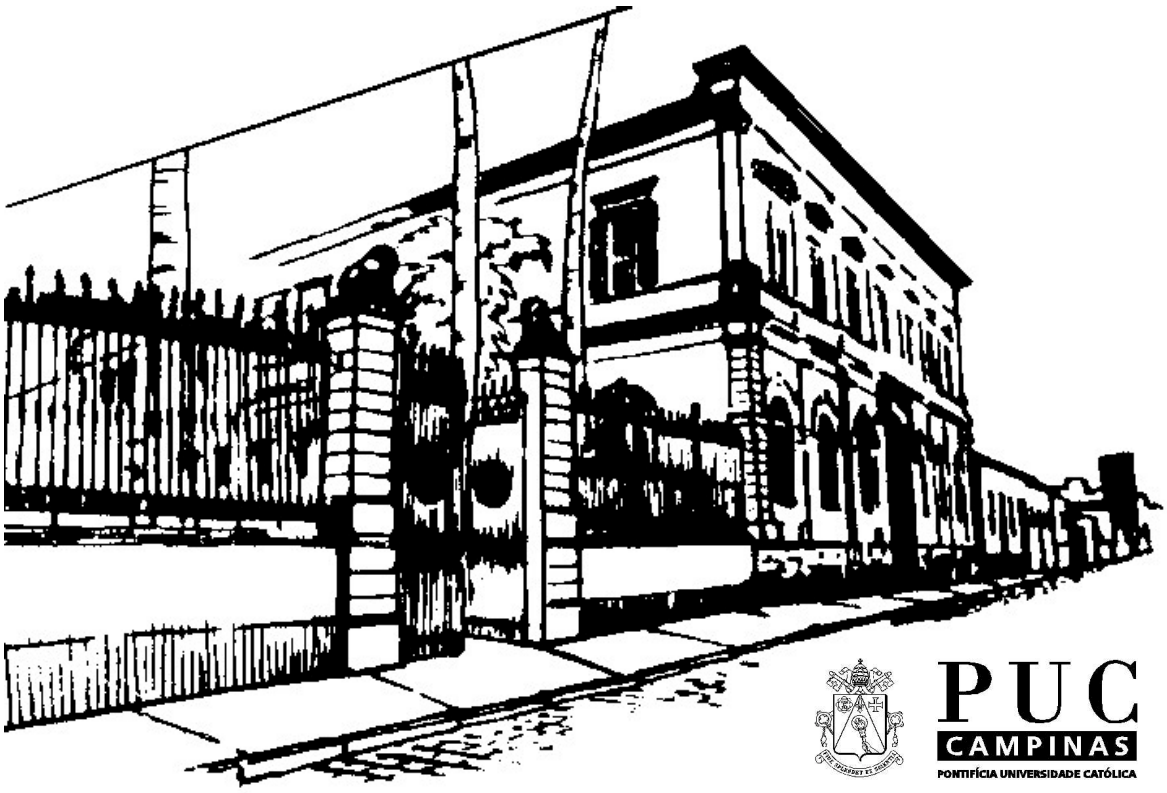

Centro de Ciências Exatas, Ambientais e de Tecnologias

Faculdade de Engenharia de Computação

Paradigmas de Linguagens de Programação I

1º Semestre de 2003
Volume 2

Prof. André Luís dos R.G. de Carvalho

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS
INSTITUTO DE INFORMÁTICA
PLANO DE ENSINO

Faculdade:	Disciplina:	Docente:	C.h. Semanal	Período Letivo:
Engenharia de Computação	Paradigmas de Linguagens de Programação I	André Luís dos Reis Gomes de Carvalho	06 (quatro horas aula)	1º Semestre de 2003

Objetivo Geral da Disciplina:

- Estudar o paradigma de programação orientada a objetos, bem como uma linguagem representativa deste paradigma.
- Estudar o meta-paradigma declarativo, e suas variantes, o paradigma funcional e o paradigma lógico, bem como linguagens representativas de cada um destes paradigmas.

Avaliação:

- 70% da nota será dada pela média ponderada de duas provas, respectivamente com pesos 2 e 1.
- 30% da nota será dada pela média ponderada de uma série de trabalhos a serem desenvolvidos durante o semestre, com pesos compatíveis com sua complexidade.

Freqüência:

- Nenhum abono de falta será concedido pelo professor. Qualquer problema neste sentido deverá ser encaminhada ao PA que tomará todas as providências no sentido de encaminhar a solução do mesmo.
- Para fins de controle de freqüência, não será permitido que alunos desta turma assistam aulas em outras turma, e nem o contrário.

Bibliografia Utilizada :

- Concepts of Programming Languages
  Sebesta, R.W.
- Conceitos de Linguagens de Programação
  Ghesi, C.; e Jazayeri, M.
- Programming Languages: Design and Implementation
  Pratt, T.W.
- Object Orientede System Analysis: Modelling the World in Data
  Shlaer, S.; e Mellor, S.J.
- Object Lifecycles: Modelling the World in States
  Shlaer, S.; e Mellor, S.J.
- Introdução à Programação Orientada a Objetos
  Takahashi, T.
- Programação Orientada a Objetos
  Takahashi T.; e Liesemberg, H.K.
- Programação Orientada para Objeto
  Cox, B.J.
- The TAO of Objects: A Beginner's Guide to Object Oriented Programming
  Entsminger, G.
- A Complete Object Oriented Design Example
  Richardson, J.E.; Schulz, R.C.; e Berard, E.V.
- Java – Como Programar
  Deitel, H.M. e Deitel, P.J.
- Java Unleashed
  Morrison, M.; December, J.; et alii
- Dominando o Java
  Naughton, P.Introdução à Programação Funcional
  Meira, S.R.L.
- Apprendre LISP
  Gnosis
- The XLISP Primer
  Bonnie J.F.
- Programação em Lógica e a Linguagem PROLOG
  Casanova, M.A.; Giorno, F.A.c.; e Furtado, A.L.
- Using Turbo Prolog
  Robinson, P.R.
- Using Turbo PROLOG
  Robinson, P.R.
- PROLOG Programming for Artificial Intelligence
  Bratko, I.
- Artigos selecionados e disponibilizados para xerox pelo professor.

Nº aulas	Conteúdos selecionados

04	O PARADIGMA DE ORIENTAÇÃO A OBJETOS:
	- INTRODUÇÃO;
	- CONCEITOS BÁSICOS:
	  - Objeto → (1) estado interno; (2) comportamento: métodos / mensagens.
	  - Classes → (1) estado; (2) comportamento: métodos / mensagens; (3) instâncias de classe.
	- FUNDAMENTOS:
	  - Encapsulamento.
	  - Herança.
	  - Polimorfismo.
	- CONCLUSÃO.

20	A LINGUAGEM DE PROGRAMAÇÃO JAVA (ASPECTOS BÁSICOS):
	
	- INTRODUÇÃO À JAVA
	  - Execução de Conteúdo → (1) o que pode-se fazer com Java: o que é java / o que é execução de conteúdo / como Java muda a WWW; (2) origens e futuro de Java: o estado corrente de Java / as possibilidades futuras de Java; (3) potencial da linguagem Java: animação / interação / interatividade e computação / comunicação / aplicações e handlers; (4) o que se torna possível com Java.
	  - Projeto Flexível e Dinâmico → (1) primeiro contato com Java: conexao com a WWW / alguns programas simples; (2) visão geral de Java: suporte a comunicação de rede / características de Java enquanto uma linguagem de programação / HotJava / Java em ação / componentes de software de Java / especificação da máquina virtual de Java / segurança em Java.
	  - Impactos na WWW → (1) visão geral da WWW: ideias que levaram à WWW / uma definição de WWW; (2) como Java transforma a WWW: suporte à interatividade / eliminação da necessidade aplicações auxiliares; (3) contribuições à comunicação na WWW; (4) impactos no potencial da WWW.
	  - Páginas Animadas → (1) applets em movimento; (2) animação; (3) sites comerciais.
	  - Páginas Interativas → (1) jogos interativos; (2) aplicações educacionais.
	  - Distribuição de Conteúdo → (1) significado de distribuição e recuperação em rede; (2) como lidar com novos protocolos e formatos; (3) recuperação e compartilhamento de informações em rede.
	
	- PRIMEIRO CONTATO

	- Ferramentas → (1) visão geral; (2) navegadors; (3) ambientes de desenvolvimento; (4) bibliotecas de programação; (5) recursos online. - O Kit de Desenvolvimento Java → (1) como obter a última versão; (2) visão geral; (3) o compilador; (4) o interpretador para a execução; (5) o visualizador de applets; (6) o depurador; (7) engenharia reversa em arquivos de classe; (8) o gerador de arquivos header e strub; (9) o gerador de documentação. - Outros Ambientes e Ferramentas → (1) ambientes de desenvolvimento; (2) bibliotecas para programação. - A LINGUAGEM DE PROGRAMAÇÃO JAVA - Fundamentos da Linguagem Java → (1) um primeor programa; (2) tokens: identificadores / palavras chave / literais / operadores / separadores / comentarios / espaços em branco; (3) tipos de dados: inteiros / reais / logico / caractere; (4) conversão de tipo; (5) blocos e escopo; (6) vetores; (7) strings. - Expressões, Operadores e Estruturas de Controle → (1) expressões e operadores: precedência de operadores / operadores de inteiros / operadores de reais / operadores lógicos / operadores de strings / o operador de atribuição; (2) estruturas de controle: seleções / repetições / break e continue. - Classes, Pacotes e Interfaces → (1) revisão de conceitos de programação orientada a objetos: objetos / encapsulamento / mensagens / classes / herança; (2) classes: declaração / derivação / redefinição de métodos / sobrecarga de métodos / modificadores de acesso / classes e métodos abstratos; (2) criação de objetos: o método de criação / o operador new; (3) destruição de objetos; (4) pacotes: declaração / importação / visibilidade das classes; (5) interfaces: declaração / implementação.
02	PRIMEIRA PROVA
22	A LINGUAGEM DE PROGRAMAÇÃO JAVA (ASPECTOS AVANÇADOS): - RESISTÊNCIA A FALHAS - Tratamento de Excessões → (1) programação em alto nível de abstração; (2) programação em baixo nível de abstração; (3) limitações do programador; (4) a cláusula finally. - AS BIBLIOTECAS DE CLASSES DA LINGUAGEM JAVA - Visão Geral das Bibliotecas de Classes → (1) o pacote Language; (2) o pacote Utilities; (3) o pacote I/O; (4) classes relacionadas com threads; (10) classes relacionadas com tratamento de erros; (5) classes relacionadas com processos. - O Pacote Language → (1) a classe Object; (2) classes relacionadas com os tipos de dados: a classe Boolean / a classe character / classes relacionadas com inteiros / classes relacionadas com reais; (3) a classe Math; (4) classes relacionadas com Strings: a classe String / a classe

	StringBuffer; (5) a classe System; (6) a classe Runtime; (7) a classe Class; (8) a classe ClassLoader. - O Pacote Utilities → (1) interfaces: enumeration / observer; (2) classes: BitSet / Date / Random / StringTokenizer / Vector / Stack / Dictionary / Hashtable / Properties / Observable. - O Pacote I/O → (1) classes de entrada: a classe InputSream / o objeto System.in / a classe BufferedInputStream / a classe DataInputStream / a classe FileInputStream / a classe StringBufferInputStream; (2) classes de saída: a classe OutputStream / a classe PrintSream / o objeto System.out / a classe BufferedOutputStream / a classe DataOutputSream / a classe FileOutputStream; (3) classes relacionadas com arquivos: a classe File / a classe RandomAccessFile. - PROGRAMAÇÃO DE APPLETS - Visão Geral de Programação de Applets → (1) o que é uma applet: applets e a WWW / diferença entre applets e aplicações; (2) os limites das applets: limites funcionais / limites impostos pelo navegador; (3) noções básicas sobre applets: herança da classe Applet / HTML; (4) exemplos básicos de applets. - O Pacote AWT (abstract window toolkit) → (1) uma applet AWT simples; (2) tratamento de eventos: tratamento de eventos em detalhes / handleEvent () ou action () / geração de eventos; (3) componentes: componentes de interface / containers / métodos comuns a todos os componentes; (4) como projetar uma interface de usuário. - O Pacote *Applets e Gráficos* → (1) características das applets; o ciclo de vida de uma applet: init () / start () / stop () / destroy (); (2) como explorar o navegador: como localizar arquivos / imagens / como usar o MediaTracker / audio; (3) contextos de uma applet: showDocument () / showStatus () / como obter parâmetros; (4) gráficos; (5) uma applet simples. - Como programar Applets → (1) projeto básico de applets: interface do usuário / projeto das classes; (2) applets no munto real: applets devem ser pequenas / como responder ao usuário. - MULTIPROGRAMAÇÃO - Threads e Multithreading → (1) o que são threads e para que elas servem; (2) como escrever applets com threads; (3) o problema do conceito de paralelismo; (4) como pensar em termos de multithreads; (5) como criar e usar threads; (6) como saber que uma thread parou; (7) thread scheduling: preemptivo X não preemptivo; como testar. - ACESSO A BASES DE DADOS - O Pacote SQL → (1) drivers; (2) classes: Connection, Statement, ResultSet. - PROGRAMAÇÃO DISTRIBUÍDA - ServerSockets e Sockets.

Nº aulas	Conteúdos selecionados
2	- Meta-Paradígma de programação declarativo (objetivos, limitações, considerações de hardware, considerações de software, motivação).
2	- Paradígma de programação funcional (funções matemáticas, funções matemáticas vs. funções de linguagens de programação, características das linguagens funcionais).
8	- M-Expressões (átomos, S-expressões, listas, funções primitivas, predicados, formas, expressões condicionais e regras de avaliação, definição de funções, invocação de funções).
2	- Linguagem de programação Lisp
2	- Paradígma de programação lógico (o que é, bancos de conhecimento, resolvedores de problemas, raciocínio reverso, lógica simbólica formal, cálculo de predicados, proposições simples, proposições compostas, quantificadores, forma clausal, resolução, cláusulas de Horn).
8	- Linguagem de programação Prolog (objetos, relações, fatos, regras, o ambiente turbo, divisões de um programa, depuração, comentários, predicados, domínios, execução, operadores relacionais, operadores aritméticos, funções, recursão, estratégia de busca, .predicados especiais, instanciação e binding, functors, listas).

Índice

Capítulo III:
O Paradigma Declarativo

Introdução

Sabemos que o paradigma imperativo coloca o programador na posição de alguém que manda (e, naturalmente, espera ser obedecido) e cujas ordens se agrupam para descrever o processo de atingir a solução de um determinado problema.

O paradigma declarativo tem esse nome porque, diferentemente do paradigma imperativo, coloca o programador na posição de alguém que declara o que é resolver um determinado problema, e não como resolvê-lo.

Surge com ele a chamada quarta geração de linguagens. E vale lembrar que enquadramos na primeira geração de linguagens as linguagens de máquina, na segunda geração de linguagens as linguagens de montagem, e na terceira geração de linguagens as chamadas linguagens de alto nível, ou, em outras palavras, a grande maioria das linguagens de uso corrente, como por exemplo, Pascal, C, Fortran, Cobol, Clipper, entre outras.

Objetivos

O paradigma declarativo tem como principal objetivo melhorar o tempo despendido na geração de um programa, possibilitando maior produtividade dos programadores já que traz a possibilidade de uma programação de altíssimo nível.

Sabemos que quase todas as coisas têm um lado bom e um lado mal. O mesmo se da com o levantamento do nível das linguagens. A simples observação das gerações de linguagens nos basta para, extrapolando, imaginar as implicações desta quarta geração de linguagens. São elas: (1) linguagens específicas e apropriadas para um domínio restrito de aplicações; e (2) linguagens que geram programas que cada vez menos uso eficiente da máquina fazem e que exigem para terem um tempo de execução aceitável, máquinas de mais alto desempenho.

Características

Como já sabemos, o paradigma declarativo coloca o programador na posição de alguém que declara o que é resolver um determinado problema, e não como resolvê-lo.

Outras aplicações também permitem a especificação de um problemas em um nível alto de abstração, ou, em outras palavras, num nível em que fica caracterizado o que fazer e não como fazer. Ferramentas CASE são um exemplo do que falamos.

Só que nesse tipo de ferramenta, que tem apenas o compromisso de produzir uma especificação, e não o de produzir um programa executável, não existe uma preocupação exagerada com questões de precisão e de não ambigüidade.

Com as linguagens declarativas as coisas se passam de forma diversa: as especificações feitas nelas precisam poder ser executadas, e por isso precisam ser precisas.

Uma das linguagens mais precisas e menos ambíguas que há é a matemática. Daí vem a tendência que toda linguagem declarativa tem de fundamentar-se na matemática. Um exemplo do que dizemos é PROLOG, uma linguagem declarativa que se fundamenta em lógica formal (dizemos que segue o paradigma lógico). Outro exemplo é LISP, uma linguagem declarativa que se fundamenta em funções matemáticas (dizemos que segue o paradigma funcional). Um último exemplo é SQL, uma linguagem declarativa que se fundamenta em álgebra relacional.

As linguagens declarativas exibem propriedades muito interessantes. Uma delas é a transparência referencial. Em linguagens que têm transparência referencial o conceito de variável inexiste ou difere profundamente do conceito convencional no sentido de que, mesmo que existam variáveis em um programa, estas não são controladas explicitamente pelo programador na sua programação, e sim pelo próprio ambiente de execução da linguagem.

Outra propriedade é a transparência de fluxo de execução, ou, em outras palavras, o caminho da execução do programa é transparente para o programador.

Essas características são muito interessantes porque tornam independentes as partes constituintes dos programas, o que leva as linguagens declarativas a serem inerentemente paralelas, ou seja, nelas, construir um programa paralelo é absolutamente natural, tão natural que o programador nem precisa intervir de forma explícita, a paralelização dos processos pode ser feita de forma automática pelo próprio ambiente de execução da linguagem.

Atente para a importância do que dizemos. Com a tecnologia de hardware evoluindo, com a surgimento das máquinas com múltiplos processadores, a existência de linguagens que

possibilitam a criação de programas paralelos de forma tão natural e simples é de extrema relevância.

Conclusão

O paradigma declarativo não é um paradigma novo. Conhece-se linguagens que o representam há pelo menos 40 anos. Apesar de ser um paradigma antigo, podemos dizer sem medo de errar que, nestes 40 anos, as linguagens declarativas jamais alcançaram níveis consideráveis de uso em aplicações reais. Uma exceção talvez seja a linguagem SQL e outras linguagens de consulta.

A colocação acima poderia nos levar a questionar: porque afinal estudamos linguagens declarativas? A resposta e simples! Apesar de antigas, as linguagens declarativas não alcançaram a preferência dos programadores de sistemas, não por serem desinteressantes, e sim por serem muito ineficientes computacionalmente, enquanto as linguagens imperativas, por exemplo, não apresentam essa desvantagem.

A maior eficiência das linguagens imperativas pode ser explicada pela íntima relação que guardam com a arquitetura das máquinas convencionais, a chamada arquitetura von Neuman. Não exploraremos com maior vagar a questão desta relação neste texto por fugir esta discussão do escopo deste texto.

Procuraremos discutir a seguir as seguintes questões: (1) Eficiência computacional é ainda um aspecto tão importante? (2) As linguagens declarativas sempre poderão ser consideradas ineficientes? (3) As linguagens declarativas são de fato artigos de museu e merecem ser esquecidas?

Comecemos pelo princípio. Eficiência computacional é ainda um aspecto tão importante? Raciocinemos! A importância atribuída à eficiência computacional está relacionada a questão do interesse que se tem em aplicações computacionais que executem num tempo aceitável.

Mas, nas modernas máquinas de que dispomos na atualidade, mesmo aplicações muito ineficientes computacionalmente executam em um tempo bastante aceitável.

Vale lembrar ainda que o preço dessas máquinas cai vertiginosamente, enquanto o mesmo não acontece com o preço da mão de obra especializada. Desta forma, melhorar a produtividade é muito interessante, mesmo que isto custe o sacrifício da eficiência computacional.

Consideremos agora a questão seguinte. As linguagens declarativas sempre poderão ser consideradas ineficientes? Na verdade, a resposta para esta questão é não. As linguagens declarativas são ineficientes em máquinas de arquitetura convencional, mas hoje em dia já existem arquiteturas que fogem deste convencional.

Sabemos que, em máquinas paralelas, as linguagens declarativas se tornam muito interessantes por duas razões. A primeira é que a velocidade das máquinas paralela torna aceitável o tempo de execução de programas declarativos. E a segunda é que as linguagens declarativas, por serem naturalmente paralelas, possibilitam explorar com mais naturalidade este tipo de máquina.

Linguagens declarativas e máquinas paralelas formam um par perfeito, assim como as linguagens convencionais formam um par perfeito com as máquinas convencionais.

Assim sendo, a resposta para nossa terceira questão parece simples. As linguagens declarativas são de fato artigos de museu e merecem ser esquecidas? A resposta claramente é não! Podemos dizer que as máquinas de quinta geração vão colocar as linguagens declarativas em seu próprio tempo.

Capítulo IV:
O Paradigma Funcional

Introdução

O paradigma funcional se baseia em funções e composições de funções matemáticas, e constitui base de projeto de uma das mais importantes classes de linguagens de programação não imperativas [Sebesta89].

O objetivo de uma linguagem funcional é imitar, de forma mais perfeita possível, as funções matemáticas. Como resultado, a abordagem à solução de problemas via uma linguagem funcional é fundamentalmente diferente da abordagem via uma linguagem imperativa.

Do mesmo modo que, em linguagens imperativas, um programa é constituído pela combinação de comandos de entrada e saída, de atribuição, e de controle de fluxo de execução, em linguagens funcionais, um programa é constituído por uma série de definições de função e de definições de aplicação de função. A execução de um programa funcional consiste da avaliação das aplicações de função especificadas.

Toda linguagem funcional provê ao programador um conjunto de funções primitivas, um conjunto de formas funcionais para a construção de funções mais elaboradas, uma operação de aplicação de função, e algumas estruturas de dados. Em linguagens funcionais bem definidas existe apenas um pequeno número de funções primitivas [Sebesta89].

Em um programa funcional, funções complicadas são sempre escritas a partir de funções mais simples, e estas últimas, a partir de funções ainda mais simples, até estarem envolvidas apenas funções primitivas da linguagem [Ghezzi/Jazayeri82].

Linguagens puramente funcionais não possuem nenhuma das estruturas fundamentais de uma linguagem imperativa. Desta forma, não se encontra nelas nem variáveis, nem comandos de atribuição, nem estruturas iterativas, o que faz com que o programador deixe de ter que se preocupar com células de memória e com a forma da execução dos programas.

Embora linguagens de programação funcionais possam ser compiladas, normalmente elas são implementadas como interpretadores.

Funções Matemáticas

Uma função consiste de um mapeamento de elementos de um conjunto em elementos de outro conjunto. Possuem três componentes básicos: o domínio, o contradomínio, e a definição. Podem ou não ter nome.

O domínio representa o conjunto dos objetos sobre os quais a função pode ser aplicada. Note que o domínio pode ser o produto cartesiano de diversos conjuntos.

O contradomínio representa o conjunto dos objetos que podem resultar da aplicação da função. E por fim, a definição é a especificação de como um elemento do contradomínio é determinado a partir de um elemento do domínio.

A título de ilustração, encontra-se definida abaixo a função **`Quadrado`**, que toma um número real e o associa a seu quadrado. A notação utilizada é a notação matemática.

`Quadrado:` `R → R`

`Quadrado``(X) = X * X`

Na primeira linha desta definição, observa-se a especificação do nome, do domínio e do contradomínio da função; e na segunda linha, a especificação da associação realizada por ela.

A aplicação de uma função é especificada pela justaposição de seu nome e do elemento do domínio ao qual se quer aplicá-la.

O correspondente elemento do contradomínio é obtido pela avaliação da expressão de associação, durante a qual, instanciar-se-á todas as ocorrências de variáveis parametrizantes da expressão, de acordo com o elemento do domínio ao qual se está aplicando a função.

A título de ilustração, é mostrada abaixo a aplicação da função **`Quadrado`** ao número real `13`. O resultado será `169`.

`Quadrado` `(13)`

Não é necessário que as funções tenham nome. A notação lambda, conforme idealizada por Church em 1941 [Dershem/Jipping90], provê meios para a definição de funções sem nome. Uma expressão lambda especifica tanto os argumentos, como a expressão de associação entre

elementos do domínio a elementos do contradomínio. Veja abaixo, a forma geral da definição de uma expressão lambda .

```
λ<argumentos>.<associação>
```

Os argumentos de uma uma expressão lambda são, muitas vezes, chamados de variáveis livres. Antes da avaliação da expressão lambda, eles representam qualquer elemento do domínio, mas durante sua avaliação, ele é instanciado como um membro determinado.

A título de ilustração, encontra-se definida abaixo uma expressão lambda que toma um número real e o associa a seu quadrado. A notação utilizada é a notação matemática.

```
λX.X*X
```

Quando uma expressão lambda é avaliada para dados argumentos, diz-se que a expressão foi aplicada àqueles argumentos.

Os mecanismos de aplicação são os mesmos para qualquer avaliação de função. Expressões lambda, como qualquer outra definição de função, podem ter mais do que um parâmetro. Veja abaixo a forma geral da aplicação de uma expressão lambda.

```
λ<argumentos>.<associação>:<aplicação>
```

A expressão abaixo ilustra a aplicação da expressão lambda definida anteriormente que associa um número real a seu quadrado. A aplicação resultará no valor `169`.

```
λX.X*X:13
```

Uma característica fundamental das funções matemáticas é que a ordem de avaliação de suas expressões de associação é controlada por expressões condicionais e por recursão, em vez de ser controlada por seqüenciação e iteração como nas linguagens de programação imperativas.

Funções matemáticas complexas são definidas em termos de outras funções. Uma função de alta ordem, ou forma funcional, é uma função que recebe funções como parâmetros e/ou retorna funções como resultado.

Uma função de alta ordem muito conhecida, é a composição de funções. Esta forma funcional toma duas funções como parâmetro e retorna uma função cujo valor é a aplicação de seu

primeiro parâmetro ao resultado de seu segundo parâmetro. Veja abaixo a definição da função H, que é a composição das funções F e G.

$$H \equiv F \circ G$$

Conclusão

É natural comparar os paradigmas imperativo e funcional. As linguagens que seguem o paradigma imperativo se baseiam diretamente no modelo de computação de von Neumann, o que torna a programação excessivamente trabalhosa, fazendo com que os programadores tenham que se preocupar com detalhes da arquitetura da máquina que são irrelevantes no contexto do problema que pretendem resolver computacionalmente. Por outro lado, consegue-se programas que executam muito eficientemente.

Já nas linguagens que seguem o paradigma funcional, existe o que é chamado de transparência referencial de variáveis, i.e., todas as inevitáveis referências a variáveis são controladas pela linguagem de modo transparente ao programador, que pode se concentrar no problema que tem para resolver, em vez de ter que se preocupar com detalhes do modelo von Neumann de computação.

Resulta disso muito menos esforço dispendido em programação, mas também programas muito menos eficientes.

As linguagens funcionais têm uma estrutura sintática muito simples, contrastando com a estrutura sintática complexa que normalmente são encontradas nas linguagens que seguem o paradigma imperativo de programação.

É muito trabalhoso escrever programas concorrentes adotando-se uma abordagem imperativa. O programador precisa dividir estaticamente o programa em partes concorrentes, e então implementá-las como tarefas.

Conceitos complexos estão envolvidos neste trabalho, tais como, cooperação entre tarefas, competição pelo processador, acesso exclusivo a estruturas de dados compartilhadas, etc. O processo como um todo é extremamente complicado e difícil de projetar, implementar, e, principalmente, testar [Sebesta89].

Numa abordagem funcional, os programas podem ser reduzidos a grafos, e então ser executados pelo processo de redução de grafos, o que pode ser feito com uma boa dose de concorrência.

Isto pode ser feito dinamicamente pelo sistema de execução, o que resulta em programas altamente manuteníveis e adaptáveis à máquina em que está rodando. Isto simplifica sobremaneira a programação, torna os programas mais legíveis, e alivia o programador de uma sobrecarga desnecessária.

Além disto, embora menos eficientes do que os programas imperativos em máquina convencionais, a programação funcional pode ser muito mais eficiente em máquinas paralelas, cada vez mais comuns em nossos dias.

Anexo I: Referências

[Sebesta89] Sebesta, R.W., "Concepts of Programming Languages", The Benjamin/Cummings Publishing Company, Inc., 1989.

[Ghezzi/Jazayeri82] Ghezzi, C., Jazayeri, M., "Programming Languages Concepts", John Wiley & Sons, Inc., 1982.

[Dershem/Jipping90] Dershem, H.L.; and Jipping, M.J., "Programming Languages: Structures and Models", Wadsworth, Inc., 1990.

Capítulo V:
A Linguagem de Programação LISP

Introdução

O paradigma funcional, como a maioria dos paradigmas de programação, desenvolveu-se em paralelo ao desenvolvimento de uma linguagem de programação. No seu caso, a linguagem foi LISP, uma abreviação de *List Processing*, desenvolvida em 1958, no MIT AI Project, por John McCarthy [Baranauskas93].

Desde então, muitas outras linguagens funcionais foram desenvolvidas, mas LISP é a mais antiga, e, ainda hoje, a mais usada delas.

Apesar de ter evoluído muito nestes últimos anos, muitos acham que ela não mais representa os conceitos atuais de programação funcional. Inclusive porque, excetuando-se sua primeira versão, todas as versões de LISP incluem alguma característica do paradigma imperativo, e.g., variáveis, atribuição e iteração.

Mas a despeito disto tudo isto, e de sua sintaxe estranha, LISP ainda representa bem os conceitos fundamentais da programação funcional [Sebesta89].

LISP foi originalmente projetada para atender às necessidades das primeiras aplicações em inteligência artificial, e por esta razão, guarda, ainda hoje, influências fortes do interesse dos grupos de pesquisa de então. Foi criada em uma época em que todo processamento era numérico, para atender uma enorme demanda por processamento simbólico e de listas.

Das pesquisas de McCarthy sobre computação simbólica, originou-se uma lista das características que uma linguagem para atender àqueles interesses deveria ter. Dentre elas estavam: método de controle de fluxo das funções matemáticas, i.e., recursão e expressões condicionais; e reaproveitamento automático de posições de memória não mais utilizadas.

Em LISP puro só existem dois tipos de dados: átomos e listas. Átomos são os símbolos de LISP. Constantes numéricas também são considerados átomos. Listas são coleções de elementos delimitados por parênteses. Os elementos de uma lista podem ser átomos ou então outras listas.

Originalmente foi definida uma notação para programas Lisp chamada Meta Expressões, ou M-expressões. Tencionava-se ter um compilador que traduzisse desta notação para código de máquina do IBM 704.

Depois, inspirado nas máquinas de Turing e motivado por certas idéias futuristas que tinha sobre processamento de listas, McCarthy teve a idéia de se ter em LISP uma função universal, que fosse capaz de avaliar qualquer outra função LISP.

O primeiro requisito para a construção desta função universal era ter uma notação única para representar dados e funções.

A notação parentizada de listas já tinha sido adotada em LISP para a representação dos dados, assim, decidiu-se criar convenções que permitissem a expressão de definições de função e de chamadas de função também sob a forma de lista.

Decidiu-se representar chamadas de função sob a forma de lista prefixa ou notação polonesa. Assim, as chamadas de função foram notadas como segue.

$$(\texttt{NomeDaFuncao } Arg_1\ Arg_2\ \ldots\ Arg_n)$$

Optou-se pelo uso de expressões lambda para a especificação da definição de funções. Foi feita uma adaptação da notação matemática destas expressões para sua representação em listas.

A fim de que as expressões lambda pudessem fazer referência umas às outras, resolveu-se associar a elas um nome. Veja abaixo como são representadas a expressões lambda.

$$(\texttt{LAMBDA } (Arg_1\ Arg_2\ \ldots\ Arg_n)\ \texttt{Associação})$$

McCarthy conseguiu desenvolver sua função universal para avaliar qualquer estrutura de dados representando código LISP. A função foi batizada de **`Eval`**. Foi percebido então, que uma implementação de **`Eval`** serviria como um interpretador da linguagem, surgindo assim o primeiro interpretador LISP.

O fato dos programas terem a mesma representação que os dados pode ser explorado de formas muito interessantes. Isto, combinado ao fato da maioria das implementações de LISP tomarem a forma de interpretadores, o que possibilita o acesso à função **`Eval`** em tempo de execução dos programas, abre caminho para que o programador escreva funções LISP que escrevem e executam código LISP.

Em concordância com as idéias originais de McCarthy sobre características de uma linguagem para processamento simbólico, LISP tem mecanismos automáticos para reaproveitamento de

memória não mais usada, e tem controle do fluxo de execução que segue o modelo das funções matemáticas, i.e., baseado em recursão e expressões condicionais.

Algumas características deste primeiro interpretador LISP perpetuaram-se em quase todas as implementações posteriores da linguagem. Uma delas é o uso de controle dinâmico de escopo, que, como é sabido, tem uma série de desvantagens [Sebesta89].

Algumas implementações mais recentes da linguagem abandonam esta característica histórica e implementam controle estático de escopo, não apenas resolvendo os problemas inerentes ao controle dinâmico, mas ainda tornando os programas mais legíveis, e simplificando ligeiramente a compilação.

Algumas Definições

Átomos

Entende-se que átomos são símbolos indivisíveis, ou seja, não se trata de uma cadeia de símbolos individualizáveis, trata-se de algo sem partes e só pode ser manipulado como um todo.

Átomos podem ser (1) inteiros, e.g., -7, 0 e 5; (2) reais, e.g., -9.7, 0.0 e 3.14; ou (3) simbólicos, e.g., pera, azul e urso.

Fica assim definido A, o conjunto dos átomos.

S-Expressões

Seja A o conjunto dos átomos. Definimos S, o conjunto das S-expressões, da seguinte maneira:

- $A \supset S$;
- Se α e $\beta \in S$, então o par $(\alpha . \beta) \in S$.

Assim, temos que –3, 5.9, amarelo, (1 . 2), (uva . 18.00), ((1 . 2) . (3 . 4)) e (1 . (2 . (3 . nil))) são S-expressões.

Listas

Seja S o conjunto das S-expressões. Definimos L, o conjunto das listas, da seguinte maneira:

- nil $\in$ L;
- Se $\alpha \in$ S e $\beta \in$ L, então o par $(\alpha . \beta) \in$ L.

Assim, temos que nil, (1 . nil), (1 . (2 . nil)), (1 . (2 . (3 . nil))), (1 . (2 . (3 . (4 . nil)))) e (7 . (3.14 . (banana . (((1 . 2) . (3 . 4)) . ((a . (b . (c . (d . nil)))) . nil))))) são listas.

É fácil perceber que essa notação é pouco confortável, além de tornar pouco legíveis as listas escritas na mesma.

Felizmente existe uma notação alternativa para tanto. Vemos logo em seguida as mesmas listas que observamos acima, só que agora escritas nessa notação simplificada: (), (1), (1 2), (1 2 3), (1 2 3 4) e (7 3.14 banana ((1 . 2) . (3 . 4)) (a b c d)).

Deve-se ressaltar que as listas denotadas segundo essa notação simplificada são efetivamente conforme expressas na primeira notação, e, portanto quando formos manipula-las, não devemos perder esse fato de vista.

LISP Puro

LISP puro tem 5 funções primitivas (car, cdr, cons, 1+ e 1-), 2 predicados (eq e atom) e 2 formas (cond e lambda).

- Car: S-A $\rightarrow$ S
  Car [$(\alpha . \beta)$] = α
- Cdr: S-A $\rightarrow$ S
  Cdr [$(\alpha . \beta)$] = β
- Cons: SxS $\rightarrow$ S-A
  Cons [$\alpha;\beta$] = $(\alpha . \beta)$

- Eq: AxA $\rightarrow$ {t, nil}

$$Eq\ [\alpha;\beta] = \begin{cases} t, \text{ se } \alpha = \beta; \text{ e} \\ nil, \text{ caso contrário.} \end{cases}$$

- Atom: S $\rightarrow$ {t, nil}

$$Atom\ [\alpha] = \begin{cases} t, \text{ se } \alpha \in A; \text{ e} \\ nil, \text{ caso contrário.} \end{cases}$$

- 1+: Z $\rightarrow$ Z

  $1+\ [\alpha] = \alpha + 1$

- 1-: Z $\rightarrow$ Z

  $1-\ [\alpha] = \alpha - 1$

M-Expressões

Como sabemos, a sintaxe que tem a linguagem LISP não é aquela que originalmente foi concebida para ela. LISP tem uma sintaxe pouco confortável, com muitos parênteses, muito diferente daquela que a princípio se concebia para a linguagem.

M-expressões representam uma sintaxe bem mais simples e muito mais fácil de ser empregada. Por esta razão, M-expressões são freqüentemente usadas como uma linguagem algorítmica para a escrita de algoritmos funcionais.

Tais algoritmos, posteriormente, poderão ser traduzidos para uma linguagem funcional real, LISP, por exemplo.

Chamada de Função

A forma geral de uma chamada de função é a seguinte:

$$F\ [PR_1; PR_2; \ldots ; PR_n]$$

onde (1) F é o nome da função; e (2) os PR_i's representam os parâmetros reais da função.

Expressões Condicionais

Expressões condicionais resultam em S-expressões. Sua forma geral é a seguinte:

$$[EL_1 \rightarrow R_1; EL_2 \rightarrow R_2; \ldots ; EL_n \rightarrow R_n]$$

onde (1) os EL_i representam expressões que resultam S-expressão que $\in$ {t, nil}; e (2) os R_i representam expressões que resultam S-expressões quaisquer.

Expressões condicionais são avaliadas da seguinte forma: (1) avalia-se cada uma das EL_i da esquerda para a direita; (2) ao se encontrar uma cujo valor seja t fica determinado que o valor da expressão condicional é R_i; (3) se todas as EL_i resultarem nil, o valor da expressão condicional é indefinido.

Definição de Funções

A forma geral de uma definição de função é a seguinte:

$$F: D \rightarrow CD$$
$$F\ [PF_1; PF_2; \ldots ; PF_n] = E$$

onde (1) F é o nome da função; (2) D representa o domínio da função F; (3) CD representa o contradomínio da função F; (4) os PF_i's representam os parâmetros formais da função F; e (5) E é a expressão de associação entre elementos de D e elementos de CD e representa uma expressão que resulta uma S-expressão.

Vale ressaltar que E pode ser uma constante, uma chamada de função ou uma expressão condicional.

Observação Final

Por convenção, empregaremos somente letras minúsculas para escrever átomos simbólicos, ao passo que as palavras que compõem os nomes de identificadores iniciar-se-ão sempre com uma letra maiúscula.

Exemplo

Abaixo encontraremos a definição de um predicado que verifica se um dado átomo ocorre em uma lista de átomos.

Ocorre: AxL → {t, nil}

Ocorre [A;L] = [Atom [L] → nil; Eq [A; Car [L]] → t; t → Ocorre [A; Cdr [L]]]

Tradução de M-Expressões para LISP

A tradução de um programa escrito sob a forma de M-expressões para LISP ocorre de forma simples e mecânica. Abaixo encontraremos as regras para realizar a referida tradução:

Tradução de Constantes

Constantes são traduzidas de forma muito simples. Átomos numéricos não sofrem nenhuma transformação no processo de tradução.

As demais constantes podem ser traduzidas de duas formas: (1) simplesmente acrescentando uma apóstrofe (') no início da constante; ou (2) envolvendo a constante entre parênteses e a precedendo da palavra reservada quote.

Veja os exemplos abaixo:

M-Expressão	Lisp
1	1
3.14	3.14
abacaxi	'abacaxi ou (quote abacaxi)
(abacate . 5.00)	'(abacate . 5.00) ou (quote (abacate . 5.00))
(1 2 3 4 5 6 7)	'(1 2 3 4 5 6 7) ou (quote (1 2 3 4 5 6 7))

Tradução de Identificadores de Parâmetros Formais

Identificadores de parâmetros formais são traduzidas de forma muito simples; não sofrem nenhuma transformação no processo de tradução.

Veja os exemplos abaixo:

M-Expressão	Lisp
A	A
X	X

Tradução de Chamada de Função

A tradução de Expressões Condicionais também não se acontece de forma complicada. Basta seguir o que indica a forma geral de tradução indicada abaixo:

M-Expressão	Lisp
F [PR_1; PR_2; ... ; PR_n]	(F TPR_1 TPR_2 ... TPR_n)

onde (1) F é o nome da função; (2) os PR_i's representam os parâmetros reais da função; e (3) os TPR_i's representam a tradução dos PR_i's.

Veja os exemplos abaixo:

M-Expressão	Lisp
Eq [A; nil]	(Eq A 'nil)
Eq [Cdr [A]; nil]	(Eq (Cdr A) 'nil)

Tradução de Expressões Condicionais

A tradução de Expressões Condicionais também não se dá de forma complicada. Basta seguir o que indica a forma geral de tradução abaixo:

M-Expressão	Lisp
[$EL_1 \rightarrow R_1$; $EL_2 \rightarrow R_2$; ... ; $EL_n \rightarrow R_n$]	(Cond (TEL_1 TR_1) (TEL_2 TR_2) ... (TEC_n TR_n))

onde (1) os EL_i representam expressões que resultam S-expressão que $\in$ {t, nil}; (2) os R_i representam expressões que resultam S-expressões quaisquer; (3) os TEL_i representam a tradução dos EL_i; e (4) os TR_i representam a tradução dos R_i.

Veja o exemplo abaixo:

M-Expressão	Lisp
[Atom [L] $\rightarrow$ nil; Eq [A; Car [L]] $\rightarrow$ t; t $\rightarrow$ Ocorre [A; Cdr [L]]]	(Cond ((Atom L) 'nil) ((Eq A (Car L)) 't) ('t (Ocorre A (Cdr L))))

Tradução de Definições de Função

A tradução de Definições de Função também é simples. Basta seguir o que indica a forma geral de tradução abaixo:

M-Expressão	Lisp
F: D $\rightarrow$ CD F [PF_1; PF_2; ... ; PF_n] = E	; F: D $\rightarrow$ CD (Defun F (TPF_1 TPF_2 ... TPF_n) TE)

onde (1) F é o nome da função; (2) D representa o domínio da função F; (3) CD representa o contradomínio da função F; (4) os PF_i's representam os parâmetros formais da função F; (5) E é a expressão de associação entre elementos de D e elementos de CD e representa uma

expressão que resulta uma S-expressão; (6) os TPF_i's representam a tradução dos PF_i's; e (7) TE representa a tradução de E.

Veja o exemplo abaixo:

M-Expressão	Lisp
Ocorre: AxL → {t, nil} Ocorre [A;L] = [Atom [L] → nil; Eq [A; Car [L]] → t; t → Ocorre [A; Cdr [L]]]	; Ocorre: AxL → {t, nil} (Defun Ocorre (A L) (Cond ((Atom L) 'nil) ((Eq A (Car L)) 't) ('t (Ocorre A (Cdr L)))))

XLISP 3.04A

λ-Expressões

Como já discutimos anteriormente, em LISP, programa e dados têm a mesma forma, podendo (1) o programa ser visto e manipulado como uma estrutura de dados; e (2) as estruturas de dados serem vistas e executadas como um trecho de programa.

λ-expressões são o que empregamos para representarmos funções como estruturas de dados manipuláveis.

Vimos na primeira parte desta apostila o que significam conceitualmente as λ-expressões, veremos a seguir de que forma uma função é representada como uma λ-expressão.

Consideremos a função Ocorre conforme definida abaixo:

```
(Defun Ocorre (A L)
       (Cond ((Atom L) 'nil)
             ((Eq A (Car L)) 't)
             ('t (Ocorre A (Cdr L)))))
```

Quando o interpretador LISP tratar essa definição, este promoverá a definição de um símbolo de nome Ocorre que terá como valor a seguinte lista:

```
'(Lambda (A L)
       (Cond ((Atom L) nil)
             ((Eq A (Car L)) t)
             (t (Ocorre A (Cdr L)))
```

Podemos entender o símbolo Ocorre como se fosse uma variável que contém a lista acima. Para, a partir dele, obter a lista que ele representa, basta aplicar as seguintes funções como segue:

```
(get-lambda-expression (function Ocorre))
```

Desta forma, poderemos manipular a referida lista, por exemplo, tomando dela uma parte. Assim, se calcularmos,

```
(Car (Cdr (Cdr (Get-Lambda-Expression (Function Ocorre)))))
```

obteremos o valor

```
'Cond
```

E a criação de símbolos não está restrita ao interpretador LISP. O programador também pode definir símbolos. Isto pode ser feito através da função SetFun. Veja abaixo outra alternativa para a definição da função Ocorre, além, naturalmente, da forma já estudada (Defun).

```
(SetFun 'Ocorre '(Lambda (A L)
                    (Cond ((Atom L) nil)
                          ((Eq A (Car L)) t)
                          (t (Ocorre A (Cdr L)))
```

Note que a mesma lista poderia ser obtida não pela escrita explícita desta constante, mas pela execução de funções que a produzissem.

Principais Funções de Biblioteca

Parte I: Funções Aritméticas

1. (truncate FLT): produz a parte inteira do real FLT;
2. (round FLT): produz o arredondamento do real FLT para o inteiro mais próximo;
3. (floor FLT): produz o arredondamento para baixo do real FLT;
4. (ceiling FLT): produz o arredondamento para cima do real FLT;
5. (float INT): produz o equivalente real do número inteiro INT;
6. (rational FLT): produz o equivalente racional do número real FLT;

7. (+ NUM_1 NUM_2 ... NUM_n): produz a soma de todos os números que recebe como argumento;
8. (- NUM_1 NUM_2 ... NUM_n): recebe uma série de números como argumento; produz o primeiro deles com sinal invertido (no caso dele ser seu único argumento) ou produz a subtração de todos os demais do primeiro (no caso de receber mais de um);
9. (* NUM_1 NUM_2 ... NUM_n): produz a multiplicação de todos os números que recebe como argumento;
10. (/ NUM_1 NUM_2 ... NUM_n): recebe uma série de números como argumento; produz o inverso de seu primeiro deles (no caso dele ser o único) ou produz a divisão do primeiro deles por todos os demais (no caso de receber mais de um);
11. (1+ NUM): produz o incremento de NUM;
12. (1- NUM): produz o decremento de NUM;
13. (rem NUM_1 NUM_2) ou (mod NUM_1 NUM_2): produz o resto da divisão inteira de NUM_1 por NUM_2;
14. (min NUM_1 NUM_2 ... NUM_n): produz o menor de todos os números que recebe como argumento;
15. (max NUM_1 NUM_2 ... NUM_n): produz o maior de todos os números que recebe como argumento;
16. (abs NUM): produz o valor absoluto de NUM;
17. (gcd INT_1 INT_2 ... INT_n): produz o máximo divisor comum dos números inteiros que recebe como argumento;
18. (lcm INT_1 INT_2 ... INT_n): produz o mínimo múltiplo comum dos números inteiros que recebe como argumento;
19. (random FLT): produz um número aleatório entre 0 e o número real FLT;
20. (setq *random-state* (make-a-random-state t)): ajusta uma nova semente geradora de números aleatórios;

(make-a-random-state): produz a atual semente geradora de números aleatórios;

21. (sin NUM): produz o seno do valor angular NUM expresso em radianos;
22. (cos NUM): produz o coseno do valor angular NUM expresso em radianos;
23. (tan NUM): produz a tangente do valor angular NUM expresso em radianos;
24. (asin NUM): produz um valor angular em radianos cujo seno vale NUM;
25. (acos NUM): produz um valor angular em radianos cujo coseno vale NUM;
26. (atan NUM): produz um valor angular em radianos cuja tangente vale NUM;
27. (expt NUM_1 NUM_2): produz NUM_1 elevado a NUM_2;
28. (exp NUM): produz e elevado NUM (onde e é o número de Neper);
29. log NUM_1 NUM_2): produz o logaritmo de NUM_1 na base NUM_2 (se NUM_2 for omitido, assume-se logaritmo natural);
30. (sqrt NUM): produz a raiz quadrada de NUM;
31. (numerator RAC): produz o numerador do número racional RAC;
32. (denominator RAC): produz o denominador do número racional RAC;
33. (complex FLT_1 FLT_2): produz um número complexo com o coeficiente real FLT_1 e o coeficiente imaginário FLT_2;
34. (realpart CPX): produz o coeficiente real do número complexo CPX;
35. (imagpart CPX): produz o coefiente imaginário do número complexo CPX;
36. (conjugate CPX): produz o conjugado do número complexo CPX;
37. (phase CPX): produz a fase do número conjugado CPX (equivale a ***(atan (/ (imagpart CPX) (realpart CPX)))***);
38. (= NUM_1 NUM_2 ... NUM_n): produz t, caso todos os números que recebe como argumento sejam iguais; produz nil, caso contrário;

39. (/= NUM_1 NUM_2 ... NUM_n): produz t, caso seu primeiro argumento seja diferente de todos os demais; produz nil, caso contrário;

40. (< NUM_1 NUM_2 ... NUM_n): produz t, caso seu primeiro argumento seja menor que todos os demais; produz nil, caso contrário;

41. (<= NUM_1 NUM_2 ... NUM_n): produz t, caso seu primeiro argumento seja menor ou igual a todos os demais; produz nil, caso contrário;

42. (> NUM_1 NUM_2 ... NUM_n): produz t, caso seu primeiro argumento seja maior que todos os demais; produz nil, caso contrário;

43. (>= NUM_1 NUM_2 ... NUM_n): produz t, caso seu primeiro argumento seja maior ou igual a todos os demais; produz nil, caso contrário;

Parte II: Funções de Sistema

1. (setfun SMB LBD): define a função cujo nome é dado pelo símbolo SMB como sendo expressa pela λ-expressão LBD;

2. (load STR): carrega o arquivo de código fonte cujo nome é expresso pelo string STR (diretórios separados por \\ e não por \; assume extensão .LSP);

3. (restore STR): carrega a área de trabalho anteriormente salva no arquivo cujo nome é expresso pelo string STR (diretórios separados por \\ e não por \; assume extensão .WKS);

4. (save STR): salva a área de trabalho no arquivo cujo nome é expresso pelo string STR (diretórios separados por \\ e não por \; assume extensão .WKS);

5. (dribble STR): salva no arquivo cujo nome é expresso pelo string STR a transcrição de uma sessão XLisp;

6. (gc): força a coleta de lixo (que normalmente acontece de forma automática sob o controle do ambiente XLisp);

7. (expand INT): expande a memória pelo acréscimo de INT novos segmentos de memória de tamanho fixo; retorna a quantidade de segmentos acrescentada;

8. (room): exibe estatísticas de alocação de memória;

9. (time FCN): executa a função FCN e produz seu tempo de execução;
10. (/ (get-internal-real-time) internal-time-units-per-second): produz o tempo de relógio decorrido;
11. (/ (get-internal-run-time) internal-time-units-per-second): produz o tempo de execução decorrido;
12. (type-of EXP): produz o tipo da expressão EXP (LIST, se nil; CONS se lista ou par com ponto; SYMBOL, se símbolo; OBJECT, se objeto; SUBR, se função de biblioteca; FSUBR, se função de biblioteca escrita em C; CLOSURE, se função de usuário; STRING, se string; FIXNUM, se inteiro; BIGNUM, se inteiro longo; RATIO, se racional; FLONUM, se real; COMPLEX, se complexo; CHARACTER, se caractere; FILE-STREAM, se arquivo; UNNAMED-STREAM, se arquivo sem nome; ARRAY, se vetor; HASH-TABLE, se tabela de espalhamento; SYM, para estruturas "sym");
13. (coerce EXP TIP): produz a conversão da expressão EXP para o tipo TIP (TIP pode ser qualquer um dos tipos produzidos pela função type-of);
14. (peek INT): produz o conteúto do endereço de memória expresso pelo número inteiro INT;
15. (poke INT_1 INT_2): coloca no endereço de memória expresso pelo número inteiro INT_1, o valor expresso pelo número inteiro INT_2;
16. (get-key): produz um inteiro que representa o código ASCII da tecla que for pressionada;
17. (exit): sai do XLisp;
18. (eval FCN): executa a função FCN;

Parte III: Funções de Seqüência (listas, vetores e strings)

1. (concatenate TIP EXP_1 EXP_2 ... EXP_n): produz um resultado do tipo TIP que equivale a concatenação das expressões EXP_i recebidas como argumento; TIP pode ser CONS, LIST, ARRAY, ou STRING;
2. (elt SEQ INT): produz o elemento de ordem INT da seqüência SEQ;
3. (length SEQ): produz o tamanho da seqüência SEQ;

4. (reverse SEQ): produz o inverso da seqüência SEQ;
5. (subseq SEQ NUM_1 NUM_2): produz a subseqüência da seqüência SEQ que se inicia em na posição expressa pelo inteiro NUM_1 e que termina na posição expressa pelo inteiro NUM_2;

Parte IV: Funções de Lista

1. (car LST): produz o primeiro elemento da lista LST;
2. (cdr LST): produz o que resta da lista LST, retirado seu primeiro elemento;
3. (cxxr LST): equivale a (cxr (cxr (LST)), onde 'x' é 'a' ou 'd';
4. (cxxxr LST): equivale a (cxr (cxr (cxr (LST))), onde 'x' é 'a' ou 'd';
5. (cxxxxr LST): equivale a (cxr (cxr (cxr (cxr (LST)))), onde 'x' é 'a' ou 'd';
6. (first LST): equivale a (car LST);
7. (second LST): equivale a (cadr LST);
8. (third LST): equivale a (caddr LST);
9. (fourth LST): equivale a (cadddr LST);
10. (rest LST): equivale a (cdr LST);
11. (last LST): produz o último elemento da lista LST;
12. (cons EXP_1 EXP_2): constroi o par com ponto (EXP_1 . EXP_2) a partir das expressões EXP_i dadas;
13. (list EXP_1 EXP_2 ... EXP_n): produz uma lista que tem por elementos as expressões EXP_i dadas;
14. (append LST_1 LST_2 ... LST_n): produz uma lista que expressa a concatenação das listas LST_i dadas;
15. (list-length LST): produz um inteiro que expressa o tamanho da lista LST;
16. (butlast LST INT): produz uma lista que tem todos os elementos presentes na lista LST exceto os INT últimos;

17. (nth INT LST): produz o elemento de ordem INT da lista LST;

18. (nthcdr INT LST): produz o que resta da lista LST, após tomado INT vezes o seu cdr;

Parte V: Funções de Bit

1. (logand INT_1 INT_2 ... INT_n): produz o **and** bit a bit dos inteiros INT_i recebidos como argumento;
2. (logior INT_1 INT_2 ... INT_n): produz o **or** bit a bit dos inteiros INT_i recebidos como argumento;
3. (logxor INT_1 INT_2 ... INT_n): produz o **xor** bit a bit dos inteiros INT_i recebidos como argumento;
4. (logeqv INT_1 INT_2 ... INT_n): produz a **equivalência** bit a bit dos inteiros INT_i recebidos como argumento;
5. (lognand INT_1 INT_2 ... INT_n): equivale a (lognot (logand INT_1 INT_2 ...INT_n));
6. (logandc1 INT_1 INT_2): equivale a (logand (lognot INT_1) INT_2);
7. (logandc2 INT_1 INT_2): equivale a (logand INT_1 (lognot INT_2));
8. (lognor INT_1 INT_2 ... INT_n): equivale a (lognot (logior INT_1 INT_2 ...INT_n));
9. (logorc1 INT_1 INT_2): equivale a (logior (lognot INT_1) INT_2);
10. (logorc2 INT_1 INT_2): equivale a (logior INT_1 (lognot INT_2));
11. (lognot INT): produz o ***not*** bit a bit do inteiro INT recebido como argumento;
12. (logtest INT_1 INT_2): produz t, caso o ***and*** bit a bit dos inteiros INTi dados produzir um valor não nulo; produz nil, caso contrário;
13. (logbitp INT_1 INT_2): produz t, caso o bit na posição INT_1 do inteiro INT_2 valer 1; produz nil, caso contrário;
14. (logcount INT): se o número inteiro INT for positivo, produz a contagem quantos bits 1 o constituem; caso contrário, produz a contagem quantos zeros constituem seu correspondente positivo;

15. (integer-length INT): produz a quantidade de bits que são necessários para representar o número inteiro INT;

16. (ash INT_1 INT_2): desloca INT_2 vezes os bits do inteiro INT_1 (para a esquerda, se INT_2 for positivo; para a direita, caso contrário);

Parte V: Funções de String

1. (string INT): produz um string contendo tão somente o caractere cujo código ASCII é expresso pelo inteiro INT;
2. (string-trim STR_1 STR_2): produz um string em tudo idêntico a STR2, exceto pelo fato de que remove de seu início e de seu final os caracteres que constituem o string STR_1;
3. (string-left-trim STR_1 STR_2): produz um string em tudo idêntico a STR2, exceto pelo fato de que remove de seu início os caracteres que constituem o string STR_1;
4. (string-right-trim STR_1 STR_2): produz um string em tudo idêntico a STR2, exceto pelo fato de que remove de seu final os caracteres que constituem o string STR_1;
5. (strcat STR_1 STR_2 ... STR_n): produz a concatenação dos strings STRi fornecidos como argumento;

Parte VI: Predicados

1. (atom EXP): produz t, caso a expressão EXP seja um átomo; produz nil, caso contrário;
2. (simbolp EXP): produz t, caso a expressão EXP seja um símbolo; produz nil, caso contrário;
3. (numberp EXP): produz t, caso a expressão EXP seja um número; produz nil, caso contrário;
4. (null EXP): produz t, caso a expressão EXP seja nil; produz nil, caso contrário;
5. (not EXP): produz t, caso a expressão EXP seja nil; produz nil, caso contrário;
6. (listp EXP): produz t, caso a expressão EXP seja uma lista; produz nil, caso contrário;
7. (endp EXP): produz t, caso a expressão EXP seja nil; produz nil, caso contrário;

8. (consp EXP): produz t, caso a expressão EXP seja uma lista não vazia; produz nil, caso contrário;
9. (constantp EXP): produz t, caso a expressão EXP seja uma constante; produz nil, caso contrário;
10. (specialp EXP): produz t, caso a expressão EXP seja um símbolo especial; produz nil, caso contrário;
11. (integerp EXP): produz t, caso a expressão EXP seja um inteiro; produz nil, caso contrário;
12. (floatp EXP): produz t, caso a expressão EXP seja um float; produz nil, caso contrário;
13. (rationalp EXP): produz t, caso a expressão EXP seja um número racional; produz nil, caso contrário;
14. (complexp EXP): produz t, caso a expressão EXP seja um número complexo; produz nil, caso contrário;
15. (stringp EXP): produz t, caso a expressão EXP seja um número string; produz nil, caso contrário;
16. (characterp EXP): produz t, caso a expressão EXP seja um caractere; produz nil, caso contrário;
17. (boundp SMB): produz t, caso o símbolo SMB tenha associado a sim algum valor; produz nil, caso contrário;
18. (fboundp SMB): produz t, caso o símbolo SMB tenha associado a sim algum valor funcional; produz nil, caso contrário;
19. (minusp NUM): produz t, caso o número NUM seja negativo; produz nil, caso contrário;
20. (zerop NUM): produz t, caso o número NUM seja zero; produz nil, caso contrário;
21. (plusp NUM): produz t, caso o número NUM seja positivo; produz nil, caso contrário;
22. (evenp NUM): produz t, caso o número NUM seja par; produz nil, caso contrário;
23. (oddp NUM): produz t, caso o número NUM seja ímpar; produz nil, caso contrário;

24. (eq EXP_1 EXP_2): produz t, caso a expressão EXP_1 seja igual à expressão EXP_2 (estão na mesma posição de memória); produz nil, caso contrário; só funciona com caracteres, símbolos e inteiros curtos;

25. (eql EXP_1 EXP_2): produz t, caso a expressão EXP_1 seja igual à expressão EXP_2; produz nil, caso contrário; funciona com todo tipo de número, desde que tenham o mesmo tipo;

26. (equal EXP_1 EXP_2): produz t, caso a expressão EXP_1 seja igual à expressão EXP_2; produz nil, caso contrário; funciona com qualquer par de expressões;

27. (typep EXP TIP): produz t, caso a expressão EXP seja do tipo TIP; produz nil, caso contrário;

Anexo I:
Exercícios Propostos

1. Escreva um predicado que verifica se um número inteiro é negativo.
2. Escreva os predicados NE, GT, GE, LT, LE (respectivamente, diferente, maior, maior ou igual, menor, e menor ou igual). Considere que o domínio de todos os predicados é o conjunto dos números inteiros.
3. Escreva as funções soma, diferença, multiplicação, quociente e resto. Considere que o domínio de todas as funções é o conjunto dos pares ordenados de números inteiros, e que o contradomínio de todas elas é o conjunto dos números inteiros.
4. Escreva uma função que calcula e retorna o fatorial de um dado numero natural.
5. Escreva uma função que calcule o mdc de dois números inteiros dados.

   Saiba que $\text{mdc}(X,Y) = \begin{cases} Y, \text{ se resto } (X / Y) = 0; \\ \text{mdc } (Y, \text{ resto } (X / Y)), \text{ cc.} \end{cases}$
6. Escreva um predicado que verifica se um átomo ocorre em uma lista de átomos.
7. Escreva uma função para incluir ordenadamente um átomo inteiro em uma lista de inteiros.
8. Escreva uma função que elimina de uma lista de átomos todas as ocorrências de um dado átomo.
9. Escreva uma função para inverter uma lista de átomos.
10. Escreva uma função para inverter uma lista de átomos e todas suas sublistas.
11. Escreva uma função para concatenar duas listas.
12. Escreva uma função para ordenar uma lista de números inteiros.
13. Escreva uma função para fazer a intercalação de duas listas ordenadas de números inteiros.

14. Considerando uma lista de átomos como sendo um conjunto, escreva as funções de intercessão e diferença.

15. Listas de propriedades são listas que armazenam uma série propriedades ou pares ordenados da forma (Propriedade,Valor). Considerando que o par com ponto `(Propriedade . Valor)` representa uma propriedade, e usando o formalismo de M-Expressões, escreva uma função que acrescenta uma nova propriedade a uma lista de propriedades. Assuma que a lista esta em ordem ascendente de Propriedade, e que sua função deve mantê-la assim.

16. Escreva um predicado que verifica se uma S-Expressão é uma lista.

17. Escreva um predicado que verifica se um átomo ocorre em uma S-Expressão.

18. Escreva uma função para contar o número de parênteses esquerdos e direitos de uma S-Expressão.

19. Escreva uma função que retorna o primeiro elemento atômico de uma S-Expressão (sei primeiro elemento, se este for um átomo, ou o primeiro átomo de seu primeiro elemento, caso contrário).

20. Escreva um predicado que verifica se uma S-Expressão ocorre como subexpressão de outra S-Expressão.

Anexo II: Referências

[Sebesta89] Sebesta, R.W., "Concepts of Programming Languages", The Benjamin/Cummings Publishing Company, Inc., 1989.

[Baranauskas93] Baranauskas, M.C.C., "Procedimento, Função, Objeto ou Lógica? Linguagens de Programação vistas pelos seus Paradígmas" *in* Computadores e Conhecimento - Repensando a Educação, Valente, J.A. (Org.), NIED, UNICAMP, 1993.

Capítulo VI:
O Paradigma Lógico

Introdução

O paradigma lógico se baseia em lógica simbólica e usa inferência lógica para produzir resultados. Linguagens lógicas são profundamente diferentes das linguagens imperativas e das linguagens funcionais [Sebesta89].

A abordagem do paradigma lógico à resolução de problemas foi desenvolvida para prova automática de teoremas [Sebesta89]. Assim, escrever um programa lógico é essencialmente a mesma coisa que escrever um programa que prova um teorema [Dershem/Jipping90].

Para compreender melhor como funciona um programa baseado neste paradigma, examinemos agora como um teorema é provado.

Parte-se de uma série de fatos que são tidos como verdadeiros, ou axiomas em linguajar matemático. Manipula-se então os axiomas usando para tanto regras de inferência lógica até se chegar à proposição que se quer provar. A proposição a ser provada é considerada uma espécie de objetivo a ser alcançado.

Um programa escrito segundo o modelo de programação lógica envolve um banco de dados composto por uma série de fatos, e por uma coleção de regras de inferência que estabelecem relações entre fatos, tudo isto expresso em uma sintaxe própria à linguagem. Este banco de dados, aliado a um processo automático de inferência, é usado para verificar a validade de novas proposições.

Cálculo de Predicados

Uma proposição é uma afirmação que pode ou não ser verdadeira, e pode envolver diversos objetos e relações entre eles. A lógica formal foi desenvolvida para prover meios para descrever proposições formalmente, de modo a verificar sua veracidade.

Lógica simbólica pode ser usada para estas três finalidades básicas: expressar proposições, expressar relacionamentos entre proposições, e descrever como proposições podem ser inferidas a partir de proposições que se sabe serem verdadeiras.

Existe muita semelhança entre a lógica formal e a matemática. Na verdade muito da matemática pode ser pensado em termos da lógica formal. O cálculo de predicados, que é uma particular forma de lógica simbólica, tem sido muito usado em programação lógica.

Proposições

Proposições, no cálculo de predicados, são afirmações a cerca de objetos. Objetos são representados por termos simples que podem ser variáveis ou constantes. Constantes são símbolos que representam um determinado objeto. Variáveis são símbolos que representam objetos genéricos, podendo representar diferentes objetos em diferentes instantes. Variáveis somente podem ser introduzidos em uma proposição se introduzidos por um quantificador universal ($\forall$) ou existencial ($\exists$).

As proposições mais simples que existem são compostas por um único termo, e são chamadas de proposições atômicas. Termos são relações matemáticas escritas sob a forma de expressão funcional. Uma expressão funcional consiste de um functor, ou símbolo de função, seguido por uma lista ordenada de argumentos. Termos representam propriedades e relações entre objetos.

Proposições compostas podem ter duas ou mais proposições atômicas conectadas por uma negação ($\neg$), por uma conjunção ($\cap$), por uma disjunção ($\cup$) ou por uma implicação ($\supset$).

Veja abaixo um exemplo de duas proposições. A primeira é uma proposição atômica, e afirma que joão é humano. A segunda é uma proposição composta, e afirma que todo humano é mortal.

```
⊃ Humano (joao)
```
($\forall$X) humano (X) $\supset$ mortal (X)

Cláusulas

Como foi descrito até agora, o cálculo de predicados permite muitas descrições diferentes para uma mesma proposição.

Para simplificar e padronizar a forma das cláusulas lógicas, define-se a forma clausal. Sem perda de generalidade, qualquer proposição pode ser reduzida à forma clausal [Sebesta89]. Veja abaixo a forma geral de uma proposição na forma clausal.

$$C_1 \cup C_2 \cup \ldots \cup C_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$$

Nesta proposição na forma clausal, tanto os A quanto os C são termos. Ela significa que se todos os A_i forem verdadeiros, pelo menos um dentre os C_i também será verdadeiro. A parte direita de uma forma clausal é chamada antecedente, e a parte esquerda, conseqüente.

Na forma clausal não se emprega o quantificador universal (∀) e tampouco o quantificador existencial (∃). Por convenção, todas as variáveis presentes em uma cláusula devem ser encaradas como qualificadas implicitamente pelo quantificador universal (∀).

Resolução

Resolução é uma regra de inferência concebida para ser aplicada a proposições na forma clausal que tem um potencial muito grande de aplicação na prova automática de teoremas [Sebesta89]. O conceito de resolução é o seguinte: suponha que existam duas proposições da forma:

$$C_1 \subset A_1$$
$$C_2 \subset A_2$$

Suponha ainda que C_1 seja idêntica a A_2. Poderíamos então chamar C_1 e A_2 de P. Assim, rescrever as duas proposições como segue:

$$P \subset A_1$$
$$C_2 \subset P$$

Agora, como A_1 implica P e P implica seja idêntica a C_2, é obvio que A_1 implica C_2. Assim, poder-se-ia escrever

$$C_2 \subset A_1$$

Este processo, através do qual pode-se obter esta proposição a partir das duas anteriores, é chamado de resolução, e constitui a base do processo de inferência automática das linguagens de programação lógicas.

Se tanto a parte antecedente quando a conseqüente das expressões possuírem diversos termos, a nova proposição inferida conterá todos os termos das duas proposições, exceto aquele comum às ambas que poderá ser cancelado.

Na verdade, o processo de inferência é bem mais complicado do que foi mostrado aqui. Por exemplo, se existirem variáveis nas proposições, será necessário encontrar valores para estas variáveis que permitam o sucesso do processo de emparelhamento.

O processo de determinação de valores para as variáveis é chamado de unificação, e a associação temporária de valores às variáveis é chamada instanciação.

É comum o processo de resolução produzir instanciações que se mostram incapazes de comprovar a veracidade da proposição objetivo, sendo forçado a retornar a objetivos anteriores e alterar instanciações que tenham sido feitas.

A propriedade crucial do mecanismo de resolução é sua habilidade de detectar inconsistências em um dado conjunto de proposições. Esta propriedade permite que se faça uso deste mecanismo para provar teoremas, o que é feito como explicado abaixo.

Imagine a prova de um teorema expresso em termos de cálculo de predicados como um conjunto pertinente de proposições ao qual se acrescenta o próprio teorema que se deseja provar negado.

O teorema negado é introduzido no conjunto de proposições para que o mecanismo de resolução acuse uma inconsistência, e o teorema seja provado por contradição. Tipicamente, o conjunto original de proposições é chamado de conjunto das hipóteses, e o teorema negado é chamado de objetivo.

Prova de teoremas é a base da programação lógica. Teoricamente, todo o mecanismo que foi visto até agora funciona muito bem. No entanto, na prática, programas lógicos podem ter um tempo de execução enorme se o banco de dados com as proposições for muito grande.

Normalmente, usa-se apenas um tipo restrito de forma clausal para representar proposições que se pretende trabalhar com o mecanismo de resolução. Este tipo especial de proposição é chamado de Cláusula de Horn.

A restrição que se impõe a estas cláusulas é que tenha a parte conseqüente vazia, ou constituída somente por uma proposição atômica.

Exemplo

Considere o seguinte banco de conhecimentos:

nasceu(jose,m,eleazar,ana)⊂
nasceu(sebastiao,m,eleazar,ana)⊂
nasceu(neusa,f,zair,maria)⊂
nasceu(paulo,m,zair,maria)⊂
nasceu(andre_luis,m,jose,neusa)⊂
nasceu(jose_luis,m,jose,neusa)⊂
nasceu(marco_antonio,m,sebastiao,zulma)⊂
nasceu(marcio_augusto,m,sebastiao,zulma)⊂
nasceu(tieni,f,sebastiao,nilce)⊂
nasceu(marcel,m,paulo,maria_luiza)⊂
nasceu(marcelo,m,paulo,maria_luiza)⊂
nasceu(marcio_luis,m,paulo,maria_luiza)⊂
…
morreu(zulma)⊂
…
casou(ana,eleazar)⊂
casou(maria,zair)⊂
casou(jose,neusa)⊂
casou(sebastiao,zulma)⊂
casou(sebastiao,nilce)⊂
…
pai(P,F)⊂nasceu(F,_,P,_)
mae(M,F)⊂nasceu(F,_,_,M)
genitor(G,F)⊂mae(G,F)
genitor(G,F)⊂pai(G,F)
avo(A,N)⊂pai(A,G)∩genitor(G,N)
…

Considere que seja feita a seguinte pergunta:

⊂avo(zair,andre)

O Algoritmo da Resolução faria as seguintes manipulações lógicas envolvendo a cláusula que exprime a pergunta e aquelas que constituem o banco de conhecimento:

Manipulação 1:
⊂avo(zair,andre)
avo(A,N)⊂pai(A,G)∩genitor(G,N)
~~avo(A,N)⊂avo(zair,andre)~~∩pai(A,G)∩genitor(G,N)
Fazendo as instanciações A\zair e N\andre, podemos cancelar os termos riscados por te-los tornado equivalentes, restando verificar ⊂pai(zair,G)∩genitor(G,andre)

Manipulação 2:
⊂pai(zair,G)∩genitor(G,andre)
pai(P,F)⊂nasceu(F,_,P,_)
~~pai(P,F)⊂pai(zair,G)~~∩nasceu(F,_,P,_)∩genitor(G,andre)
Fazendo as instanciações P\zair e F\G, podemos cancelar os termos riscados por te-los tornado equivalentes, restando verificar ⊂nasceu(F,_,zair,_)∩genitor(F,andre)

Manipulação 3:
⊂nasceu(F,_,zair,_)∩genitor(F,andre)
nasceu(jose,m,eleazar,ana)⊂
~~nasceu(jose,m,eleazar,ana)⊂nasceu(F,_,zair,_)~~∩genitor(F,andre)
Como não existem instanciações capazes de possibilitar o cancelamento os termos riscados, já que os mesmo jamais poderão se tornar equivalentes, desfaremos a Manipulação 3, voltanto a ter que verificar ⊂nasceu(F,_,zair,_)∩genitor(F,andre)

Manipulação 4:
⊂nasceu(F,_,zair,_)∩genitor(F,andre)
nasceu(sebastiao,m,eleazar,ana)⊂
~~nasceu(sebastiao,m,eleazar,ana)⊂nasceu(F,_,zair,_)~~∩genitor(F,andre)
Como não existem instanciações capazes de possibilitar o cancelamento os termos riscados, já que os mesmo jamais poderão se tornar equivalentes, desfaremos a Manipulação 4, voltanto a ter que verificar ⊂nasceu(F,_,zair,_)∩genitor(F,andre)

Manipulação 5:
⊂nasceu(F,_,zair,_)∩genitor(F,andre)
nasceu(neusa,f,zair,maria)⊂
~~nasceu(neusa,f,zair,maria)⊂nasceu(F,_,zair,_)~~∩genitor(F,andre)
Fazendo as instanciações F\neusa, podemos cancelar os termos riscados por te-los tornado equivalentes, restando verificar ⊂genitor(neusa,andre)

Manipulação 6:
⊂genitor(neusa,andre)
genitor(G,F)⊂mae(G,F)
~~genitor(G,F)⊂genitor(neusa,andre)~~∩mae(G,F)
Fazendo as instanciações G\neusa e F\andre, podemos cancelar os termos riscados por te-los tornado equivalentes, restando verificar ⊂mae(neusa,andre)

Manipulação 7:
⊂mae(neusa,andre)
mae(M,F)⊂nasceu(F,_,_,M)
~~mae(M,F)⊂mae(neusa,andre)~~∩nasceu(F,_,_,M)
Fazendo as instanciações M\neusa e F\andre, podemos cancelar os termos riscados por te-los tornado equivalentes, restando verificar ⊂nasceu(andre,_,_,neusa)

Manipulação 8:
⊂nasceu(andre,_,_,neusa)
mae(M,F)⊂nasceu(F,_,_,M)
~~mae(M,F)⊂mae(neusa,andre)~~∩nasceu(F,_,_,M)
Fazendo as instanciações M\neusa e F\andre, podemos cancelar os termos riscados por te-los tornado equivalentes, restando verificar ⊂nasceu(andre,_,_,neusa)

Manipulação 9:
⊂nasceu(andre,_,_,neusa)
nasceu(jose,m,eleazar,ana)⊂
~~nasceu(jose,m,eleazar,ana)⊂nasceu(andre,_,_,neusa)~~
Como não existem instanciações capazes de possibilitar o cancelamento os termos riscados, já que os mesmo jamais poderão se tornar equivalentes, desfaremos a Manipulação 3, voltanto a ter que verificar ⊂nasceu(andre,_,_,neusa)

Manipulação 10:
⊂nasceu(andre,_,_,neusa)
nasceu(sebastiao,m,eleazar,ana)⊂
~~nasceu(sebastiao,m,eleazar,ana)⊂ nasceu(andre,_,_,neusa)~~
Como não existem instanciações capazes de possibilitar o cancelamento os termos riscados, já que os mesmo jamais poderão se tornar equivalentes, desfaremos a Manipulação 4, voltanto a ter que verificar ⊂nasceu(andre,_,_,neusa)

Manipulação 11:
⊂nasceu(andre,_,_,neusa)
nasceu(neusa,f,zair,maria)⊂
~~nasceu(neusa,f,zair,maria)⊂ nasceu(andre,_,_,neusa)~~
Como não existem instanciações capazes de possibilitar o cancelamento os termos riscados, já que os mesmo jamais poderão se tornar equivalentes, desfaremos a Manipulação 4, voltanto a ter que verificar ⊂nasceu(andre,_,_,neusa)

Manipulação 12:
⊂nasceu(andre,_,_,neusa)
nasceu(paulo,f,zair,maria)⊂
~~nasceu(paulo,f,zair,maria)⊂ nasceu(andre,_,_,neusa)~~
Como não existem instanciações capazes de possibilitar o cancelamento os termos riscados, já que os mesmo jamais poderão se tornar equivalentes, desfaremos a Manipulação 4, voltanto a ter que verificar ⊂nasceu(andre,_,_,neusa)

Manipulação 13:
⊂nasceu(andre,_,_,neusa)
nasceu(andre,m,jose,neusa)⊂
~~nasceu(andre,m,jose,neusa)⊂nasceu(andre,_,_,neusa)~~
Podemos cancelar os termos riscados em razão dos mesmo serem equivalentes, restando verificar ⊂

A cláusula constituida apenas por um símbolo de implicação (⊂) representa sucesso, i.e., a pergunta original tem resposta positiva.

Conclusão

As linguagens lógicas são muitas vezes chamadas de declarativas porque os programas lógicos são constituídos por declarações, que chamamos de proposições, em vez de serem constituídos por atribuições e comandos para controle do fluxo de execução.

Dentre as áreas onde atual e potencialmente podem ser encontradas aplicações baseadas no paradigma funcional, podemos destacar: sistemas gerenciadores de bancos de dados relacionais, sistemas especialistas, processamento de linguagens naturais e educação.

Uma característica essencial das linguagens lógicas é que a semântica de cada um de seus elementos constituintes, as proposições, pode ser compreendida de maneira isolada, posto que o significado de uma proposição não interfere no significado de outra, o que não acontece nas linguagens que seguem o paradigma imperativo [Sebesta89].

Linguagens lógicas são não procedimentais, o que significa que, diferentemente do que acontece nas linguagens imperativas, nestas linguagens não se especifica como atingir um resultado, e sim a forma do resultado.

A diferença fundamental é que, nas linguagens lógicas, assume-se a existência de recursos suficientes para que seja determinada a forma de se atingir um certo resultado.

Para tanto, tudo que deve existir é um meio conciso de fornecer as informações relevantes no contexto do problema a ser resolvido, e um mecanismo de inferência para se chegar aos resultados desejados.

O cálculo de predicados provê a forma para comunicar as informações relevantes, e o mecanismo de resolução provê a técnica de inferência.

Anexo I: Referências

[Sebesta89] Sebesta, R.W., "Concepts of Programming Languages", The Benjamin/Cummings Publishing Company, Inc., 1989.

[Dershem/Jipping90] Dershem, H.L; Jipping, M.J., "Programming Languages: Structures and Models", Wadsworth, Inc., 1990.

Capítulo II:
A Linguagem
de Programação PROLOG

Introdução

A linguagem de programação de uso corrente que melhor representa o paradigma lógico é a linguagem Prolog, que é uma abreviação de programação em lógica.

Sua sintaxe é uma versão modificada de cálculo de predicados. Foi projetada no início dos anos 70, num trabalho conjunto de Alain Colmerauer e Phillippe Roussel, da Universidade de Aix-Marsseille, e de Robert Kowalski, da Universidade de Edimburgo. Sua primeira implementação data do ano de 1972.

Algumas Definições

Símbolo

O conceito de símbolo é idêntico ao conceito de átomo simbólico na linguagem LISP. Assim, os entendemos como algo que simboliza um objeto de interesse do mundo real.

Veja abaixo exemplos de símbolos:

```
jose    pera    cadeira    verde
```

Note que sempre grafamos símbolos empregando letras minúsculas.

Predicados

Entendemos que um predicado é uma forma de expressar uma propriedade de um símbolo ou um vínculo entre símbolos. Predicados tem forma funcional e podem não ter argumentos ou ter qualquer número de argumentos. Predicados podem ter múltiplas definições.

Veja abaixo exemplos de predicados:

```
pessoa (jose)      fruta (goiaba)      pai (joao, maria)
```

Eles representam, respectivamente, que:

- José é uma pessoa;
- Goiaba é uma fruta; e
- João é pai de Maria.

Note que sempre grafamos predicados empregando letras minúsculas.

Fatos

Um fato é uma afirmação a respeito de um ou mais símbolos. Um fato sempre é sempre representado por um predicado seguida por um ponto final.

Veja abaixo exemplos de fatos:

```
pessoa(jose).

fruta(goiaba).

pai(joao, maria).
```

Eles nos afirmam, respectivamente, que:

- José é uma pessoa;
- Goiaba é uma fruta; e
- João é pai de Maria.

Regras

Uma regra é uma afirmação condicional a respeito de um ou mais símbolos. Uma regra tem sempre a seguinte forma geral:

$P_1(E_{a1},E_{a2},...,E_{an})$:-[not]$P_2(E_{b1},E_{b2},...,E_{bm})${|,|;|[not]$P_3(E_{c1}, E_{c2},...,E_{ck})$}.

Onde os P_i's são predicados, os E_i's são os envolvidos nos predicados, o :- deve ser entendido como implicação, o not deve ser entendido como negação, a vírgula deve ser entendida com conjunção e o ponto-e-vírgula como disjunção.

Uma regra pode envolver símbolos e/ou variáveis. Quando temos um símbolo envolvido em uma regra, entendemos que afirmações estão sendo feitas a respeito de um objeto específico simbolizado por aquele símbolo. Quando temos uma variável envolvida em uma regra, entendemos que afirmações estão sendo feitas a um objeto genérico, não a um específico.

Veja abaixo alguns exemplos de regras:

```
bom_aluno(victor_hugo):-assiste_aulas(victor_hugo),
                        estuda_em_casa(victor_hugo).
```

```
avo(A,N):-pai(A,P),pai(P,N).
```

Note que sempre grafamos símbolos e predicados com letras minúsculas e variáveis com letras maiúsculas.

SWI-Prolog

Comentários

Comentários em SWI-PROLOG são como em C, i.e., tudo quando estiver delimitado pelos caracteres `/*` e `*/` será considerado um comentário.

Exemplo de um Programa

```
pai(jose,andre_luis).
pai(jose,jose_luis).
pai(andre_luis, victor_hugo).
avo(A,N):-pai(A,P),pai(P,N).
```

Para carregar um Programa

Gravar o programa em um arquivo com extensão .PL. Supondo que a pasta Bin do SWI-Prolog foi colocada no Path e que seu programa foi gravado em um arquivo de nome Progrma.PL, para acionar o SWI-Prolog, deve-se dar o seguinte comando:

PlWin –f Programa.PL

Exemplo de Execução

```
?- pai(jose,andre_luis).

Yes
?- pai(jose,Quem).

Quem = andre_luis ;
Quem = jose_luis ;

No
?- pai(P,F).
P = jose          F = andre_luis ;
P = jose          F = jose_luis ;
P = andre_Luis    F = victor_hugo ;
```

```
No
?- pai(andre_luis,_).

Yes
?- pai (jose,Q),pai(Q,victor_hugo).
Q = andre_luis ;

No
?-
```

Predicados Predefinidos

Verificação de Tipo

1. **var**(TERMO**)**

   Sucede se TERMO for uma variável livre;

2. **nonvar**(TERMO**)**

   Sucede se TERMO não for uma variável livre;

3. **integer**(TERMO)

   Sucede se TERMO representar um número inteiro;

4. **float**(TERMO)

   Sucede se TERMO representar um número real;

5. **number**(TERMO)

   Sucede se TERMO representar um número (inteiro ou real);

6. **atom**(TERMO)

   Sucede se TERMO representar um símbolo;

7. **string**(TERMO)

   Sucede se TERMO representar uma cadeia de caracteres;

8. **atomic**(TERMO)

   Sucede se TERMO representar um símbolo, uma cadeia de caracteres, um número inteiro ou um número real;

9. **compound**(TERMO)

   Sucede se TERMO for composto, i.e., representar uma lista ou um functor;

10. **ground**(TERMO)

    Sucede se TERMO não estiver associado a uma variável livre.

Comparação e Unificação

Termos compostos são primeiramente verificados levando-se em conta, nesta ordem, (1) a quantidade de seus argumentos; (2) seu nome (alfabeticamente); (3) recursivamente, seus argumentos, da esquerda para a direita.

1. TERMO1 **==** TERMO2

   Sucede se TERMO1 for igual a TERMO2.

2. TERMO1 **\==** TERMO2

   Sucede se TERMO1 não for igual a TERMO2. Tem o mesmo significado que \TERMO1 == TERMO2.

3. TERMO1 **=** TERMO2

   Unifica TERMO1 com TERMO2. Sucede se a unificação tiver sucesso.

4. unify_with_occurs_check(TERMO1,TERMO2)

   Unifica TERMO1 com TERMO2, se assegurando que TERMO1 não ocorra em TERMO2. Sucede se a unificação tiver sucesso.

5. TERMO1 **\=** TERMO2

   Sucede se a unificação de TERMO1 com TERMO2 não for possível. Tem o mesmo significado que \TERMO1=TERMO2.

6. TERMO1 **=@=** TERMO2

   Sucede se TERMO1 for estruturalmente igual a TERMO2.

7. TERMO1 **\=@=** TERMO2

   Sucede se TERMO1 não for estruturalmente igual a TERMO2. Tem o mesmo significado que \TERMO1 **=@=** TERMO2.

8. TERMO1 **@<** TERMO2

   Sucede se TERMO1 for menor que TERMO2.

9. TERMO1 **@=<** TERMO2

   Sucede se TERMO1 for menor ou igual a TERMO2.

10. TERMO1 **@>** TERMO2

    Sucede se TERMO1 for maior que TERMO2.

11. TERMO1 **@=>** TERMO2

    Sucede se TERMO1 for maior ou igual a TERMO2.

Predicados de Controle

1. **fail**

   Sempre falha.

2. **true**

   Sempre sucede.

3. **repeat**

   Sempre sucede e prove um número infinito de pontos de escolha.

4. **!**

   Cut. Descarta pontos de escolha que o precedem na cláusula.

Operadores

Aritméticos

Em SWI-Prolog os operadores aritméticos são os seguintes: ****** ou ^ (potência), ***** (multiplicação), + (adição), - (menos unário), - (subtração), / (divisão), // (divisão inteira); **mod** (resto da divisão inteira).

De Bit

Em SWI-Prolog os operadores binários são os seguintes: /\ (and bit a bit), \/ (or bit a bit), **xor** (xor bit a bit), \ (not bit a bit), << (deslocamento de bits para a esquerda) e >> (deslocamento de bits para a direita).

Funções Matemáticas

1. **abs**: valor absoluto;

2. **acos**: inverso do cosseno;
3. **asin**: inverso do seno;
4. **atan**: inverso da tangente;
5. **ceil** ou **ceiling**: arredondamento para o próximo inteiro;
6. **cos**: cosseno;
7. **e**: constante de Neper;
8. **exp**: exponenciação (base e);
9. **float**: conversão explícita para real;
10. **float_fractional_part**: parte fracionária de um real;
11. **float_integer_part**: parte inteira de um real;
12. **floor**: arredondamento para o inteiro predecessor;
13. **integer** ou **round**: arredondamento para o inteiro mais próximo;
14. **log**: logarítmo natural;
15. **log10**: logarítmo de base 10;
16. **max**: máximo de dois números;
17. **min**: mínimo de dois números;
18. **random**: gera um número aleatório;
19. **rem**: resto da divisão inteira;
20. **truncate**: elimina a parte fracionária;
21. **pi**: constante pi;
22. **sin**: seno;
23. **sqrt**: raiz quadrada;
24. **tan**: tangente.

Functors

Functors são domínios que representam argumentos de predicados que são eles próprios predicados.

Exemplo de Programa

```
pagou(jose,comida(giovanetti,100.00)).
pagou(jose,comida(nacional,50.00)).
pagou(jose,aluguel(otot,1000.00)).
pagou(jose,loja(renner,70.00)).
pagou(jose,loja(skina,50.00)).

pagou(raul,comida(nacional,30.00)).
pagou(raul,comida(allesbier,100.00)).
pagou(raul,loja(skina,50.00)).

pagou(joao,comida(nacional,40.00)).
pagou(joao,comida(allesbier,70.00)).
pagou(joao,comida(nacional,50.00)).
pagou(joao,aluguel(serra,1000.00)).
pagou(joao,loja(americana,70.00)).
pagou(joao,loja(skina,30.00)).
```

Exemplo de Execução

```
?- pagou(jose,Oque).
Oque = comida(giovanetti,100.00) ;
Oque = comida(nacional,50.00) ;
Oque = aluguel(otot,1000.00) ;
Oque = loja(renner,70.00) ;
Oque = loja(skina,50.00) ;

No
?- pagou(Quem,comida(allesbier,Quanto)).
Quem = raul    Quanto = 100.00 ;
Quem = joao    Quanto = 70.00 ;

No
```

Listas

Listas são coleções de elementos e mesma natureza. Uma lista é representada por uma série de símbolos separados por vírgulas e entre colchetes. Veja o exemplo abaixo:

```
[maca, uva, pera, goiaba, abacaxi]
```

Para manipularmos uma lista, devemos promover uma unificação entre a lista e um padrão. Por exemplo, se desejássemos saber o primeiro elemento da lista acima e o que resta dela retirado o primeiro elemento faríamos:

```
[maca, uva, pera, goiaba, abacaxi] = [Primeiro | Resto]
```

e teríamos a variável `Primeiro` unificada com o símbolo maca e a variável `Resto` unificada com a lista `[uva, pera, goiaba, abacaxi]`.

Um outro exemplo seria se desejássemos saber os 3 primeiros elementos da lista acima e o que resta dela retirados os 3 primeiros elementos faríamos:

```
[maca, uva, pera, goiaba, abacaxi] = [P, S, T | Resto]
```

e teríamos a variável `P` unificada com o símbolo `maca`, a variável `S` unificada com o símbolo `uva`, a variável `T` unificada com o símbolo `pera` e a variável `Resto` unificada com a lista `[goiaba, abacaxi]`.

Vale ressaltar a possibilidade de fazermos em SWI-PROLOG listas de listas.

Exemplo de Programa 1

Considere o programa abaixo que implementa um predicado que recebe dois parâmetros, (1) e (2), ambos listas de números inteiros, sucedendo, se ambas as listas forem idênticas, e falhando, caso contrário.

```
identicas([],[]).
identicas([P1|R1],[P2|R2]):-P1==P2,identicas(R1,R2).
```

Exemplo de Execução

```
?- identicas([1,2,3],[1,2]).

No
?- identicas([1,2],[1,2,3]).

No
?- identicas([1,2,3],[1,2,4]).

No
?- identicas([1,2,3],[1,2,3]).

Yes
```

Exemplo de Programa 2

Considere o programa abaixo que implementa um predicado que recebe uma lista de símbolos e um símbolo, e instancia uma variável com a lista que se obtém retirada a primeira ocorrência do dado símbolo da referida lista.

```
remove ([P|R],P,R).
remove ([P|R],E,[P|NR]):-P \== E,remove(R,E,NR).
```

Exemplo de Execução

```
?- remove([maca,uva,pera,uva,abacaxi],uva,S).
S = [maca, pêra, uva, abacaxi]

Yes
```

Anexo I: Exercícios Propostos

1. Escreva um programa SWI-Prolog que implemente um predicado que recebe dois parâmetros, (1) um número inteiro; e (2) uma lista de números inteiros. O predicado deverá ter sucesso, no caso do referido número inteiro pertencer à lista de números inteiros dada, e insucesso, no caso contrário.

2. Escreva um programa em SWI-Prolog que implemente um predicado que recebe três parâmetros: (1) um símbolo; (2) uma lista de símbolos; e (3) uma variável não instanciada. O predicado deverá instanciar a variável em questão com a lista que se obtem da remoção de todas as ocorrências do referido símbolo da lista de símbolos dada.

3. Escreva um programa SWI-Prolog que implemente um predicado que recebe três parâmetros: (1) um número inteiro; (2) uma lista de números inteiros; e (3) uma variável não instanciada. O predicado deverá instanciar a variável em questão com a lista que se obtem da inclusão ordenada do referido número inteiro na lista de números inteiros dada.

4. Escreva um programa SWI-Prolog que implemente um predicado que recebe três parâmetros: os parâmetros (1) e (2) serão listas de números inteiros; e o parâmetro (3) será uma variável não instanciada. O predicado deverá instanciar a variável em questão com a lista que se obtem da concatenação das duas listas de números inteiros dadas.

5. Escreva um programa SWI-Prolog que implemente um predicado que recebe dois parâmetros: (1) uma lista de números inteiros; e (2) uma variável não instanciada. O predicado deverá instanciar a variável em questão com a lista que se obtem da ordenação da lista de números inteiros dada.

6. Escreva um programa SWI-Prolog que implemente um predicado que recebe três parâmetros: (1) uma lista de números inteiros; e (2) uma variável não instanciada. O predicado deverá instanciar a variável em questão com a lista que se obtem da retirada do elemento mais à direita da referida lista.

7. Escreva um programa SWI-Prolog que implemente um predicado que recebe dois parâmetros: (1) uma lista de números inteiros; e (2) uma variável não instanciada. O

predicado deverá instanciar a variável em questão com o maior número inteiro da lista de números inteiros dada.

8. Escreva um programa SWI-Prolog que implemente um predicado que recebe dois parâmetros: (1) uma lista de números inteiros; e (2) uma variável não instanciada. O predicado deverá instanciar a variável em questão com a lista que se obtem da retirado do maior número inteiro da lista de números inteiros dada.

9. Escreva um programa SWI-Prolog que implemente um predicado que recebe dois parâmetros: (1) uma lista de números inteiros; e (2) uma variável não instanciada. O predicado deverá instanciar a variável em questão com o número real que representa a média aritmética dos elementos da lista de números inteiros dada.

Anexo II: Referências

[Sebesta89] Sebesta, R.W., "Concepts of Programming Languages", The Benjamin/Cummings Publishing Company, Inc., 1989.

[Baranauskas93] Baranauskas, M.C.C., "Procedimento, Função, Objeto ou Lógica? Linguagens de Programação vistas pelos seus Paradígmas" *in* Computadores e Conhecimento - Repensando a Educação, Valente, J.A. (Org.), NIED, UNICAMP, 1993.