) if atom Jorge Juan cons car Jorge (cat cdr Jorge JUan)(cat '(a b c)'(d e f)
S-exp:> ((' (lambda (cat) (cat (' (a b c)) (' (d e f))))) (' (lambda (Jorge Juan) (if (atom Jorge) Juan
(cons (car Jorge) (cat (cdr Jorge) JUan))))))
Valor:>> a
Função:> a ... nao e uma S_expressao
```

define

```
define (cat Jorge Juan)if atom Jorge Juan cons car Jorge (cat cdr Jorge Juan)
Define:> cat to be !!
(lambda (Jorge Juan) (if (atom Jorge) Juan (cons (car Jorge) (cat (cdr Jorge) Juan))))
função:> define ... nao e uma S_expressao
```

cat

```
cat
S-exp:> cat
Valor:>> cat
Função:> cat ... nao e uma S_expressao

define (cat x y) if atom x y cons car x (cat cdr  x y)
Define:> cat to be !!
(lambda (x y) (if (atom x) y (cons (car x) (cat (cdr x) y))))
função:> define ... nao e uma S_expressao

cat
S-exp:> cat
Valor:>> (lambda (x y) (if (atom x) y (cons (car x) (cat (cdr x) y))))
Função:> () ... nao e uma S_expressao
```

size

```
size abc
S-exp:> (size abc)
Valor:>> 3
Função:> public Lisp_exp TAMANHO( Lisp_exp x ) {
            return new Lisp_exp(x.tos().length());
            }

size Jorge Juan Zavaleta Gavidia
S-exp:> (size Jorge)
Valor:>> 5
Função:> public Lisp_exp TAMANHO( Lisp_exp x ) {
            return new Lisp_exp(x.tos().length());
            }

size '(Jorge Juan Zavaleta)
S-exp:> (size (' (Jorge Juan Zavaleta)))
Valor:>> 21
Função:> public Lisp_exp TAMANHO( Lisp_exp x ) {
            return new Lisp_exp(x.tos().length());
            }
```

length

```
length '(Jorge Juan Zavaleta)
S-exp:> (length (' (Jorge Juan Zavaleta)))
Valor:>> 3
Função:> public Lisp_exp COMPRIMENTO( Lisp_exp x ) {
        return new Lisp_exp(count(x));
        }

length display bits 'a
Display :> (0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0).
S-exp:> (length (display (bits (' a))))
Display :> (0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0).
Valor:>> 16
Função:> public Lisp_exp COMPRIMENTO( Lisp_exp x ) {
        return new Lisp_exp(count(x));
        }

length display bits 'abc
Display :> (0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 1 0).
S-exp:> (length (display (bits (' abc))))
Display :> (0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 1 0).
Valor:>> 32
Função:> public Lisp_exp COMPRIMENTO( Lisp_exp x ) {
        return new Lisp_exp(count(x));
        }
```

Soma +

```
+ abc 15
S-exp:> (+ abc 15)
Valor:>> 15
Função:> public Lisp_exp ADIÇÃO(Lisp_exp x, Lisp_exp y) {
        return new Lisp_exp(x.nval + y.nval);
        }

+ '(abc) 150
S-exp:> (+ (' (abc)) 150)
Valor:>> 150
Função:> public Lisp_exp ADIÇÃO(Lisp_exp x, Lisp_exp y) {
        return new Lisp_exp(x.nval + y.nval);
        }

+ 5 59
S-exp:> (+ 5 59)
Valor:>> 64
Função:> public Lisp_exp ADIÇÃO(Lisp_exp x, Lisp_exp y) {
        return new Lisp_exp(x.nval + y.nval);
        }
```

Subtração -

```
- 5 10
S-exp:> (- 5 10)
Valor:>> 0
Função:> public Lisp_exp SUBTRAÇÃO(Lisp_exp x, Lisp_exp y) {
```

```
        if (x.nval <= y.nval) return zero;
        else return new Lisp_exp(x.nval - y.nval);
         }

- 10 5
S-exp:> (- 10 5)
Valor:>> 5
Função:> public Lisp_exp SUBTRAÇÃO(Lisp_exp x, Lisp_exp y) {
        if (x.nval <= y.nval) return zero;
        else return new Lisp_exp(x.nval - y.nval);
         }
```

Multiplicação

```
* 5 6
S-exp:> (* 5 6)
Valor:>> 30
Função:> public Lisp_exp MULTIPLICAÇÃO(Lisp_exp x, Lisp_exp y) {
        return new Lisp_exp(x.nval * y.nval);
        }
```

Potenciação ^

```
^ 5 5
S-exp:> (^ 5 5)
Valor:>> 3125
Função:> public Lisp_exp POTENCIAÇÃO(Lisp_exp x, Lisp_exp y) {
        return new Lisp_exp(power(x.nval,y.nval));
         }

^ 2 5
S-exp:> (^ 2 5)
Valor:>> 32
Função:> public Lisp_exp POTENCIAÇÃO(Lisp_exp x, Lisp_exp y) {
        return new Lisp_exp(power(x.nval,y.nval));
         }
```

Menor <

```
< 4 9
S-exp:> (< 4 9)
Valor:>> true
Função:> public boolean MENOR(Lisp_exp x, Lisp_exp y) {
        if (x.nval < y.nval) return true2;
        else return false2;
         }

< 5 5
S-exp:> (< 5 5)
Valor:>> false
Função:> public boolean MENOR(Lisp_exp x, Lisp_exp y) {
        if (x.nval < y.nval) return true2;
        else return false2;
         }
```

Maior >

```
> 6 5
S-exp:> (> 6 5)
Valor:>> true
Função:> public boolean MAIOR(Lisp_exp x, Lisp_exp y) {
        if (x.nval > y.nval) return true2;
        else return false2;
         }

> 5 5
S-exp:> (> 5 5)
Valor:>> false
Função:> public boolean MAIOR(Lisp_exp x, Lisp_exp y) {
        if (x.nval > y.nval) return true2;
        else return false2;
         }
```

Menor igual (<=)

```
<= 5 3
S-exp:> (<= 5 3)
Valor:>> false
Função:> public boolean MENOR_IG(Lisp_exp x, Lisp_exp y) {
        if (x.nval <= y.nval) return true2;
        else return false2;
         }

<= 5 5
S-exp:> (<= 5 5)
Valor:>> true
Função:> public boolean MENOR_IG(Lisp_exp x, Lisp_exp y) {
        if (x.nval <= y.nval) return true2;
        else return false2;
         }
```

Maior igual (>=)

```
>= 5 8
S-exp:> (>= 5 8)
Valor:>> false
Função:> public boolean MAIOR_IG(Lisp_exp x, Lisp_exp y) {
        if (x.nval >= y.nval) return true2;
        else return false2;
         }

>= 10 10
S-exp:> (>= 10 10)
Valor:>> true
Função:> public boolean MAIOR_IG(Lisp_exp x, Lisp_exp y) {
        if (x.nval >= y.nval) return true2;
        else return false2;
         }
```

Igual (=)

```
= 6 5
S-exp:> (= 6 5)
Valor:>> false
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }
= 6 6
S-exp:> (= 6 6)
Valor:>> true
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }
= abc 0
S-exp:> (= abc 0)
Valor:>> false
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }

= 0003 3
S-exp:> (= 3 3)
Valor:>> true
Função:> public boolean IGUAL(Lisp_exp x, Lisp_exp y) {
        if (eq(x,y)) return true2;
        else return false2;
         }
```

Valores

```
0099
S-exp:> 99
Valor:>> 99
Função:> 99 ... nao e uma S_expressao

let 99 45 cons 99 cons 99 nil
S-exp:> ((' (lambda (99) (cons 99 (cons 99 nil)))) 45)
Valor:>> (99 99)
Função:> (99 99) ... nao e uma S_expressao


define 99 10
Define:> 99 to be !!
10
função:> define ... nao e uma S_expressao
```

eval

```
eval display '+ display 5 display 15
Display :> (+ (display 5) (display 15)).
S-exp:> (eval (display (' (+ (display 5) (display 15)))))
Display :> (+ (display 5) (display 15)).
Display :> 5.
Display :> 15.
```

```
Valor:>> 20
Função:> public Lisp_exp FEVAL() {
        push_env();
        v = eval(x,d);
        pop_env();
        return v;
         }
```

try

```
try 0 display '+ display 5 display 15 nil
Display :> (+ (display 5) (display 15)).
S-exp:> (try 0 (display (' (+ (display 5) (display 15)))) nil)
Display :> (+ (display 5) (display 15)).
Valor:>> (sucesso 20 (5 15))
Função:> public Lisp_exp FTRY() {
        binary_data.push(z);
        suppress_display.push(true2);
        suppressed_displays.push(nil);
        long xx = x.nval;
        if (x == no_time_limit) xx = infinity;
        push_env();
        if(xx < d)
                v = eval(y,xx);
        else
                v = eval(y,d);
        pop_env();
        Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
        suppress_display.pop();
        binary_data.pop();
        if(v == data_err) return
                jn(failure,jn(out_of_data,jn(sd_ds,nil)));
        if(v !== time_err) return
                jn(success,jn(v,jn(sd_ds,nil)));
        if ( xx < d)  return
                jn(failure,jn(out_of_time,jn(sd_ds,nil)));
        else
                return time_err;
         }


define 5!
let (f x) if = display x 0 1*x(f - x 1)(f 5)
Display :> 5.
Display :> 4.
Display :> 3.
Display :> 2.
Display :> 1.
Display :> 0.
Define:> 5! to be !!
((' (lambda (f) (f 5))) (' (lambda (x) (if (= (display x) 0) 1*x (f (- x 1))))))
função:> define ... nao e uma S_expressao

eval 5!
S-exp:> (eval 5!)
```

```
Display :> 5.
Display :> 4.
Display :> 3.
Display :> 2.
Display :> 1.
Display :> 0.
Valor:>> 120
Função:> public Lisp_exp FEVAL() {
        push_env();
        v = eval(x,d);
        pop_env();
        return v;
         }

try 0 5! nil
S-exp:> (try 0 5! nil)
Valor:>> (falha fora do tempo ())
Função:> public Lisp_exp FTRY() {
        binary_data.push(z);
        suppress_display.push(true2);
        suppressed_displays.push(nil);
        long xx = x.nval;
        if (x == no_time_limit) xx = infinity;
        push_env();
        if(xx < d)
                v = eval(y,xx);
        else
                v = eval(y,d);
        pop_env();
        Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
        suppress_display.pop();
        binary_data.pop();
        if(v == data_err) return
                jn(failure,jn(out_of_data,jn(sd_ds,nil)));
        if(v !== time_err) return
                jn(success,jn(v,jn(sd_ds,nil)));
        if ( xx < d)  return
                jn(failure,jn(out_of_time,jn(sd_ds,nil)));
        else
                return time_err;
         }

try 0 'read-bit nil
S-exp:> (try 0 (' (read-bit)) nil)
Valor:>> (falha fora dos dados ())
Função:> public Lisp_exp FTRY() {
        binary_data.push(z);
        suppress_display.push(true2);
        suppressed_displays.push(nil);
        long xx = x.nval;
        if (x == no_time_limit) xx = infinity;
        push_env();
        if(xx < d)
                v = eval(y,xx);
        else
                v = eval(y,d);
        pop_env();
        Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
```

```
            suppress_display.pop();
            binary_data.pop();
            if(v == data_err) return
                    jn(failure,jn(out_of_data,jn(sd_ds,nil)));
            if(v !== time_err) return
                    jn(success,jn(v,jn(sd_ds,nil)));
            if ( xx < d)  return
                    jn(failure,jn(out_of_time,jn(sd_ds,nil)));
            else
                    return time_err;
             }

bits 'a
S-exp:> (bits (' a))
Valor:>> (0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0)
Função:> public Lisp_exp BITS( Lisp_exp x ) {
            return to_bits(x);
            }

try 0'read-exp '(0 1 1 0 0 0 0 1)
S-exp:> (try 0 (' (read-exp)) (' (0 1 1 0 0 0 0 1)))
Valor:>> (falha fora dos dados ())
Função:> public Lisp_exp FTRY() {
            binary_data.push(z);
            suppress_display.push(true2);
            suppressed_displays.push(nil);
            long xx = x.nval;
            if (x == no_time_limit) xx = infinity;
            push_env();
            if(xx < d)
                    v = eval(y,xx);
            else
                    v = eval(y,d);
            pop_env();
            Lisp_exp sd_ds = (Lisp_exp) suppressed_displays.pop();
            suppress_display.pop();
            binary_data.pop();
            if(v == data_err) return
                    jn(failure,jn(out_of_data,jn(sd_ds,nil)));
            if(v !== time_err) return
                    jn(success,jn(v,jn(sd_ds,nil)));
            if ( xx < d)  return
                    jn(failure,jn(out_of_time,jn(sd_ds,nil)));
            else
                    return time_err;
             }
```

bits

```
bits ()
S-exp:> (bits ())
Valor:>> (0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 1 0 1 0)
Função:> public Lisp_exp BITS( Lisp_exp x ) {
            return to_bits(x);
            }
```

Glossário

Ambiente: O conjunto completo de recurso de hardware e software colocados à disposição do usuário de um sistema.

Applet: Um programa Java que pode ser incluído em uma página HTML com o elemento <APPLET> e pode ser visualizado em um browser compatível com Java.

Browser: Programa aplicativo que permite pesquisar informações na World Wide Web, carregando documentos, imagens e arquivos correlatos. Também chamado de *navegador* Web.

Cliente Servidor: Arquitetura de computação que distribui o processamento entre os clientes e servidores da rede. Os clientes solicitam informação dos servidores. Os servidores armazenam dados e programas, e fornecem serviços da rede aos clientes. Essa organização explora com eficiência a capacidade de computação disponível dividindo a aplicação em dois componentes distintos: um cliente e um servidor.

Hipertexto: Uma metáfora para a apresentação de informações nas quais textos, imagens, sons e ações fiquem interligados em uma teia complexa e não linear de associações que permitem ao usuário percorrer assuntos interrelacionados independentemente da ordem em que os tópicos são apresentados.

Host: O computador central, ou computador de controle, de um ambiente de processamento em rede ou distribuído, que fornece serviços acessados por outros computadores ou terminais através da rede.

Internet: Rede de computadores distribuídos globalmente que trocam informação via protocolo TCP/IP.

Multimídia: A associação de som, gráficos, animação e vídeo. No mundo dos computadores, a multimídia é um subconjunto da hipermídia, que combina os elementos da multimídia como o hipertexto, o qual faz a vinculação das informações.

Página: Um arquivo simples de HTML.

Site: Jargão da Internet, corresponde a um endereço Internet na World Wide Web. Denota um computador host que possui uma página de apresentação WWW.

String: Seqüência de caracteres literais ou de texto.

URL: Acrossemia de Uniform Resource Locator, denominação genérica de um endereço lógico na Internet.

Bibliografia

[AND 87] ANDERSON, John R.; CORBETT, Albert T.; REISER, Brian J. **Essential LISP**. Reading, MA.: Addison-Wesley Publishing Company Inc., 1987.

[AUG 94] AUGUSTIN, Iara. **pEiffel:Uma biblioteca de classes para paralelismo em Eiffel**. Porto Alegre: CPGCC da UFRGS, 1994.

[ARN 96] ARNOLD, Ken; GOSLING, James. **The Java Programming Language**. Reading, MA.:Addison-Wesley Publishing Company Inc., 1996.

[BIR 88] BIRD, Richard; WADLER, Philip. **Introduction to Functional Programming**. New York: Prentice Hall International Series in Computer Science, 1988.

[BOO 94] BOOCH, G. Design an application framework. **Dr. Dobbs J.**, New York, v.19, n.2, p.24-32, 1994.

[CAV 94] CAVALHERO, Gerson Geraldo H. **Um modelo para linguagens Orientadas a Objetos distribuído**. Porto Alegre: CPGCC da UFRGS, 1994.

[CEJ 95] CEJTIN, Henry; JAGANNATHAN, Suresh; KELSEY, Richard. Higher-Order Distributed Objects. **ACM Transactions on Programming Languages and Systems**, New York, v.17, n.5, p. 704-739, 1995.

[CHA 96] CHAITIN, G. J. **An Invitation to Algorithmic Information Theory**. Disponível por WWW em Http://www.research.ibm.com/ people/c/ chaitin. (Setembro, 1996).

[CHA 96a] CHAITIN, G. J. **The Limits of Mathematics**. Disponível por WWW em Http://www.research.ibm.com/people/c/chaitin (Setembro, 1996).

[DEC 95] DECEMBER, J. **Presenting Java**. Indianapolis, IN: Sams.net Publishing, 1995.

[FIE 88] FIELD, Anthony J.; HARRISON, Peter G. **Functional Programming**. Wokingham, England: Addison-Wesley Publishing Company, 1988.

[FLA 96] FLANAGAM, D. **Java in a NutShell**. Sebastopol, CA: O'Reilly & Associates, Inc., 1996.

[GEY 86] GEYER, Claudio Fernando Resin. **Implementação de um Interpretador para uma Linguagem Funcional com Coletor de Lixo Concorrente**. Porto Alegre: CPGCC da UFRGS, 1986.

[GLA 84] GLASER, Hugh; HANKIN, Chris; TILL, David. **Principles of Functional Programming**. Englewood cliffs, NJ.: Prentice-Hall International Inc., 1984.

[GUN 92] GUNTER, Carl A . **Semantics of Programming languages:** Structures and Techniques. London, England: The MIT Press Cambridge, 1992.

[HAS 84] HASEMER, Tony. **Looking at LISP**. Reading, CA.: Addison-Wesley Publishing Company, 1984.

[HER 80] HERNDERSON, P. **Functional Programming:** Application and Implementation. Englewood Cliffs, NJ.: Prentice-Hall International Series in Computer Science, 1980.

[JAW 96] JAWORSKI, Jamie. **Java Developer's Guide**. Indianapolis, IN.: Sam.net, 1996.

[MAC 87] MACLENNAN, Bruce J. **Principles of Programming Languages:** Design, Evaluation, and Implementation. New York: Holt, Rinehart and Winston, 1987.

[MAR 95] MARTINS, Joyce. **Uma Linguagem de Especificação formal Orientada a Objetos**. Porto Alegre: CPGCC da UFRGS, 1995.

[MAU 72] MAURER, W. D. **The Programmer's Instroduction to Lisp**. New York: Macdonald London and American Elsevier Inc., 1972.

[NG 95] NG, K.W.; CHI, Keung. I+: A Multiparadigm Language for objet oriented declarative programming. **Computer Languages**, Washington, v.21, n.2. p. 81-100, 1995.

[PAL 95] PALSBERG, Jens; XIAO, Cun; LIEBERHERR, Karl. Efficient Implementation of Adaptive Software. **ACM Transactions on Programming Languages and Systems**, New York, v.17, n.2, p. 264-292, 1995.

[PLA 93] PLASMEIJER, Rinus; VAN EEKELEN, Marko. **Functional Programming and Parallel Graph Rewriting**. Wokingham, England: Addison-Wesley, 1993.

[SLO 95] SLOANE, Anthony M. An Evaluation of an Automatically Generated Compiler. **ACM Transactions on Programming Languages and Systems**, New York, v.17, n.5, p.693-703, 1995

[SUN 95] SUN Microsystem 1995, **Writing Java Programs**. Disponível por WWW em http://java.sun.com/doc/programmer.html (Dezembro 1995).

[SUN 95a] SUN Microsystem 1995, **The Java Language Tutorial**. Disponível por WWW em http://java.sun.com/tutorial/index.html (Dezembro 1995).

[SUN 95b] SUN Microsystem 1995, **The Java Language Environment**: A White Paper. Disponível por WWW em http://java.sun.com (Dezembro 1995).

[WEI 67] WEISSMAN, Clark. **Lisp 1.5 Primer**. Belmont, CA.: Dickenson Publishing Company Inc., 1967.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Programação Funcional Usando Java
por

Jorge Juan Zavaleta Gavidia

Dissertação apresentada aos Senhores:

Profa. Dra. Karin Becker (PUCRS)

Profa. Dra. Maria Lúcia Blanck Lisbôa

Prof. Dr. Daltro José Nunes

Vista e permitida a impressão.
Porto Alegre, 10/07/97.

Prof. Dr. Paulo Alberto de Azeredo.
Orientador.

Prof. Flávio Rech Wagner
Coordenador do Curso de Pós-Graduação
em Ciência da Computação
Instituto de Informática - UFRGS