Universidade de Aveiro

Departamento de Matemática

Texto de apoio à disciplina de

Introdução à Programação em Lógica

Delfim Fernando Marado Torres

Matemática Aplicada e Computação

1999

<delfim@mat.ua.pt>
Last Revision Date: 16/July/1999
<http://www.mat.ua.pt/delfim>

Intróito

A linguagem Prolog (PROgramar em LOGica) representa um paradigma de programação diferente do estilo procedimental mais conhecido. Assim, o uso do Prolog proporciona-nos, como se espera tornar claro ao longo do presente texto, uma abordagem aos problemas diferente. Se nos primórdios do Prolog, na década de 70, a sua utilização se restringia à comunidade de Inteligência Artificial (IA), a qual era, e ainda é, relativamente pequena quando comparada com outras comunidades, nos dias que correm, e não obstante o facto do Prolog ser uma linguagem pouco eficiente e pouco apropriada para cálculos numéricos, o interesse nesta linguagem transborda claramente os limites da comunidade IA, na medida em que lhe foram reconhecidas qualidades indiscutíveis na fase do ciclo de desenvolvimento de software correspondente à *prototipagem*.

O gosto pela "Programação em Lógica" nasceu em mim quando aluno da licenciatura em Eng. Informática na Universidade de Coimbra. No quarto ano tive de escolher a especialização (ramo) que pretendia: a escolha foi fácil —Computação. Foi esta escolha que me permitiu entrar em contacto com o Prolog, a computação simbólica, a Inteligência Artificial, os Sistemas Periciais e as técnicas matemáticas da Computação. Quando vim trabalhar para o Departamento de Matemática da Universidade de Aveiro, no início do ano lectivo de 1994/95, o curso de Matemática Aplicada e Computação (MAC) estava apenas em funcionamento há um ano. O Prof. Domingos Cardoso tinha previsto uma cadeira de "Introdução à Programação em Lógica" (IPL) no terceiro ano do currículo de MAC, tendo chegado mesmo a escrever um texto ([4]) com as matérias programáticas que achava adequadas para tal curso. Tive a oportunidade e o prazer de leccionar as aulas teórico-práticas, e no presente ano lectivo de 1998/99 também as teóricas, durante todas as "edições" da cadeira de IPL. Na primeira edição de 1995/1996, as aulas teóricas estiveram a cargo do Mestre Tim Hultberg. Nos dois anos lectivos seguintes, foi convidado o Prof. Pedro Rangel Henriques, do Departamento de Informática da Universidade do Minho, para a leccionar. Foi o Prof. Pedro Rangel Henriques que veio dar uma dinâmica muito própria ao curso, muito na linha do Deransart ([8]) do Departamento de Matemática e Informática da Universidade de Orleães. A abordagem adoptada revelou-se, na nossa opinião, muito eficaz, levando os alunos a interiorizar muito bem os conceitos da programação declarativa segundo o paradigma lógico; a dominar o uso de predicados (cláusulas de Horn), unificação e recursividade na resolução de problemas; e a estudar e a implementar estruturas de dados lineares e não-lineares com os respectivos algoritmos de manuseamento em *Programação Lógica*, desenvolvendo correcta e eficientemente programas na linguagem Prolog.

O presente texto segue a estrutura do curso de IPL tal como foi leccionado pelo Prof. Pedro Rangel Henriques e por mim nos anos lectivos de 1996/97 e 1997/98. Muitos dos exercícios propostos no final de cada capítulo são da autoria do Prof. Pedro Rangel. O texto foi escrito no decorrer do segundo semestre de 1998/99, para servir de apoio ao curso de IPL. À medida que o ia escrevendo, ia-o disponibilizando no endereço

`<http://www.mat.ua.pt/delfim/cadeiras/ipl99/ipl.htm>`

contribuindo assim para uma experiência de *"Ensino à Distância"* via World Wide Web que aí procurei preconizar. Será também neste local onde se poderá, no futuro, encontrar as alterações ou acrescentos ao presente texto que vierem a ser aconselháveis. Todos os comentários serão sempre bem vindos e poderão ser feitos por correio electrónico para `<delfim@mat.ua.pt>`. Desde já obrigado.

Não obstante os fundamentos teóricos que apoiam o Prolog estarem enraizados no 'Cálculo de Predicados de Primeira Ordem', não pressupomos aqui qualquer tipo de formação prévia

em Lógica da parte dos leitores. O presente texto é auto-contido e estamos convictos de que qualquer pessoa interessada, independentemente da formação anterior, estará apta às aprendizagens que aqui nos propomos ensinar. O material apresentado é, nem mais nem menos, do que aquele que foi leccionado em 1999 com uma escolaridade, por semana, de 3 horas de aulas Teóricas e 2 horas de aulas Teórico-Práticas.

O texto está organizado em essencialmente três partes. Numa primeira parte, *"Aulas Teóricas"*, tenta-se motivar o leitor para as matérias em questão, apresenta-se a matéria fundamental, os conceitos, as definições, os métodos e justificações. A segunda parte, *"Aulas Teórico-Práticas"*, é constituída por guiões detalhados para as aulas teórico-práticas. Estas aulas destinam-se à resolução de exercícios de consolidação no quadro e no computador. Por último na terceira parte, *"Enunciados de Trabalhos e Exames"*, além do exame Final e de Recorrência de 1999, são apresentados os trabalhos que têm vindo a ser propostos aos alunos, extra aulas, na forma de dois projectos a serem apresentados pelos vários grupos de dois alunos durante o semestre, a funcionar no computador e acompanhados dum pequeno relatório de desenvolvimento. Esta componente prática teve sempre um grande peso na nota final (50%) e mereceu, por parte dos discentes, um grande envolvimento com resultados francamente surpreendentes.

Prova do empenhamento dos alunos de IPL, no qual os referidos projectos práticos tiveram com certeza um papel importante, foi o entusiasmo com que um grupo de alunos de IPL, do ano lectivo de 1996/97, deram andamento, em 1998, a uma ideia do Prof. Pedro Rangel Henriques: a organização de um concurso/encontro nacional de Programação em Lógica. Com o apoio do Conselho Directivo do Departamento de Matemática da Universidade de Aveiro (na altura presidido pelo Prof. Helmuth Malonek) e da reitoria da UA, esse grupo de alunos promoveram umas jornadas tipo Olimpíadas de 26 a 28 de Abril de 1998, entre alunos das diversas universidades e politécnicos do país. O mote foi a resolução de problemas usando a programação lógica (Prolog), promovendo a boa convivência/competição entre alunos e o uso da programação lógica na resolução de "charadas" e desafios intelectuais. Tendo nascido aqui no Departamento, o Concurso/Encontro Nacional de Programação em Lógica (CeNPL) passou agora a ser organizado numa base anual pelas várias universidades do país e tudo indica estar a tornar-se num evento mediático. O CeNPL'99 realizou-se de 13 a 15 de Abril de 1999 na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, enquanto o próximo concurso, CeNPL'2000, será realizado no Departamento de Informática da Universidade do Minho.

Quase a terminar, quero agradecer aos meus professores de licenciatura, Prof. Amílcar Cardoso e Prof. Ernesto Costa, pela forma como, tão bem, souberam incutir o gosto nestas matérias; e deixar um agradecimento muito especial ao Prof. Pedro Rangel Henriques pela amizade e por todas as empolgantes e gratificantes conversas que me proporcionou. É a ele que se deve o presente texto. Resta-me agradecer ao Mestre Eugénio Rocha e ao Prof. Domingos Cardoso pela leitura de uma pré-versão deste texto e aos alunos da cadeira de Introdução à Programação em Lógica, quer do ano lectivo de 1998/1999 quer de anos anteriores, por todas as questões que colocaram e pela forma interessada com que acompanharam a matéria.

Por último quero deixar uma palavra (apelo) aos futuros leitores destas notas: divirtam-se com o Prolog, a Informática e a Matemática e, porque não, participem nos próximos CeNPL's!

Delfim F. Marado Torres

Aveiro,
Julho de 1999.

Conteúdo

Aulas Teóricas

1

Significado lógico de um programa em lógica

1. Introdução intuitiva ao formalismo

Vamos usar a *programação em lógica* com a finalidade de resolver problemas. Cada *problema* consistirá num certo número de *questões* referentes a certos *objectos*. Esses objectos verificam certas *propriedades* e conservam entre si certas *relações*. Os objectos, com as suas propriedades e relações, constituem o *universo* do problema.

1.1. Problema 1. O problema consiste em estudar as relações de parentesco. Os objectos do universo serão certos humanos.

O *programa em lógica* exprimirá o conhecimento que temos das propriedades e relações do universo. Ao *descrevermos* o universo do problema, pretendemos depois obter, de um modo automático, a solução do problema, quer dizer, as *respostas* às questões.

1.2. Conceitos e definições. O conhecimento associado a um problema é potencialmente infinito, mas para o formular, bem como para formular as questões, deveremos utilizar recursos finitos, mais precisamente, uma *linguagem* lógica. Aos constituintes elementares da linguagem chamamos *símbolos*.

Certos símbolos são chamados de *constantes*. Estes são os símbolos que representam os objectos "privilegiados" do universo. Exemplos de constantes para o Problema n^o1 podem ser:

```
ana  zulmira  luis  eusebio
```

Utilizamos também outros símbolos chamados *variáveis*. As variáveis são os símbolos que representam qualquer objecto do universo. Uma variável servirá numa questão, por exemplo, para representar um objecto desconhecido. Elas serão também necessárias para formular propriedades que se verificam para todos os objectos do universo. Por convenção, as variáveis serão os símbolos que começam por uma letra maiúscula. Seguem-se exemplos de variáveis para o Problema n^o1:

```
Filho  F  Mae  M
```

Constantes e variáveis são dois casos particulares de *termos*.

Os termos representam os objectos do universo. Contudo, um termo que é uma variável, não representa um objecto bem determinado do universo, a não ser que fixemos uma certa *afectação*. Uma afectação, em lógica, é uma função que associa ("afecta") uma variável a um objecto fixo do universo, ao qual chamamos então de *valor* da variável por essa afectação.

Um termo que é uma constante, tem também um valor, valor esse independente de toda a afectação: é sempre o objecto do universo representado por essa constante. Por exemplo, o termo `ana` tem por valor uma rapariga, ou uma senhora, bem determinada (pressupondo a linguagem bem escolhida...).

É importante notar que os objectos do universo não são necessariamente todos os valores das constantes. Por exemplo, para o nosso problema n^o1, podem existir humanos que não têm

o "privilégio" de estar representados pelas constantes da linguagem. No entanto, um qualquer objecto do universo pode sempre ser o valor de uma variável. Basta fixar convenientemente uma afectação.

Em lógica, chamamos às propriedades e relações *predicados*. Certos símbolos da linguagem são designados *símbolos de predicados*: são os símbolos que representam os predicados do universo. Eis alguns exemplos de símbolos de predicados para o nosso Problema 1:

```
feminino
```

para representar a propriedade "é do sexo feminino";

```
pai
```

para representar a relação "é pai de".

Dizemos que a relação "é pai de" é uma relação binária, ou de *aridade* 2, porque ela relaciona 2 objectos. Por outro lado, uma propriedade como "é do sexo feminino", é de aridade 1. Cada símbolo de predicado tem portanto uma certa aridade. Para tornar explícita a aridade, escrevemos: `feminino/1`, `pai/2`.

Com os símbolos dos predicados e dos termos, construímos os *atomos@átomos*. Alguns exemplos de átomos para o Problema 1, são:

```
feminino(ana)
pai(luis, eusebio)
pai(P,F)
```

De maneira geral, um átomo escreve-se com um símbolo de predicado e um número de termos igual à aridade do símbolo do predicado em questão (bem como parêntesis curvos e eventualmente vírgulas). Estes termos são os *argumentos* do átomo.

Um *atomo fechado@átomo fechado* é um átomo que não contém qualquer variável. No universo, das duas uma: ou ele é *verdadeiro* ou então *falso*. Por exemplo, para o problema considerado, o átomo `pai(luis,eusebio)` é verdadeiro se e somente se o humano representado pela constante `luis` é o pai do humano representado pela constante `eusebio` (estamos a usar a convenção, arbitrária, que o primeiro argumento (aqui `luis`) corresponde ao pai e que o segundo argumento (aqui `eusebio`) corresponde ao filho).

De um modo geral, um átomo fechado é verdadeiro no universo considerado, se os valores dos seus termos verificam o predicado representado pelo símbolo de predicado do átomo em questão.

Para um átomo não fechado, quer dizer, com pelo menos uma variável, tudo isto não faz sentido senão para uma afectação fixa, que determina um valor para cada variável.

Para o Problema n^o1, consideremos um predicado de aridade 3, denotado por `progenitores`, para representar a seguinte relação: o átomo

```
progenitores(M,P,F)
```

é verdadeiro, para uma certa afectação, se, e somente se, o valor da variável `F` for filho da mãe representada pelo valor da variável `M` e do pai representado pelo valor da variável `P`.

Consideremos, também, dois símbolos de predicados de aridade 2: `mae` e `progenitor`, para representar, respectivamente, as relações "é a mãe de" e "é um dos dois progenitores de".

Com os átomos, construímos as *cláusulas* (a terminologia exacta é a de "cláusulas definidas", mas como são as únicas cláusulas consideradas neste curso, serão denotadas simplesmente por *cláusulas*). Alguns exemplos de cláusulas para o problema considerado são:

```
progenitores(M,P,F) :- mae(M,F), pai(P,F).
progenitor(B,A) :- mae(B,A).
progenitor(B,A) :- pai(B,A).
pai(luis,ana) :-.
```

Uma cláusula começa sempre por um átomo, chamado de *cabeça* da cláusula, seguida de `:-` e de uma sucessão finita de átomos, apelidada de *corpo* da cláusula. Uma cláusula termina sempre com um ponto. Os átomos do corpo são separados por vírgulas. Pode no entanto acontecer o corpo ser vazio como a última cláusula acima.

As cláusulas com corpo vazio são chamadas de *factos* enquanto às outras apelidamos de *regras*. Para simplificar a notação escrevemos `pai(luis,ana).` em vez de `pai(luis,ana) :-.`

Para uma afectação fixa, cada átomo de uma cláusula é, ou verdadeiro, ou falso, no universo considerado. A cláusula, ela mesma, terá, no universo, um valor lógico, ou verdadeiro ou falso:

> a cláusula é falsa se, e somente se, a cabeça é falsa e todos os átomos do corpo são verdadeiros.

Isto não é mais do que uma maneira rigorosa de dizer que `:-` deve ser visto como o sinal de implicação $\Leftarrow$ da lógica, a vírgula como o $\wedge$ lógico, e a cláusula como a seguinte implicação: Se todos os átomos do corpo são verdadeiros Então o átomo da cabeça é verdadeiro. No entanto, é importante notar que quando o corpo é vazio, a cláusula (facto) é verdadeira se, e somente se, a cabeça é verdadeira.

No universo do Problema n^o1, a cláusula

```
progenitores(M,P,F) :- mae(M,F), pai(P,F).
```

é verdadeira (não importa qual a afectação).

Uma cláusula é *válida* no universo em questão, se ela é sempre verdadeira independentemente da afectação: em lógica dizemos que as variáveis da cláusula são quantificadas universalmente. A cláusula precedente é então válida no universo contextual do Problema 1.

De modo semelhante, um átomo é *válido* no universo, se ele for verdadeiro qualquer que seja a afectação considerada. Um facto é então válido se, e somente se, o seu átomo de cabeça é válido.

Para um átomo ou uma cláusula que não possuam variáveis (átomo fechado, cláusula fechada) ser verdadeira ou falsa não depende absolutamente nada das afectações e, por conseguinte, o conceito de validade no universo não faz intervir o de afectação. Por exemplo, no nosso Problema 1, a cláusula

```
pai(luis, ana) :- .
```

isto é, o facto

```
pai(luis, ana).
```

e o átomo `pai(luis, ana)` são válidos no universo se, e somente se, o humano (valor de) `luis` é o pai do humano (valor de) ana.

Para o Problema 1, consideramos agora um símbolo de predicado de aridade 2, chamado `avo`, para representar a relação "é um dos avós de". No universo do problema, a cláusula seguinte é válida:

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

Do que foi dito, esta cláusula pode ser lida como:

> "Qualquer que sejam `A`, `P` e `N`, se `progenitor(A,P)` e `progenitor(P,N)` são ambos verdadeiros então `avo(A,N)` é verdadeiro".

O mesmo é dizer:

> "Qualquer que sejam `A` e `N`, se existe um `P` tal que `progenitor(A,P)` e `progenitor(P,N)` são ambos verdadeiros então `avo(A,N)` é verdadeiro".

Esta dupla leitura (mudando "qualquer que seja `P`" para "existe um `P`") pode ser justificada rigorosamente (logicamente) sempre que uma variável (no nosso caso `P`) aparecer no corpo da cláusula sem aparecer na cabeça.

Chamamos *programa em lógica* (em Prolog) a uma sucessão finita de cláusulas. Na prática essas cláusulas são agrupadas do seguinte modo: juntamos aquelas que começam pelo mesmo símbolo de predicado na cabeça.

Eis o nosso primeiro programa em Prolog para o Problema 1:

```
mae(aldina,ana).
mae(aldina,eusebio).
mae(rosa,luis).
mae(ines,aldina).

pai(luis,ana).
pai(luis,eusebio).
pai(paulo,luis).
pai(gustavo,aldina).
```

```
progenitores(M,P,F) :- mae(M,F), pai(P,F).

progenitor(P,F) :- mae(P,F).
progenitor(P,F) :- pai(P,F).

avo(A,N) :- progenitor(A,B), progenitor(B,N).
```

2. O que se espera de um programa em lógica

O universo de que falamos é o universo contextual do problema, que comporta um conjunto (não vazio!) de questões às quais procuramos responder. Uma *questão* pode ser representada recorrendo aos átomos. Segue-se um exemplo de uma possível questão no universo do Problema 1:

```
?- avo(rosa,X).
```

O significado é: "Quais são os objectos `X`, do universo, para os quais o átomo `avo(rosa,X)` é verdadeiro?" O mesmo é dizer, numa linguagem mais natural: "Quem são os netos de rosa?"

Suponhamos que sabemos que os átomos `avo(rosa,ana)` e `avo(rosa,eusebio)` são válidos no universo. Então uma *resposta* possível à questão é

```
X = ana
```

e uma outra é

```
X = eusebio
```

Consideremos agora a questão

```
?- avo(Y,ana).
```

Uma resposta possível é

```
Y = rosa
```

Se a questão for

```
?- avo(rosa,ana).
```

a resposta é uma simples confirmação:

```
yes
```

Podemos então considerar que temos uma solução do problema, se soubermos quais são os átomos válidos no universo. O que esperamos de um programa em lógica (o que esperamos do interpretador Prolog) é que ele seja capaz de *produzir* esses átomos.

Falta definir o que entendemos por "átomos produzidos por um programa". Para tal, necessitamos dar um significado, uma *semântica*, ao programa. Iremos dar várias definições deste significado, começando pelo mais natural no contexto da lógica e acabando com a mais operacional. No entanto estas definições serão, num certo sentido, equivalentes.

3. Regresso ao formalismo

A noção de *consequência lógica* é uma noção intuitivamente clara e é rigorosamente definida em lógica. Intuitivamente, um átomo `A` é consequência lógica de um programa `P`, se podermos demonstrar que `A` é válido, qualquer que seja a interpretação dos símbolos, com o único pressuposto que as cláusulas de `P` são válidas. Limitamo-nos aqui a esta ideia intuitiva de consequência lógica.

Alguns exemplos de átomos que são consequência lógica do programa Prolog acima são:

```
mae(rosa,luis), pai(paulo,luis),
progenitores(rosa,paulo,luis), progenitor(rosa,luis),
pai(luis,ana), progenitor(luis,ana), avo(rosa,ana).
```

Podemos achar estes átomos, aplicando a definição de "consequência lógica" dada na cadeira de "Lógica de Predicados de Primeira Ordem". Será, no entanto, mais cómodo com a definição construtiva, equivalente, que damos à frente.

Denotamos por *CL(P)*, o conjunto de átomos que são consequência lógica do programa *P*. É precisamente este conjunto *CL(P)*, que consideramos como sendo o conjunto dos átomos "produzidos" por *P*. Desta maneira, damos um significado lógico a um programa.

4. Exercícios

Nos exercícios que se seguem, descreva o universo do discurso, identificando os objectos/entidades, as suas propriedades e relações, ... Defina depois os símbolos de predicados (não se esquecendo de indicar as respectivas aridades), as constantes, ... Especifique, por fim, um programa Prolog. Dê depois exemplos de questões e possíveis respostas.

Exercício do Exame Final de 1997: Problema 2. Para construir um Sistema de Informação para uma Unidade de Arqueologia, foram identificados as seguintes Entidades e Relações:
o Arqueólogo (descrito pelo código interno, nome e instituição onde trabalha) que *fez* uma Escavação (caracterizada por um identificador, local onde se situa e sector onde se está a escavar) e o Achado (descrito por um identificador, uma classe (tipo de objecto), o material e o estado de conservação) que é *encontrado* pelo Arqueólogo nessa Escavação.

Antes de se criar a Base de Dados Relacional, que servirá de suporte ao sistema informático final, está-se a pensar fazer um protótipo do sistema num ambiente de programação declarativo, lógico.

Escolha os **predicados** (indique o seu *nome* e todos os seus *argumentos*) que deve usar em Prolog para modelar as entidades e as relações entre elas (para concretizar a sua resposta, mostre, como exemplo, 1 ou 2 factos concretos para cada um desses predicados).

Exercício do Exame Final de 1997: Problema 3. Numa empresa fabricante de equipamento eléctrico para cozinhas, encontram-se as seguintes proposições sobre a constituição de um **fogão**:

- Um fogão é composto por uma estrutura e um cordão eléctrico.
- Umas das componentes da estrutura é uma resistência de aquecimento.
- A resistência de aquecimento é em metal.
- Outra parte da estrutura é o painel do fogão.
- O painel tem um botão.
- Os botões são sempre feitos em plástico.
- O cordão eléctrico é composto de fio metálico.
- Parte do cordão eléctrico é um isolador.
- O isolador é feito de fibra plástica.

Responda, então, às alíneas seguintes:

a): Recorrendo apenas aos predicados `parte_de` e `feito_em` (note que uma palavra diferente não corresponde necessariamente a um predicado ou argumento distinto), escreva um conjunto de cláusulas Prolog que descrevam precisamente o conhecimento contido nas frases acima.

b): Tomando em consideração a Base de Conhecimento criada na alínea anterior, diga como procederia, face a um Interpretador de Prolog, para obter resposta à seguinte questão: Que objectos (simples) são de metal ?

Exercício do Exame de Recorrência de 1997: Problema 4. Para construir um Sistema de Informação para uma Biblioteca, foram identificados as seguintes Entidades e Relações:
o Livro (descrito pelo ISBN, titulo, assunto e editora) que *é escrito* por um Autor (caracterizado por um nome, o país de onde é natural, e o estilo principal) e o Utente (descrito por um código, o nome, a classe (docente, aluno, externo), e o telefone) que *requisita* um livro numa data para *leitura interna*, ou *domiciliária*.

Antes de se criar a Base de Dados Relacional, que servirá de suporte ao sistema informático final, está-se a pensar fazer um protótipo do sistema num ambiente de programação declarativo, lógico.

Escolha os **predicados** (indique o seu *nome* e todos os seus *argumentos*) que deve usar em Prolog para modelar as entidades e as relações entre elas (para concretizar a sua resposta, mostre, como exemplo, 1 ou 2 factos concretos para cada um desses predicados).

Exercício do Exame de Recorrência de 1997: Problema 5. No livro *100 Jogos Lógicos* da colecção *O Prazer da Matemática* da Gradiva, pode ser encontrado o jogo que se segue.

Um motorista exprime as suas impressões sobre automóveis da seguinte forma:

- Uma tracção à frente dá boa estabilidade;
- Uma viatura pesada deve ter bons travões;
- Todas as viaturas de motor muito potente são caras;
- As viaturas ligeiras não têm boa estabilidade;
- Uma viatura de motor pouco potente não pode ter bons travões.

Admitirá ele a existência de uma viatura de tracção à frente que seja cara?

Responda, então, às alíneas seguintes:

a): Escreva cláusulas em Prolog que lhe permitam *classificar* viaturas de acordo com os seguintes parâmetros: tracção (à frente, ou a trás); potência (muita, média, pouca); classe (ligeiro, ou pesado, ...).

b): Recorrendo aos predicados que criou na alínea anterior, acrescente à BC mais cláusulas Prolog para traduzir as afirmações do motorista acima apresentadas, de tal modo que seja possível, posteriormente, fazer inferências sobre a estabilidade, preço, ou travões de uma viatura.

Exercício do Exame de Recurso de 1997: Problema 6. Para gerir uma Central de Reservas de Automóveis e de Hotéis portugueses foram identificadas as seguintes Entidades e Relações:
a Viatura (especificada pela marca, modelo, matrícula, cilindrada e número de lugares), o Hotel (descrito pelo nome, classificação (em número de estrelas), localização, telefone), o Cliente (caracterizado por um nome, bilhete de identidade, profissão, telefone), e a Reserva que é feita por um Cliente para uma Viatura, ou para um Hotel, e que é descrita por: um código de reserva; número de pessoas; data de aluguer (dia, mês); número de dias da ocupação; e, caso se trate de um Hotel, regista-se ainda o número de quartos e respectivo tipo (duplo, simples, ou suite).

Antes de se criar a Base de Dados Relacional, que servirá de suporte ao sistema informático final, está-se a pensar fazer um protótipo do sistema num ambiente de programação declarativo, lógico.

Escolha os **predicados** (indique o seu *nome* e todos os seus *argumentos*) que deve usar em Prolog para modelar as entidades e as relações entre elas (para concretizar a sua resposta, mostre, como exemplo, 1 ou 2 factos concretos para cada um desses predicados).

Exercício do Exame de Recurso de 1997: Problema 7. No regresso duma longa viagem por Portugal, Timóteo passou pela Central de Reservas (modelada no problema anterior) e fez, dos Hotéis que frequentou, as seguintes observações[1]:

- Sempre que a comida era boa, os empregados eram delicados;
- Todos os Hotéis abertos durante todo o ano tinham vista para o mar;
- A comida só era má em alguns hotéis baratos;
- Os Hotéis com piscina tinham os muros cobertos de madressilva;
- Os Hotéis que estavam abertos apenas durante uma estação do ano tinham empregados indelicados;
- Nenhum Hotel barato admitia a presença de cães;
- Os Hotéis sem piscina não tinham vista para o mar.

A dita Central de Reservas pretende, obviamente, incorporar na BC do seu sistema de prototipagem em Prolog estas informações preciosas. Responda então às alíneas seguintes:

a): Recorrendo apenas aos predicados de aridade 2 (cujo 1º argumento será sempre o Hotel em causa) `comida, empregados, abertura, custo, vista, piscina, muro, proibido` escreva cláusulas Prolog que traduzam as opiniões acima expressas pelo Timóteo.

b): Tomando em consideração o conjunto de cláusulas que criou na alínea anterior, diga como formulava ao Interpretador de Prolog (IP) a seguinte questão: "quais são os Hotéis que servem boa comida tendo os muros cobertos com madressilva?"

[1] Problema inspirado no Jogo 49 da livro *100 Jogos Lógicos* da colecção *O Prazer da Matemática* da Gradiva.

Mais um Exercício: Problema 8. Escreva um programa para que eu possa saber de quem é que eu gosto. (Os critérios são deixados ao seu gosto!)

Acrescente depois uma cláusula que traduza o seguinte conhecimento:

Eu convido para jantar quem eu gosto ou então quem preciso de convidar.

Defina, depois, o predicado "preciso de convidar". (Os critérios são, também aqui, deixados a seu gosto . . .)

2

Significado operacional não determinístico – árvore de prova

1. Substituição e instância

Usamos a notação

$$s = \{\texttt{X} \leftarrow \texttt{ana},\ \texttt{Z} \leftarrow \texttt{Y}\}$$

para designar uma função s do conjunto das variáveis no conjunto dos termos, que à variável `X` associa o termo `ana`, à variável `Z` associa o termo `Y` e deixa inalteradas as outras variáveis.

Pondo de parte as questões formais da possível confusão entre "linguagem" e "meta-linguagem", esta função s pode ser definida da maneira habitual: para toda a variável V

$$s(V) = \begin{cases} \texttt{ana} & \text{se} \quad V = X \\ \text{Y} & \text{se} \quad V = Z \\ \text{V} & \text{se} \quad V \neq X \text{ e } V \neq Z \end{cases}$$

Dizemos que s é uma *substituição*. Uma substituição aplica-se, de um modo natural, a termos, átomos ou cláusulas. Por exemplo, se aplicarmos a substituição s à cláusula

```
avo(X,Z) :- progenitor(X,Y), progenitor(Y,Z).
```

obteremos a cláusula

```
avo(ana,Y) :- progenitor(ana,Y), progenitor(Y,Y).
```

A esta cláusula resultante chamamos *instância* da primeira cláusula pela substituição s.

Uma instância por uma substituição, obtém-se substituindo as variáveis pelos termos indicados na substituição.

Se X_1, X_2, ... X_n são as únicas variáveis X para as quais $s(X) \neq X$ e se

$$s(X_1) = t_1,\ s(X_2) = t_2,\ \ldots\ s(X_n) = t_n$$

utilizamos a notação

$$s = \{X_1 \leftarrow t_1,\ \ldots,\ X_n \leftarrow t_n\}$$

A substituição identidade é denotada por { } e deixa inalterada todas as variáveis, e deste modo cada termo, átomo, cláusula, ...

A imagem de um termo (átomo, cláusula) por uma substituição é chamada instância do termo (átomo, cláusula) pela substituição.

Note-se que uma instância de uma instância de t é uma instância de t. Mais formalmente, a composição de duas substituições, como uma função sobre o conjunto dos termos (átomos, cláusulas) é ainda uma substituição.

Estas noções permitem dar um sentido rigoroso à noção de *questão* e *resposta*.

Seja `P` um programa e `A` um átomo. Uma *resposta lógica* (segundo `P`) à questão

```
?- A.
```

é uma substituição s tal que `s(A)`, instância de `A` por s, é uma consequência lógica de `P`, quer dizer, $s(\texttt{A}) \in CL(P)$.

Segue-se um exemplo para o nosso problema 1: à questão

```
?- avo(rosa,N).
```

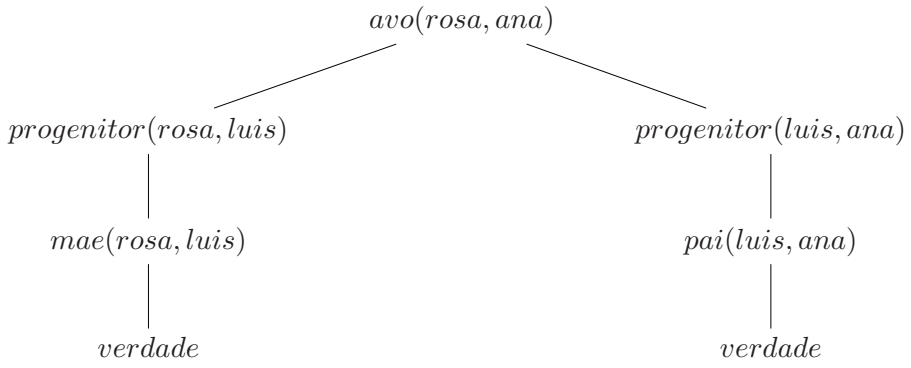
uma das respostas lógicas, de acordo com o programa que fizemos, é

$$\{N \leftarrow ana\}$$

2. Árvore de prova

Uma árvore de prova é uma entidade perfeitamente definida que constitui, por si só, uma prova, intuitivamente evidente, que um átomo é consequência lógica de um programa.

Vejamos, primeiro, um exemplo de uma árvore de prova para o problema 1:



Estando feito o programa `P` (o "nosso primeiro programa em Prolog"), uma leitura de baixo para cima desta árvore conduz à convicção que o átomo na raiz, `avo(rosa,ana)`, é consequência lógica de `P`, visto que esta leitura (de baixo para cima) segue o sentido dos símbolos `:-` nas cláusulas em `P` e estes símbolos são interpretados como implicações.

Não vamos dar uma definição formal da noção de *arvore de prova@árvore de prova* segundo `P`, mas apenas reter que é uma árvore (finita), orientada, cujos *nós* são constituídos por átomos, cujas folhas (nós terminais) são constituídas pelos átomos fictícios `verdade` e cujos nós não terminais verificam a seguinte propriedade: a cada nó não terminal corresponde uma instância de uma cláusula do programa `P`, a cabeça da instância da cláusula coincidindo com o átomo contido nesse nó, os átomos do corpo dessa instância de cláusula sendo os átomos que constituem os filhos do nó. No caso em que o corpo é vazio, colocamos o átomo fictício `verdade` como único filho.

Observe-se que toda a sub-árvore de uma árvore de prova é também uma árvore de prova. Por outras palavras, todo o átomo contido num nó de uma árvore de prova é átomo raiz de uma árvore de prova.

Por outro lado, uma árvore de prova pode comportar variáveis e então toda a instância de uma árvore de prova é ainda uma árvore de prova.

Com as noções introduzidas é possível demonstrar o seguinte teorema.

Teorema: Um átomo é consequência lógica de `P` se, e somente se, é raiz de uma árvore de prova segundo `P`.

Assim sendo, a notação $CL(P)$ é exactamente o conjunto dos átomos que são raízes de uma árvore de prova segundo `P`.

Este resultado pode ser visto como um resultado de equivalência entre um significado lógico e um significado construtivo baseado na noção de prova.

Resta agora ver como produzir, de uma maneira efectiva, as árvores de prova, isto é, uma maneira de obter as respostas lógicas a uma questão.

3. Unificação

O átomo contido num nó não terminal de uma árvore de prova, é uma instância de uma cabeça de cláusula. Excluindo a raiz, esse átomo é também uma instância de um átomo do corpo de uma cláusula. A construção de uma árvore de prova, em grosso modo, reduz-se à procura de instâncias comuns a dois átomos. É este facto que motiva as considerações que se seguem.

Para explicarmos o conceito de unificação, na sua forma mais geral, e também para estarmos de acordo com a funcionalidade do Prolog, vamos, antes de mais, generalizar o conceito de átomo. Tínhamos dito que um átomo escreve-se com um símbolo de predicado e um número de termos igual à aridade do símbolo do predicado em questão e que estes termos constituíam os chamados argumentos do átomo. Vamos agora permitir (definição recursiva!) que os argumentos possam ser, eles mesmo, átomos.

Dois átomos $atomo_1$ e $atomo_2$ são *unificáveis*, se existir uma substituição `s` tal que $s(atomo_1) = s(atomo_2)$. Nesse caso dizemos que s é um *unificador* de $atomo_1$ e $atomo_2$.

Por exemplo os átomos $atomo_1 = p(X, b)$ e $atomo_2 = p(a, Y)$ são unificáveis pelo unificador $s = \{X \leftarrow a, Y \leftarrow b\}$.

Uma constante é apenas unificável com ela própria e com uma variável.

Intuitivamente, um unificador do $atomo_1$ e do $atomo_2$, é uma substituição que consegue "instanciar" suficientemente as variáveis de modo a satisfazer a equação $atomo_1 = atomo_2$.

Um *unificador minimal*, é um unificador que instancia o menos possível as variáveis. Por exemplo os dois átomos $atomo_1 = p(X, Y)$ e $atomo_2 = p(Y, X)$ são unificáveis por $s_1 = \{Y \leftarrow X\}$, por $s_2 = \{X \leftarrow Y\}$, por $s_3 = \{X \leftarrow a,\ Y \leftarrow a\}$, ... mas s_1 e s_2 são os unificadores minimais. Formalmente, o unificador s é minimal se, para todo o unificador s', existe uma substituição s'' tal que $s' = s'' \circ s$ (composição de substituições). Pode-se mostrar que se existir um unificador entre dois átomos, então existe um unificador minimal.

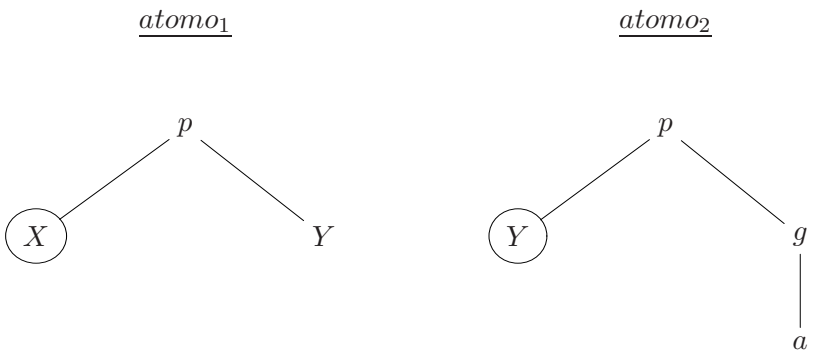
Também é possível demonstrar que se `X` é uma variável e `atomo` um átomo, então `atomo` e `X` são unificáveis se, e somente se, `atomo` não contiver a variável `X` (quando a variável `X` não tem *ocorrência* em `atomo`).

Existe um algoritmo (algoritmo de Robinson) que, dados dois átomos, calcula um unificador minimal, se os os dois átomos são unificáveis, ou então termina com a indicação da impossibilidade de unificação.

Seguem-se dois exemplos simples que ilustram o método. Ajuda representar os átomos na forma de árvore. A ideia é tentar construir o unificador por aproximações sucessivas, por composição de substituições. Em cada etapa procuramos uma variável que é uma folha numa das árvores e tentamos unificá-la com a sub-árvore correspondente na outra árvore. Nos esquemas que se seguem, essas variáveis e essas sub-árvores são envoltas com uma curva fechada.

Primeiro Exemplo: $atomo_1 = p(X, Y)$, $atomo_2 = p\,(Y, g(a))$

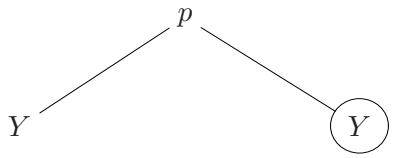
Etapa nº1



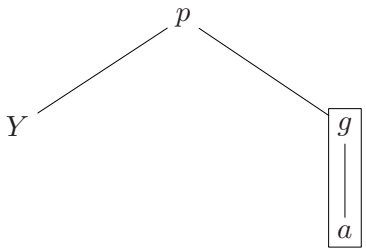
$s_1 = \{X \leftarrow Y\}$

Etapa nº2

$\underline{s_1\,(atomo_1)}$



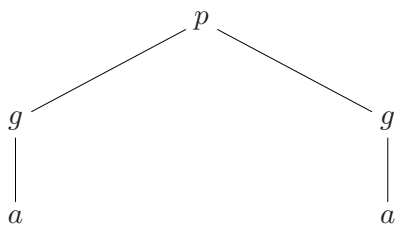
$\underline{s_1\,(atomo_2)}$



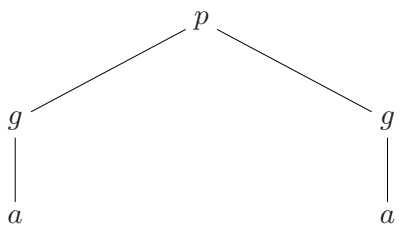
$s_2 = \{Y \leftarrow g(a)\}$

Etapa nº3

$\underline{s_2\,(s_1(atomo_1))}$



$\underline{s_2\,(s_1(atomo_2))}$



Conclusão: unificador minimal dado por

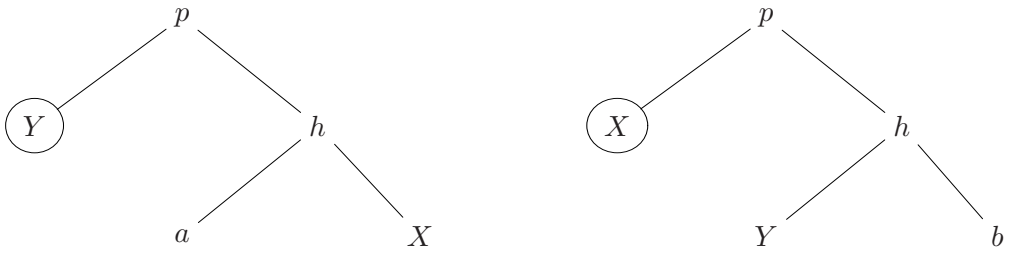
$$s_2 \circ s_1 = \{X \leftarrow g(a),\, Y \leftarrow g(a)\}$$

Segundo Exemplo:

$$atomo_1 = p\,(Y, h(a, X))\,, \quad atomo_2 = p\,(X, h(Y, b))$$

Etapa nº1

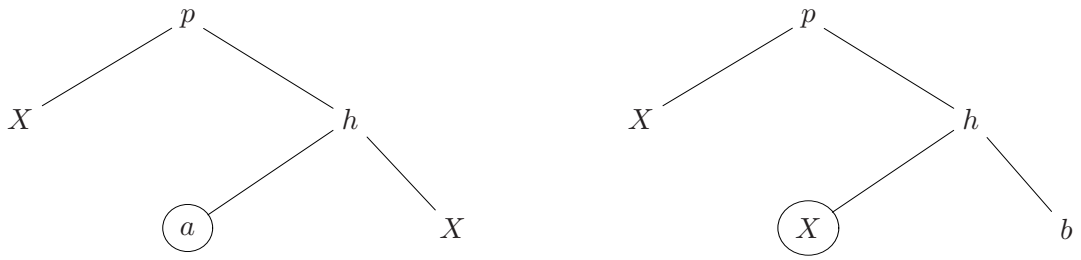
$\underline{atomo_1}$ $\underline{atomo_2}$



$s_1 = \{Y \leftarrow X\}$

Etapa nº2

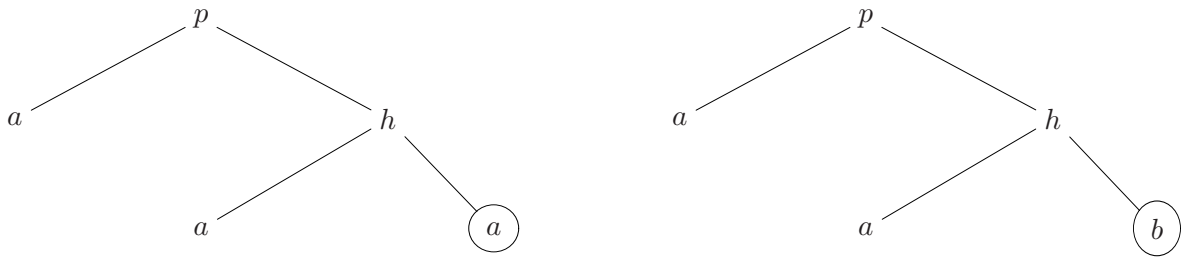
$\underline{s_1\,(atomo_1)}$ $\underline{s_1\,(atomo_2)}$



$s_2 = \{X \leftarrow a\}$

Etapa nº3

$\underline{s_2\,(s_1(atomo_1))}$ $\underline{s_2\,(s_1(atomo_2))}$



Conclusão: os átomos não são unificáveis (a constante a não unifica com a constante b).

Dois átomos $atomo_1$ e $atomo_2$ são duas *variantes*, se podermos passar de um ao outro por uma simples mudança do nome das variáveis (duas variáveis distintas mudadas sempre por duas variáveis distintas). Por outro lado, $atomo_2$ é uma *renomeação* do $atomo_1$, em relação a um conjunto V_1 de variáveis, se $atomo_2$ é uma variante do $atomo_1$ e se $atomo_2$ não contiver nenhuma variável em V_1.
As noções de unificação e de renomeação, definidas para átomos, estendem-se, da maneira natural, para as cláusulas.

4. Construção de uma árvore de prova

Obtemos a noção de *arvore de prova parcial@árvore de prova parcial* a partir da noção de árvore de prova, modificando a condição sobre as folhas. Numa árvore de prova parcial, uma

folha pode conter *verdade* ou um átomo. Uma árvore de prova (total) é uma árvore de prova parcial em que todas as folhas contêm *verdade*.

Existe um algoritmo que, dado um átomo `A`, constrói, eventualmente, uma árvore de prova com um átomo na raiz que é uma instância de `A`. Esse algoritmo é *"não determinístico"*, na medida que contém *escolhas* a serem feitas.

Vejamos um exemplo que mostra o método. A ideia é considerar o átomo inicial `A` como constituindo ele mesmo uma árvore de prova parcial e ir depois "desenvolvendo" essa árvore de prova parcial pelas suas folhas e instanciando-a.

O exemplo consiste em considerar o programa `P` que fizemos no âmbito do problema n^o1 e partir do átomo inicial `avo(rosa,N)` correspondente à questão

```
?- avo(rosa,N).
```

Etapa nº1 A árvore de prova parcial reduz-se a uma única folha

$$avo(rosa, N)$$

Vamos *escolher* uma cláusula em `P` para "pendurar" filhos a esta folha, mas as variáveis dessa cláusula não deverão ter relação com as da árvore de prova parcial, o que nos leva a substituir a cláusula por uma sua renomeação em relação às variáveis da árvore de prova parcial. De um ponto de vista lógico esta renomeação não tem qualquer reflexo, uma vez que as variáveis de uma cláusula são quantificadas universalmente.

Escolhemos uma cláusula (renomeada) cuja cabeça possa ser unificada com a folha

```
avo(rosa,N)
```

Neste caso existe apenas uma possibilidade:

```
avo(A1,N1) :- progenitor(A1,B1), progenitor(B1,N1).
```

O algoritmo de unificação dá-nos o unificador minimal

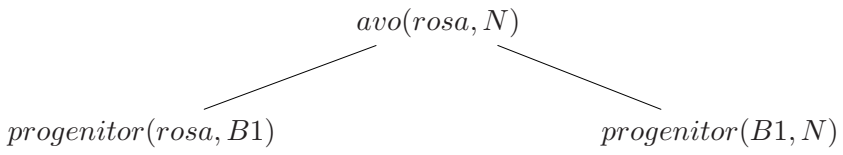
$$s_1 = \{\texttt{A1} \leftarrow \texttt{rosa}, \texttt{N1} \leftarrow \texttt{N}\}$$

Notar que podíamos também tomar o outro unificador minimal:

$$\{\texttt{A1} \leftarrow \texttt{rosa}, \texttt{N} \leftarrow \texttt{N1}\}$$

Instanciamos a árvore de prova parcial por s_1 assim como a cláusula (renomeada). Por construção a folha da árvore de prova parcial coincide agora com a cabeça da cláusula. Acrescentamos os átomos do corpo da cláusula como filhos da folha, obtendo então uma nova árvore de prova parcial:

Etapa nº2



Vamos *escolher* uma das folhas desta árvore de prova parcial. Consideremos, por exemplo, a folha `progenitor(rosa,B1)`. De seguida, *escolhemos* uma cláusula (renomeada) de `P` cuja cabeça possa ser unificada com a folha escolhida. Consideremos a cláusula

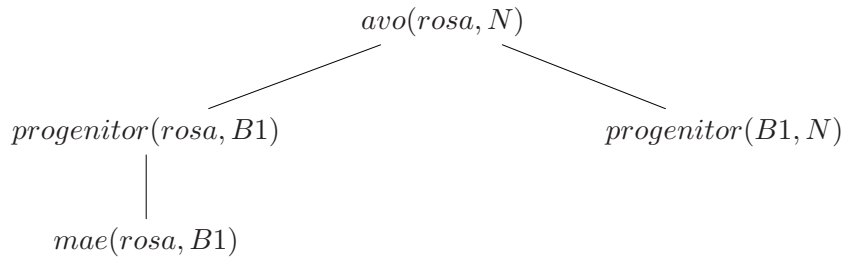
```
progenitor(P2,F2) :- mae(P2,F2).
```

O algoritmo de unificação dá-nos

$$s_2 = \{\texttt{P2} \leftarrow \texttt{rosa}, \texttt{F2} \leftarrow \texttt{B1}\}.$$

Instanciamos por s_2 a árvore de prova parcial e a cláusula, acrescentando os átomos do corpo da cláusula como filhos da folha escolhida. Obtemos:

Etapa nº3

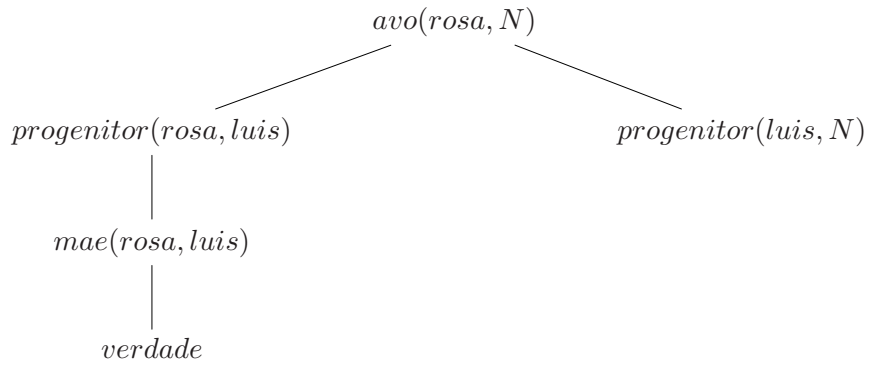


Escolhemos uma folha. Seja ela `mae(rosa,B1)`. A única cláusula cuja cabeça pode ser unificada com a folha considerada é o facto

```
mae(rosa,luis).
```

O unificador é $s_3 = \{\texttt{B1} \leftarrow \texttt{luis}\}$. Instanciamos ao mesmo tempo por s_3 a árvore e o facto. Neste caso o corpo da cláusula é vazio (facto) pelo que acrescentamos como filho o átomo fictício *verdade*, obtendo

Etapa nº4



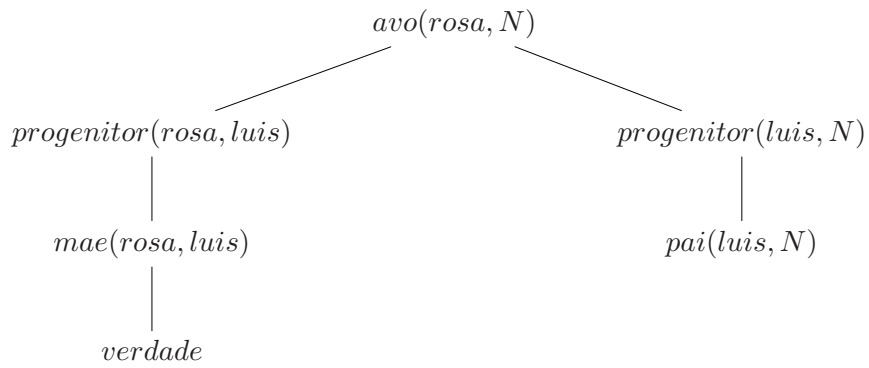
Apenas nos resta a folha `progenitor(luis,N)`. *Escolhamos* uma cláusula (renomeada). Seja ela

```
progenitor(P4,F4) :- pai(P4,F4).
```

Um unificador minimal é $s_4 = \{\texttt{P4} \leftarrow \texttt{luis}, \texttt{F4} \leftarrow \texttt{N}\}$.

Instanciando a árvore e cláusula por s_4, e acrescentando o átomo do corpo da cláusula como filho, obtemos:

Etapa nº5



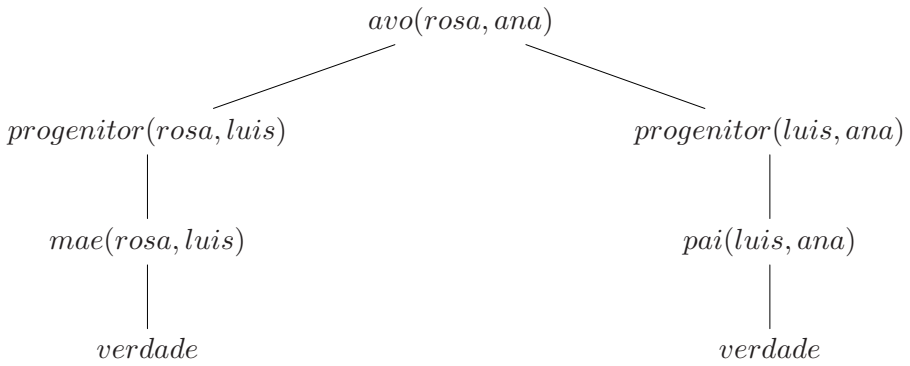
Escolhemos uma cláusula (renomeada) possível de instanciar com a folha `pai(luis,N)`. Consideremos, por exemplo, o facto

```
pai(luis,ana).
```

O unificador é $s_5 = \{N \leftarrow ana\}$.

Instanciando por s_5 e atendendo que a cláusula escolhida é um facto, acrescentamos como filho o átomo fictício *verdade*, obtendo finalmente a árvore de prova apresentada na secção 2:

Etapa nº6



O processo de construção de uma árvore de prova é, então, um processo por aproximações sucessivas. Se acharmos a composição dos unificadores usados em cada etapa, obtemos uma substituição s. No exemplo precedente, $s = s_5 \circ s_4 \circ s_3 \circ s_2 \circ s_1$. Esta substituição verifica a seguinte propriedade: o átomo da raiz da árvore de prova é uma instância $s(\mathtt{A})$ do átomo inicial `A`.

A s chamamos *resposta calculada* à questão

```
?- A.
```

Resulta que toda a resposta calculada é uma resposta lógica. O que falta examinar é a completude do algoritmo de construção de árvores de prova, isto é, se poderemos obter deste modo todas as respostas lógicas.

O algoritmo que vimos é não determinístico. No exemplo considerado fomos escolhendo quer as folhas quer as cláusulas. Se tivéssemos escolhido as folhas de outra maneira, mas conservando as mesmas escolhas de cláusulas, teríamos obtido a mesma árvore de prova.

Por outro lado, a escolha diferente das cláusulas ter-nos-ia levado a uma outra árvore de prova e a uma outra resposta calculada: $\{\mathtt{N} \leftarrow \mathtt{eusebio}\}$; ou então a um impasse, quer dizer, a uma impossibilidade de continuar a construção, por falta de uma cláusula cuja cabeça fosse passível de ser unificada com a folha escolhida.

Como já foi referido, existem duas fontes de não determinismo:

- não determinismo do tipo f: escolha da folha;
- não determinismo do tipo c: escolha da cláusula.

É possível formular rigorosamente, e demonstrar, um resultado de completude que diz que, para obter todas as respostas lógicas, é suficiente *fixar*, arbitrariamente, as escolhas do tipo f e tentar *todas* as escolhas do tipo c.

Observe-se, por último, que foi apenas para simplificar que representámos uma questão por um átomo. Chamamos *alvo* a uma sucessão finita de átomos. Uma questão pode ser representada por um alvo. Um exemplo de uma questão para o nosso problema 1 pode ser:

```
?- avo(A,ana), progenitor(A,aldina).
```

Esta questão significa: "Quem são os avós de ana que são também progenitores de aldina?".

Poderemos sempre considerar uma questão reduzida a um só átomo. Para isso basta acrescentar uma cláusula adequada ao programa. Para o exemplo acima, podemos acrescentar a cláusula

```
questao(A) :- avo(A,ana), progenitor(A,aldina).
```

e tomar como nova questão o átomo `questao(A)`:

```
?- questao(A).
```

5. Exercícios

Problema 9. Para cada um dos seguintes pares de átomos, determine, se possível, o unificador minimal.

a): $p(a, X)$ e $p(Y, b)$;
b): $p(X, X)$ e $p(Y, Z)$;
c): $p(X, Y)$ e $p(Y, Z)$;
d): $p\,(t(X, t(X, b)))$ e $p\,(t(a, Z))$;
e): $p\,(t(X, t(X, b)))$ e $p\,(t(a, t(Z, Z)))$;
f): $p(X, f(Y))$ e $p(f(Y), X)$;
g): $p(X, f(X))$ e $p(f(Z), f(Z))$;
h): $a(A, B, d(a, n))$ e $a(d(U, X), Y, d(U, Z))$;
i): $q(V, X, X, V)$ e $q(Z, Z, Y, c)$;
j): $a(d(a, d(X, nil)), d(Y, nil), Z)$ e $a(d(U, X), Y, d(U, Z))$.

Problema 10. Considere a seguinte história:

> Toda a pessoa que passa no exame de IPL e ganha no totoloto é feliz. Todo aquele que estuda ou tem sorte passa em todos os exames. O João não estuda mas tem sorte. Toda a pessoa que tem sorte ganha no totoloto.

a): Crie um programa em lógica (em Prolog) que descreva esta história.
b): Prove, construindo uma árvore de prova, que o João é feliz.

Problema 11. Considere a seguinte história:

> Todas as pessoas que não são pobres e são espertas são felizes. As pessoas que sabem ler não são estúpidas. O João sabe ler e não é pobre. As pessoas felizes têm vidas excitantes.

a): Crie um programa em lógica (em Prolog) que descreva esta história. (Note que pode ter necessidade de acrescentar informação adicional, de senso comum, para que a prova da alínea seguinte se possa efectivar.)
b): Prove, construindo uma árvore de prova, que existe uma pessoa com uma vida excitante.

Problema 12. Considere as seguintes afirmações:

- Todas as companhias que valem menos do que outra companhia ou que são inconstantes são um mau investimento;
- A companhia Investimentos Inteligentes Lda. ou é um mau investimento ou existe uma outra companhia que cresce mais lentamente;
- A companhia Investimentos Cretinos Lda. cresce mais lentamente do que qualquer outra companhia.

a): Crie um programa em lógica (em Prolog) que represente o conhecimento expresso nestas afirmações.
b): Prove, construindo uma árvore de prova, que existe uma companhia que é um mau investimento.

Problema 13. Considere o seguinte conjunto de afirmações:

1. Os Ferrari são carros italianos.
2. Os Alfa Romeu são carros italianos.
3. Os carros italianos têm bom arranque.
4. Os Volkswagen são carros alemães.
5. Os carros parados durante muito tempo apresentam problemas de arranque.
6. Os carros italianos que ficam fora da garagem apresentam problemas de arranque com o tempo frio.

7. O carro do João é um Alfa Romeu.
8. O carro do João fica fora da garagem.
9. O tempo está frio.

Represente o conhecimento acima expresso, da forma que lhe parecer mais conveniente, e construa uma árvore de prova que mostre que o carro do João tem problemas de arranque.

Problema 14. Considere o seguinte conjunto de afirmações:

1. Os guardas fiscais inspeccionam todas as pessoas que entram neste país e que não sejam VIPs.
2. Alguns traficantes de droga entraram neste país e foram apenas revistados por traficantes de droga.
3. Nenhum traficante de droga é VIP.

Represente o conhecimento acima expresso, da forma que lhe parecer mais conveniente, e construa uma árvore de prova que mostre que *alguns guardas fiscais são traficantes de droga.*

Problema 15. Imagine a seguinte situação: existem dois blocos, A e B, e uma mesa. Os blocos encontram-se em cima da mesa, com B sobre A.

a): Exprima, em notação lógica (do Prolog) as seguintes regras de conhecimento:
   1. Se um objecto está sobre outro então está acima dele.
   2. O predicado *acima de* é transitivo.

b): Usando uma árvore de prova, e dada a situação descrita, prove que B está acima da mesa.

Problema 16.

a): Crie um programa em lógica (em Prolog) que exprima as seguintes afirmações:
   - Israel retaliará se for atacada por um país árabe.
   - Israel não ataca países seus aliados.
   - Dois países são aliados se têm um inimigo comum.
   - A Arábia Saudita, os EUA a Inglaterra e Israel são inimigos do Iraque.
   - A Síria, a Arábia Saudita e o Iraque são países árabes.
   - Nem todos os países árabes são aliados.
   - Qualquer país ou é aliado ou inimigo de Israel.

b): Prove por meio de uma árvore de prova que, admitindo que o Iraque atacou Israel, Israel retaliará.

Problema 17. Considere os seguintes factos:

- O João, a Susana, o Basílio e a Elvira são sócios da Associação Académica.
- O João é casado com a Susana.
- A Elvira é irmã do Basílio.
- Todas as esposas de um sócio da Associação Académica são sócias da Associação Académica.
- A última reunião de sócios da Associação Académica decorreu em casa do João.

a): Represente logicamente os factos acima descritos.

b): Prove, por meio de uma árvore de prova, que a última reunião de sócios da Associação Académica decorreu em casa da Susana. Caso necessite acrescente a informação adicional necessária e suficiente para que a prova se possa efectivar.

3
Significado operacional determinístico

1. Estratégia, árvore de procura

Por estratégia entendemos um método determinístico de construir árvores de prova, com o intuito de obter as respostas calculadas.

Do que vimos, uma estratégia pode ser definida fixando:

- por um lado as escolhas do tipo `f`;
- por outro lado um método que permita tentar todas as escolhas do tipo `c`.

Designamos por *escolha standard do tipo* `f`, a escolha da folha mais à esquerda na árvore de prova parcial. Foi a escolha adoptada no exemplo em 2.4.

Estando fixa a escolha do tipo `f`, podemos representar todas as tentativas de escolha do tipo `c`, por uma *arvore de procura@árvore de procura*.

A *arvore de procura standard@árvore de procura standard*, é aquela que corresponde à escolha standard do tipo `f`.

O princípio é o seguinte: partimos de uma questão

```
?- A.
```

Cada nó da árvore de procura contém um *alvo*, isto é, uma sucessão finita de átomos. O alvo da raiz é o átomo `A`.

Em cada nó da árvore de procura, um dos átomos do alvo é designado por *atomo escolhido@átomo escolhido*. Na árvore de procura standard, o átomo escolhido é sempre o *primeiro* átomo do alvo.

Um nó da árvore de procura tem tantos filhos quantas as cláusulas do programa (renomeadas) cuja cabeça possa ser unificada com o átomo escolhido desse nó.

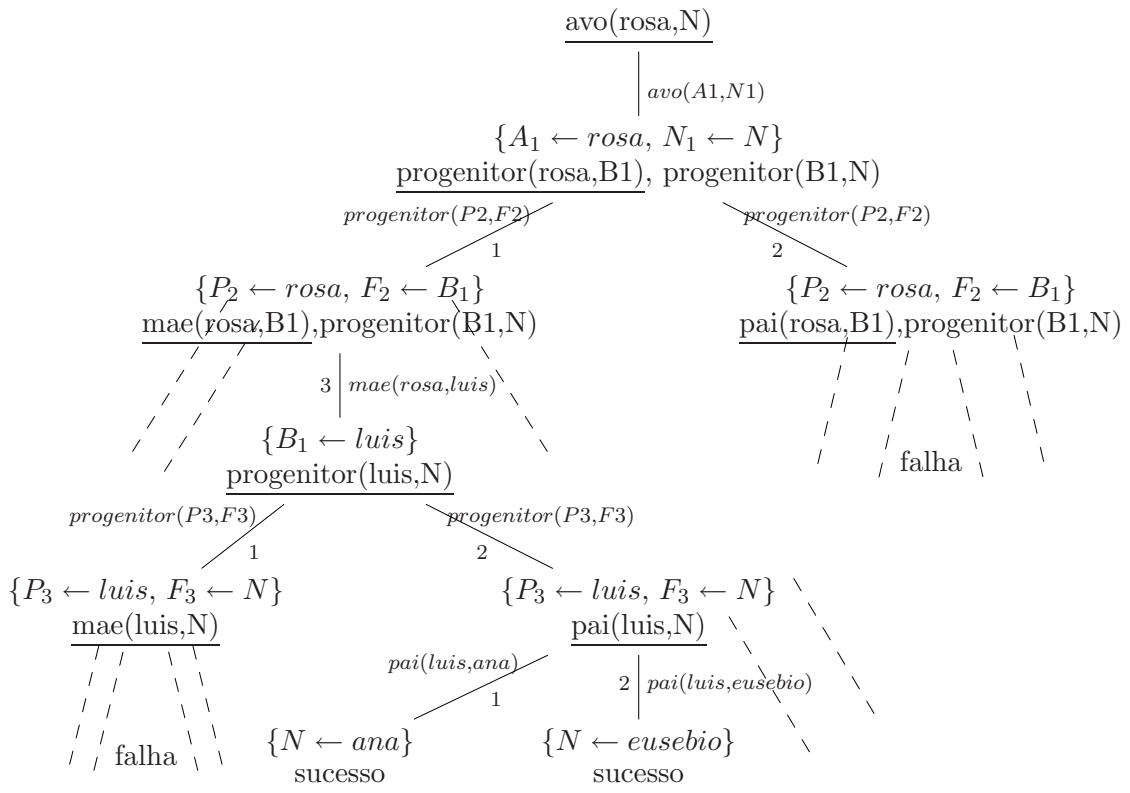
O alvo de um filho é obtido a partir do alvo do pai, unificando o átomo escolhido com uma cabeça de cláusula (renomeada); instanciando por esse unificador o alvo do pai e a cláusula; e substituindo o átomo escolhido pelo corpo da cláusula.

A árvore de procura leva também indicações suplementares. Um exemplo são as substituições utilizadas.

Vejamos um exemplo que mostra o método de construção. Retomemos a questão

```
?- avo(rosa,N).
```

Árvore de procura:



Para facilitar a construção da árvore de procura, acrescentámos a cada arco pai–filho a cabeça da cláusula (renomeada) correspondente, bem como o número desta cláusula no programa. Por outro lado, cada nó leva a substituição que define o unificador minimal entre o átomo escolhido do nó pai e a cabeça da cláusula indicada sobre o arco pai–filho (podemos considerar que o nó raiz leva a substituição identidade { }).

É por conseguinte fácil, inscrevendo sistematicamente estas indicações, construir a árvore de procura descendo da raiz e substituindo o átomo escolhido (que sublinhamos) pelo corpo da cláusula, depois de instanciada. Na prática, para estarmos seguros de que tentamos todas as cláusulas possíveis de serem unificadas, tentamos todas as cláusulas com o mesmo símbolo de predicado do átomo escolhido, colocando a pontilhado as que não são unificáveis.

Graças às cláusulas cujo corpo é vazio, factos, podemos acabar em nós nos quais o alvo é vazio. Esses nós são os nós terminais (folhas) da árvore de procura, e neles colocamos a menção `sucesso`. Mas existem outros nós terminais: aqueles para os quais nenhuma cláusula (renomeada) tem uma cabeça que pode ser unificada com o átomo escolhido e portanto não possuem filhos. Juntamos a esses nós a menção `falha`.

É importante notar que os filhos de um nó são ordenados (esquerda–direita) pela ordem das cláusulas no programa.

Uma árvore de procura pode ser infinita, quer dizer, poderá ter ramos infinitos que não são, portanto, terminados por uma folha `falha` ou `sucesso`. Veremos exemplos à frente.

A árvore de procura do exemplo anterior é a *árvore de procura standard* (oriunda do átomo `avo(rosa,N)`), já que o átomo escolhido em cada nó (o átomo sublinhado) é sempre o primeiro.

Vemos facilmente que cada ramo da árvore de procura, retrata a história das tentativas de construção de uma árvore de prova. Por exemplo, o ramo que acaba na folha `sucesso` com $\{N \leftarrow ana\}$, retrata exactamente a construção da árvore de prova do capítulo anterior: 6 etapas de construção que correspondem a 6 nós no ramo. Os ramos que acabam em folhas `falha` correspondem às tentativas infrutíferas (situações de impasse); os ramos que acabam em folhas `sucesso` correspondem às tentativas de êxito na construção de uma árvore de prova total. Cada nó da árvore de procura corresponde a uma etapa na construção de uma árvore de prova. Nessa etapa já construímos uma árvore de prova parcial e o alvo contido no nó não

é mais do que a sucessão de folhas dessa árvore de prova parcial. A escolha do átomo num nó da árvore de procura, corresponde à escolha da folha na árvore de prova parcial.

A cada ramo `sucesso` da árvore de procura, corresponde uma resposta calculada à questão de onde é oriunda a árvore de procura: é a resposta calculada que decorre da construção da árvore de prova associada a esse ramo `sucesso` e que pode ser obtida pela composição das substituições indicadas em cada nó do ramo. Quando a resposta calculada não é evidente, mencionamo-la também ao lado da menção `sucesso`. É o que fazemos no exemplo seguinte.

Mas antes vamos introduzir o conceito (recursivo) de lista.

2. Listas

Uma *lista* ou é vazia, representada por `[ ]`, ou então é constituída por um elemento, designado por *cabeça da lista*, e uma lista (possivelmente vazia) com os restantes elementos. Nesta última situação, a lista é representada por

```
[X|R]
```

onde `X` representa a cabeça da lista e `R` a lista dos restantes elementos.

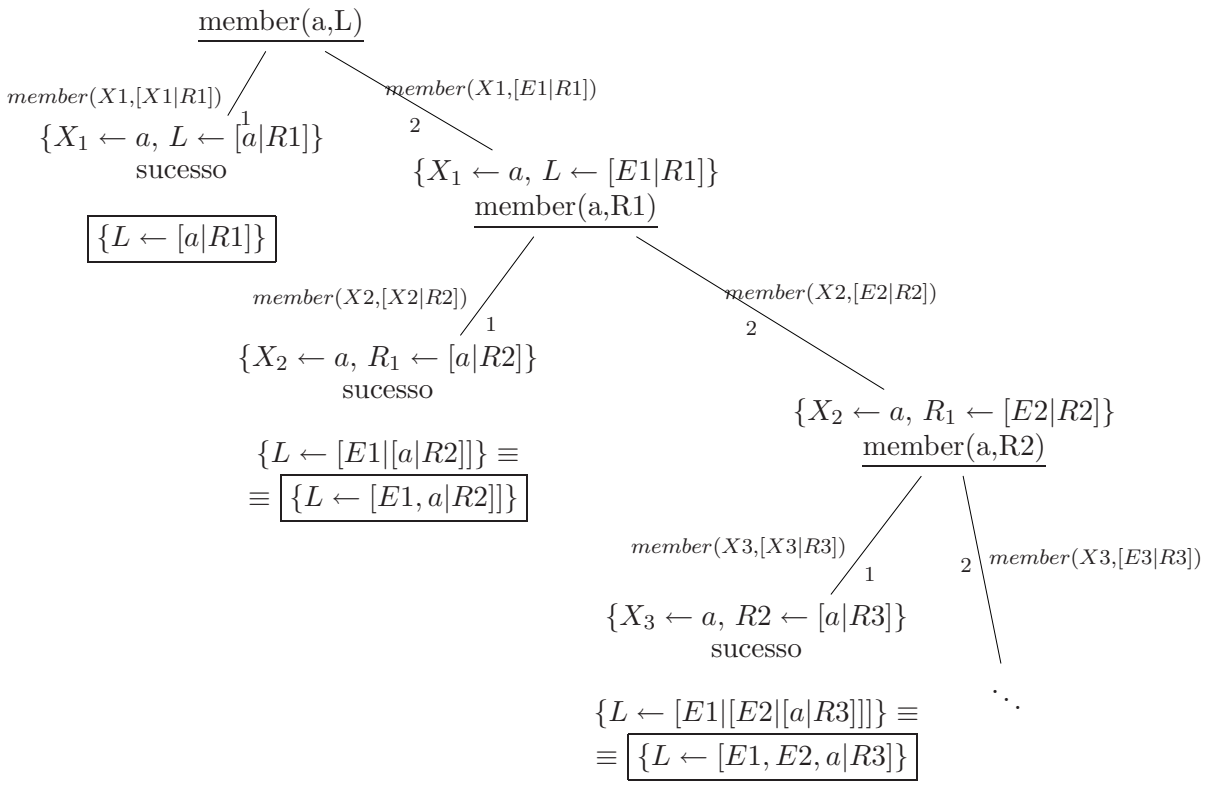
Na posse da definição de lista, vamos implementar o predicado `member/2` (que se encontra disponível na maior parte das implementações de Prolog) e que resulta verdadeiro se, e somente se, o objecto no primeiro argumento pertencer à lista no segundo argumento:

```
member(X,[X|R]).                % a cabeca da lista pertence ‘a lista
member(X,[Y|R]) :- member(X,R). % se nao ’e cabeca, deve pertencer ‘a
                                % lista dos restantes elementos
```

Vamos colocar a questão

```
?- member(a,L).
```

que significa: “Quais são as listas `L` que contêm a constante `a`?”; e construir a árvore de procura que ela origina:



Vemos que esta árvore de procura tem um *ramo infinito* e contém também um número infinito de respostas calculadas.

Reparar que as respostas lógicas à questão não são todas directamente calculadas. No entanto as respostas calculadas são suficientes para conhecer todas as respostas lógicas. Por exemplo a resposta lógica $\{L \leftarrow [a,\ a]\}$ é obtida a partir da resposta calculada $\{L \leftarrow [E1, a|R2]\}$

(segundo sucesso a partir da esquerda), instanciando `E1` com `a` e `R2` com a lista vazia; ou também a partir da resposta calculada $\{L \leftarrow [a|R1]\}$ instanciando `R1` com `[a]`.

De um modo mais geral, o resultado de completude enunciado no capítulo anterior pode ser formulado em termos da árvore de procura: para achar *todas* as respostas à questão

```
?- A.
```

é suficiente construir *completamente* uma (só) árvore de procura (qualquer uma) originada por `A`. Basta então construir *completamente* a árvore de procura standard.

Em lugar de partirmos de uma questão com apenas um átomo `A`, a árvore de procura pode ser originada, de um modo mais geral, por um alvo (ver comentário no final da secção 2.4).

3. Estratégia standard

Fixamos a primeira componente de uma estratégia, isto é, as escolhas do tipo `f`, adoptando a escolha standard: consideramos sempre a árvore de procura standard.

A segunda componente de uma estratégia, isto é, o método de tentar todas as escolhas do tipo `c`, pode agora ser descrito como um *percurso* da árvore de procura.

Designamos por *percurso standard*, o percurso "em profundidade primeiro e da esquerda para a direita" da árvore de procura.

Chamamos *estratégia standard* à estratégia definida pela árvore de procura standard e pelo percurso standard.

Na prática procede-se a um "percurso-construção" da árvore de procura. É um meio de compreender o cálculo efectuado por um interpretador de um programa em lógica (interpretador Prolog) quando submetemos uma questão: ele efectua o percurso-construção da árvore de procura oriunda da questão e a cada construção de uma folha `sucesso`, ele faz sair como *resultado* a resposta calculada dessa folha. Na prática, o resultado é a restrição da resposta calculada às variáveis que figuram na questão. Se não houver nenhuma variável na questão, o resultado consiste apenas na indicação de `sucesso`.

É, em particular, um meio de compreender:

a): A ordem pela qual são produzidas as respostas (ordem induzida pela ordem do percurso-construção).

b): A razão da presença eventual de variáveis nas respostas produzidas por um interpretador Prolog.

c): A razão de um "bloqueamento" eventual, assim como a ausência de certas respostas (ver comentário à frente).

O Prolog pode ser visto como um interpretador que segue a estratégia standard. No entanto, além da parte "puramente lógica", o Prolog tem partes "impuras" ou "não lógicas", que veremos mais à frente. Embora estes elementos "não lógicos" não tenham uma interpretação na lógica clássica, podemos, mesmo assim, descrevê-los em termos da árvore de procura.

Certas primitivas da linguagem, que não têm senão um significado operacional, podem ser descritas em termos da modificação dinâmica da árvore de procura no decorrer do percurso-construção.

É importante observar que pode acontecer o percurso-construção apenas construir efectivamente uma *parte* da árvore de procura. Em particular com a estratégia standard o percurso-construção pode perder-se num ramo infinito e portanto jamais percorrer certos ramos ulteriores que ficarão por construir.

É então possível, se a árvore de procura tiver ramos infinitos, que certas respostas não sejam jamais calculadas, porque as folhas `sucesso` correspondentes não serão jamais atingidas (veja-se o exemplo que se segue). Por esta razão, dizemos que a estratégia standard é *incompleta*.

A escolha da estratégia standard (incompleta) por parte do Prolog, justifica-se por razões de ordem prática, pelos problemas de gestão de memória.

Vamos agora mostrar, com um exemplo, a influência da *ordem das cláusulas* num programa. Para isso basta relembrar o programa que fizemos com o predicado `member/2` e árvore de procura que construímos. Com a estratégia standard, obtemos uma infinidade de respostas, pela ordem:

```
L = [a|R1];
L = [E1,a|R2];
...
```

Se agora invertermos a ordem das duas cláusulas

```
member(X,[Y|R]) :- member(X,R).
member(X,[X|R]).
```

resulta claro que obtemos uma nova árvore de procura que é exactamente a simétrica da anterior. Em particular, agora o ramo infinito é o ramo mais à esquerda. Com a estratégia standard, a procura vai prosseguir indefinidamente sem nunca obtermos uma resposta.

4. Visão procedimental da programação em lógica

O cálculo efectuado por um interpretador de Prolog pode ser descrito em termos de avanços e retrocessos ("back-tracking") que são o reflexo do percurso standard.

O percurso standard da árvore de procura (standard) é uma sucessão de *avanços* (= descidas) e de *retrocessos* (= subidas) de um nó a outro, partindo da raiz. Um avanço deixa eventualmente a escolha de cláusulas em espera. Um retrocesso é provocado pela passagem a uma folha da árvore de procura (`falha` ou `sucesso`) e regressa ao último nó onde deixámos escolhas em espera, de maneira a tentar a primeira escolha que resta.

Cada avanço está associado a uma transformação (alvo e instanciação) e cada retrocesso está associado à anulação dessa transformação.

A *visão procedimental* da programação em lógica, consiste em ver o átomo `A` como uma *chamada de procedimento*, cujo efeito é construir uma árvore de prova a partir de `A`. Um alvo `B` é então visto como uma sucessão de chamadas de procedimentos.

Seja

```
A0 :- A1, ..., Am.
```

a cláusula que faz passar do nó corrente ao seu filho no ramo da árvore de procura, isto é, a cláusula que permite juntar os filhos à folha `A` da árvore de prova parcial corrente. Então o que vemos como a chamada do procedimento `A` é primeiro a unificação de `A` com `A0`, que produz um unificador minimal `s`, seguido da sucessão de chamadas dos procedimentos `s(A1), ..., s(Am)`.

Isto permite ver a unificação como um modo de *transmissão de parâmetros* e a cláusula `A0 :- A1, ..., Am.` como uma *definição de procedimento*. Os parâmetros são os argumentos dos átomos.

Um procedimento pode possuir várias definições, que são as cláusulas no programa. Por exemplo no "nosso primeiro programa em Prolog", as cláusulas `progenitor` contêm duas definições do "procedimento `progenitor`".

Falamos de *"procedimentos não determinísticos"*. Um procedimento pode ter uma definição recursiva como

```
member(X,[Y|R]) :- member(X,R).
```

Segundo esta visão procedimental, um programa em lógica é o análogo (não determinístico) de uma sucessão de definições de procedimentos; uma questão (= um alvo, = uma sucessão de chamadas de procedimentos) é o análogo ao *programa principal* clássico.

Nesta analogia, a palavra *variável* pode ser enganosa. A variável "em lógica" não conhece a afectação no sentido clássico (algorítmico): ela sofre *instanciações*. Intuitivamente, o objecto que ela representa é precisado pelas instanciações sucessivas que ela sofre, mas esse objecto não muda bruscamente de valor como numa atribuição clássica do tipo `X := X + 1`.

Podemos entretanto prosseguir a analogia no que concerne ao *domínio de validade* das variáveis de um procedimento. Cada variável é *local* a uma chamada de procedimento. Não existem variáveis globais. Do ponto de vista da implementação de um interpretador de Prolog, este facto corresponde à necessidade de renomear as cláusulas, embora isso não seja mais do que um reflexo do facto de que em cada cláusula as variáveis são quantificadas universalmente.

Para acabar a analogia com uma linguagem baseada na noção de procedimento, podemos dizer que a função de *teste* é assegurada pela unificação. Por exemplo

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R).
```

é o análogo da estrutura

Se o primeiro argumento = cabeça da lista *Então*

primeiro argumento pertence à lista

Senão

...

Mesmo para descrever o cálculo *determinístico* seguindo a estratégia standard, é por vezes cómodo utilizar esta *terminologia procedimental*. Dizemos que a chamada do procedimento `A` *tem êxito* se uma das tentativas de construção de uma árvore de prova a partir de `A` teve êxito. Dizemos que essa chamada *fracassou em tempo finito* se todas as tentativas fracassaram sem nenhuma "se perder" num ramo infinito.

5. Exercícios

5.1. Exercícios com árvores de procura.

Exercício do Exame Final de 1997: Problema 18. Observe com cuidado a seguinte Base de Factos

```
% exerceProfRisco(S,P)  :-  a pessoa S exerce a profissao P que
%                           'e de alto risco.
exerceProfRisco(ana,professor) .   exerceProfRisco(rui,professor) .
exerceProfRisco(to,metalurgico) .  exerceProfRisco(xico,bombeiro) .
exerceProfRisco(marta,bombeiro) .  exerceProfRisco(hugo,fuzileiro) .
% fumaMuito(P)  :- a pessoa P fuma muitos cigarros por dia.
fumaMuito(sofia) .                 fumaMuito(rui) .
fumaMuito(ze) .                    fumaMuito(mariana) .
fumaMuito(pedro) .
% dorme(S,H)  :- a pessoa S dorme em media H horas por noite.
dorme(tita,7) .                    dorme(pedro,5) .
dorme(joao,4) .                    dorme(ze,5) .
dorme(sofia,6) .                   dorme(mariana,8) .
% cardiaco(P) :- a pessoa P sofre de doenca do coracao.
cardiaco(nuno) .
cardiaco(necas) .
cardiaco(andre) .
```

e o seguinte programa Prolog

```
morreCedo(P) :-  exerceProfRisco(P,_).
morreCedo(P) :-  fumaMuito(P),  dormePouco(P).
morreCedo(P) :-  cardiaco(P).

dormePouco(P) :-  dorme(P,H),  H < 6.
```

Responda, então, às alíneas seguintes:

a): Usando a *Árvore de Prova*, diga qual a resposta de um Interpretador de Prolog (IP), à questão

```
morreCedo( pedro ).
```

b): Usando a *Árvore de Procura*, mostre todas as respostas que um Interpretador de Prolog (IP), daria à questão

```
dormePouco( QUEM ).
```

c): Usando a *Árvore de Procura*, mostre que um Interpretador de Prolog (IP), teria de fazer *backtracking* (automático) para responder afirmativamente, "`yes`", à questão

```
morreCedo( nuno ).
```

d): Como interpretaria o facto dum IP responder negativamente, "`no`", à questão

```
morreCedo( joana ).
```

Exercício do Exame de Recurso de 1997: Problema 19. Para fazer a Gestão de cada Hotel ligado a uma Central de Reservas, construiu-se a seguinte base de factos (BF):

```
% cargo(C)  :-  o nome C define um cargo profissional em hotelaria.
cargo(director).
cargo(recepcionista).
cargo(grumete).
cargo(governanta).
cargo(barman).
% exerceCargo(P,C,H)  :- a pessoa P exerce o cargo C no hotel H.
exerceCargo(pedro,grumete,ibis).
exerceCargo(delfim,director,penta).
exerceCargo(mario,barman,alfa).
exerceCargo(zezinha,governanta,afonso5).
exerceCargo(analia,recepcionista,ibis).
exerceCargo(luis,recepcionista,penta).
% manda(C,S)  :- a pessoa que exerce o cargo C 'e
                 o chefe directo da pessoa que exerce o cargo S.
manda(director,recepcionista).
manda(director,governanta).
manda(recepcionista,grumete).
manda(governanta,barman).
% dormeMuito(P)  :- a pessoa P dorme muitas horas.
dormeMuito(antonio).
dormeMuito(sandra).
% deita(P,H)  :- a pessoa P deita-se as H horas.
deita(antonio,0).
deita(ana,03).
deita(sandra,22).
% hospedada(P,H)  :- a pessoa P esta hospedada no hotel H.
hospedada(antonio,penta).
hospedada(ana,ibis).
hospedada(sandra,afonso5).
```

e o seguinte programa Prolog:

```
% chamaBarman(P) :-
%  a pessoa P chama por um barman.
chamaBarman(P) :-  dormeMuito(P).
chamaBarman(P) :-  deitaTarde(P).
chamaBarman(P) :-  mandadoPor(barman,P).

% mandadoPor(S,C) :-
%  a pessoa S recebe ordens (directas ou nao) de C.
mandadoPor(S,C) :-  manda(C,S), !.
mandadoPor(S,C) :-  manda(C1,S), mandadoPor(C1,C).

deitaTarde(P)   :-  deita(P,H), H>0, H<12.
```

Responda, então, às alíneas seguintes:

a): Usando a Árvore de Prova, diga qual a resposta de um Interpretador de Prolog (IP), à questão

```
mandadoPor(grumete,director).
```

b): Usando a Árvore de Procura, mostre qual seria a 1ª e as restantes respostas de um Interpretador de Prolog (IP), à questão

```
exerceCargo(Quem,OQue,ibis).
```

c): Usando a Árvore de Procura, diga se um Interpretador de Prolog (IP), teria de fazer retrocesso (*backtracking*) para responder afirmativamente à questão

```
chamaBarman(ana).
```

d): Usando o predicado `manda/2` acima, escreva um novo predicado

`manda1(Nome1,Nome2)`

que permita saber que a pessoa `Nome1` manda directamente na pessoa `Nome2`, atendendo ao cargo que exercem e ao local de trabalho (o hotel terá de ser o mesmo).

e): Escreva as cláusulas que deveriam ser acrescentadas à BC para se poder validar que todos os factos `exerceCargo/3` (acima exemplificados) estavam aplicados consistentemente, i.é, relacionavam uma *pessoa* com um *cargo* num determinado *hotel*.

f): Escreva um predicado `barmen(P,LN)` que construa a lista de nomes, `LN`, de todos os Barman que podem ser chamados pela pessoa `P` (essa pessoa terá de estar hospedada num hotel e estar em condições de chamar um barman).

Exercício do Exame de Recorrência de 1998: Problema 20. Considere o seguinte enunciado:

> Quando a Alice entrou na floresta do esquecimento, ela não se esqueceu de tudo, apenas de certos factos. Frequentemente, ela esquecia-se do seu nome, mas o que ela se esquecia com mais regularidade era do dia da semana. O leão e o unicórnio são visitantes frequentes da floresta. Estes dois, são criaturas muito estranhas. O leão mente às Segundas, Terças e Quartas e diz a verdade nos restantes dias da semana. O unicórnio, por seu lado, mente às Quintas, Sextas e Sábados, mas diz a verdade nos outros dias da semana.
>
> Um dia a Alice encontrou o leão e o unicórnio debaixo de uma árvore. Eles fizeram as seguintes declarações:
>
> **Leão::** ontem foi um dos dias em que eu menti.
>
> **Unicórnio:** ontem foi um dos dias em que eu menti.
>
> Destas afirmações, Alice, que era uma rapariga inteligente, foi capaz de deduzir o dia da semana. Qual era ele?

Pretende-se que:

a): implemente um programa em Prolog que seja capaz de responder à pergunta acima formulada.

b): indique o modo como se colocaria a questão ao interpretador para obter a solução por este produzida.

c): justifique, por meio de uma *Árvore de Procura*, como é que o Prolog chegaria à solução.

Possível resolução das alíneas a) e b) do problema 20.

```
% a)
      ontem(domingo,sabado).
      ontem(sabado,sexta).
      ontem(sexta,quinta).
      ontem(quinta,quarta).
      ontem(quarta,terca).
      ontem(terca,segunda).
      ontem(segunda,domingo).

      mente(leao,segunda).
      mente(leao,terca).
      mente(leao,quarta).
      mente(unicornio,quinta).
      mente(unicornio,sexta).
      mente(unicornio,sabado).

      nao(P) :- P, !, false.
      nao(_).


      hoje(Animal,Hoje) :-
        mente(Animal,Hoje), ontem(Hoje,Ontem),
        nao(mente(Animal,Ontem)).
```

```
        hoje(Animal,Hoje) :-
          mente(Animal,Ontem),ontem(Hoje,Ontem),
          nao(mente(Animal,Hoje)).

        hoje(Dia) :-
          hoje(leao,Dia),
          hoje(unicornio,Dia).

% b)

        ?- hoje(D).
           D = quinta
```

Exercício do Exame de Recurso de 1998: Problema 21. Em resposta ao conhecimento expresso nas frases

- Se um homem é incompetente, não é prudente;
- Se um homem é ignorante, não tem esperança;
- Se um homem é violento, é incompetente;
- Se um homem não é prudente, é ignorante:

resolveu-se considerar a seguinte BC:

```
homem(incompetente,nao_prudente).
homem(ignorante,sem_esperanca).
homem(violento,incompetente).
homem(nao_prudente,ignorante).

homem(X,Y) :- homem(X,Z),
              homem(Z,Y).
```

Supondo que forçamos sempre o backtracking (digitando `;`), cada vez que obtemos uma resposta do interpretador de `Prolog`, descreva, justificando por meio da árvore de procura, o que se obteria colocando a seguinte questão:

```
?- homem(nao_prudente,X).
```

5.2. Exercícios sobre listas.

Oito Exercícios resolvidos sobre listas: Problema 22.

a): Define dois predicados, *comprimento_par(Lista)* e *comprimento_impar(Lista)*, que são verdadeiros se o argumento *Lista* for respectivamente uma lista de comprimento par ou ímpar.

b): Define o predicado *capicua(Lista)*. Uma lista diz-se capicua se for indiferente ler a lista da esquerda para a direita ou da direita para a esquerda. Exemplos: $[a, n, a]$, $[m, a, d, a, m]$.

c): Define o predicado *shift(Lista1, Lista2)* de tal modo que Lista2 é a Lista1 "shiftada" rotacionalmente de um elemento para a esquerda. Por exemplo:

$$? - \text{shift}([1,2,3,4,5], \text{L1}), \text{shift}(\text{L1}, \text{L2}).$$
$$\text{L1} = [2,3,4,5,1]$$
$$\text{L2} = [3,4,5,1,2]$$

d): Define o predicado *traduz(Lista1, Lista2)* que traduz os números de 0 a 9 que possam aparecer na Lista1 pelas palavras correspondentes. Por exemplo:

$$? - \text{traduz}([3,5,a,1,3], \text{L}).$$
$$\text{L} = [\text{tres,cinco,a,um,tres}]$$

e): Define o predicado *subconjunto(Cnj, Subcnj)* onde *Cnj* e *Subcnj* são listas representando dois conjuntos. Pretendemos ser capazes de usar esta relação não só para verificar a relação subconjunto, mas também sermos capazes de gerar todos os possíveis subconjuntos de um dado conjunto. Por exemplo

```
?-subconjunto([a,b,c],S).

S = [a,b,c] ;

S = [b,c] ;

S = [c] ;

S = [ ] ;

...
```

f): Define o predicado *divide_lista(L, L1, L2)* de tal modo que os elementos de L sejam particionados entre *L1* e *L2*, e *L1* e *L2* tenham aproximadamente o mesmo comprimento. Por exemplo,

$$? - \text{divide\_lista}([a,b,c,d,e],[a,c,e],[b,d]).$$

yes

g): Define o predicado *nivela(Lista,ListaNivelada)* onde *Lista* pode ser uma lista de listas e *ListaNivelada* é a *Lista* "nivelada". Por exemplo:

$$? - \text{nivela}([a,b,[c,d],[],[[[e]]],f], L).$$

$$L = [a,b,c,d,e,f]$$

h): Define o predicado *members2(X,Y,L)*, que deve ser verdadeiro apenas quando X e Y são membros de L e X precede Y.

Possíveis soluções.

```
% a)

comprimento_par([]).
comprimento_impar([_]).

comprimento_par([_|L]) :- comprimento_impar(L).
comprimento_impar([_|L]) :- comprimento_par(L).


% b)

capicua([]).
capicua([_]).
capicua([X|L1]) :- inverte(L1,[X|L2]), capicua(L2).

% b)

capicua(L) :- inverte(L,L).

% c)

shift([],[]).
shift([X|L],S) :- concatena(L,[X],S).

% d)

significa(0,zero).
significa(1,um).
```

```
significa(2,dois).
significa(3,tres).
significa(4,quatro).
significa(5,cinco).
significa(6,seis).
significa(7,sete).
significa(8,oito).
significa(9,nove).
significa(X,X).

traduz([],[]).
traduz([X|L],[Y|T]) :- significa(X,Y), traduz(L,T).

% e)

del(X,[X|T],T).
del(X,[Y|T],[Y|R]) :- del(X,T,R).

subconjunto(L,L).
subconjunto(L,S) :- del(X,L,L1), subconjunto(L1,S).


% f)

comp([],0).
comp([X|L],N) :- comp(L,N1), N is N1 + 1.

permutacao([],[]).
permutacao(L,[X|P]) :- del(X,L,L1), permutacao(L1,P).

divide_lista(L,L1,L2) :-
  permutacao(L,P),
  concatena(L1,L2,P),
  comp(L,N),
  comp(L1,N1),
  comp(L2,N2),
  (N2 is N // 2 ; N1 is N // 2).

% g)

nivela([],[]).
nivela([[]|L],N) :-
  nivela(L,N), !.
nivela([[X|Y]|L],N) :-
  nivela([X|Y],N1),
  nivela(L,N2),
  concatena(N1,N2,N) ,!.
nivela([X|L],[X|N]) :-
  nivela(L,N).

% h)

members2(X,Y,[X|L]) :-
  member(Y,L).
members2(X,Y,[_|L]) :-
  members2(X,Y,L).
```

Exercício do Exame de Recorrência de 1997: Problema 23. Pretende-se a implementação em Prolog de um codificador de mensagens (e de um descodificador de mensagens encriptadas), pela substituição simples de uma letra (ou dígito), de a-z (ou 0-9), por outra, também de a-z (ou 0-9). Uma das formas mais simples de substituição é deslocar o alfabeto de uma quantidade específica (chamemos-lhe um *delta*).
Por exemplo, se cada letra e cada dígito for deslocada de 3, então

a b c d e f g h i j k l m n o p q r s t u v w x y z 0123456789

ficaria

d e f g h i j k l m n o p q r s t u v w x y z a b c 3456789012

e a mensagem '*encontra-me ao anoitecer*' ficaria '*hqfrqwud-ph dr dqrlwhfhu*'.

Usando o algoritmo acima descrito e o predicado pré-definido `name/2` (recorde-se que este predicado transforma uma frase (1º argumento) numa lista (2º argumento) com os códigos ASCII dos seus caracteres), implemente o predicado `code( F,FCode,Delta )` que recebe a frase `F` e a devolve codificada em `FCode` usando um deslocamento `Delta` para rodar as letras e os dígitos.
Exemplo:

```
?- code('encontra-me ao anoitecer',C,3).
C = 'hqfrqwud-ph dr dqrlwhfhu'
```

Exercício do Exame de Recorrência de 1997: Problema 24. Escreva em Prolog cláusulas adequadas (podendo recorrer, se necessário a predicados pré-definidos) para definir:

a): O predicado `profundidade( E,L,N )` que, dado um elemento `E` e uma lista `L` que pode conter elementos que são novamente listas, determina a profundidade `N` da primeira ocorrência desse elemento na lista (supõe-se que o nível da lista mais externa é 1). Exemplo:

```
? profundidade(7,[3,[5,7],[4,[7,0]]],N).
N = 2
```

b): O predicado `dobra( L,LL )`, sendo `L` e `LL` listas, em que `LL` contém todos os elementos de `L` duplicados. Por exemplo:

```
?- dobra([1,2,3],[1,1,2,2,3,3])
yes
```

c): O predicado `separa( L1,(Ln,Ll) )` que recebe como entrada uma lista `L1` formada por números e letras, contendo eventualmente repetições, e devolve como saída um par em que o primeiro elemento `Ln` é a lista apenas dos números e o segundo elemento `Ll` é a lista apenas das letras. No resultado não podem aparecer repetições, podendo recorrer a predicados de Prolog pré-definidos. Exemplo:

```
?- separa([b,d,2,a,5,b],R).
R = ([2,5],[d,a,b])
?- separa([4,d,5,4,q],R).
R = ([5,4],[d,q])
```

Exercício do Exame de Final de 1998: Problema 25. Nas aulas teórico-práticas foi sugerida a seguinte solução para o problema do macaco e das bananas:

```
accao(situacao(centro,cima_caixa,centro,nao_tem),
      apanhar,
      situacao(centro,cima_caixa,centro,tem)).
accao(situacao(P,chao,P,TN),
      subir,
      situacao(P,cima_caixa,P,TN)).
accao(situacao(P1,chao,P1,TN),
      empurrar(P1,P2),
      situacao(P2,chao,P2,TN)).
accao(situacao(P1,chao,C,TN),
      andar(P1,P2),
      situacao(P2,chao,C,TN)).
```

```
% cumpre_obj(S) <=> macaco pode obter banana partindo de S

cumpre_obj(situacao(_,_,_,tem)).
cumpre_obj(E) :-
      accao(E,A,S),
      cumpre_obj(S).
```

Depois de fazer o `'consult'` do ficheiro, bastava perguntar ao interpretador:

```
?- cumpre_obj(situacao(porta,chao,janela,nao_tem)).
yes
```

Com este exercício pretende-se que alterem o programa acima, de modo a sabermos qual a solução "que o macaco encontra" para apanhar a banana.

Possível resolução.

```
% Basta alterar o predicado cumpre_obj

cumpre_obj(situacao(_,_,_,tem),[]).
cumpre_obj(E,[Accao|R]) :-
  accao(E,Accao,S),
  cumpre_obj(S,R).

% ?- cumpre_obj(situacao(porta,chao,janela,nao_tem),L).
```

Exercício do Exame de Final de 1998: Problema 26. Com este exercício pretende-se que implementem um jogo da vida unidimensional com regras de evolução fixas, codificadas da seguinte maneira:

```
regra(0,0,0).
regra(0,1,1).
regra(1,0,1).
regra(1,1,0).
```

Passamos a explicar. Convencionamos que 0 representa uma célula morta enquanto 1 representa uma célula viva. Uma colónia de seres unicelulares será então representada por uma lista de zeros e uns. Por exemplo,

```
[0,1,0,1,1,0,1,0,1,0,0,1,1,0]
```

Com as regras de evolução, vamos poder obter as gerações que se seguem à colónia inicial. As regras apresentadas acima, significam que:

- se houver um 0 (na lista) com um 0 à sua direita, não se altera o 0 (a célula continua morta na próxima geração);
- se houver um 0 com um 1 à sua direita, então o 0 passa a 1;
- se houver um 1 com um 0 à sua direita, não se altera o 1;
- se houver um 1 com um 1 à sua direita, então o 1 passa a zero.

Pretendemos:

a): Que implementem um predicado `proxima_geracao/2` que recebe no primeiro argumento uma lista de 0's e 1's e retorne no segundo argumento, de acordo com as regras descritas, a colónia de células que a sucede.
Exemplos:

```
?- proxima_geracao([1,0,0,0,1,1],L).
   L = [1,0,0,1,0,1]

?- proxima_geracao([1,1,1,1,0],L).
   L = [0,0,0,1,0]
```

b): Que implementem um predicado `jogo_vida/3` que recebe no primeiro argumento o número `N` de gerações seguintes que pretendemos conhecer; no segundo argumento a colónia de células "actual"; e que retorna no último argumento a lista com as próximas `N` gerações (cada geração de células é uma lista de 0's e 1's). *Exemplo:*

```
?- jogo_vida(2,[1,0,1],L).
   L = [[1,1,1],[0,0,1]]
```

Possível resolução.

a):
```
proxima_geracao([],[]).
proxima_geracao([X],[X]).
proxima_geracao([X,Y|R],[Z|N]) :-
  regra(X,Y,Z),
  proxima_geracao([Y|R],N).
```

b):
```
jogo_vida(0,_,[]).
jogo_vida(N,L,[P|R]) :-
  proxima_geracao(L,P),
  N1 is N-1,
  jogo_vida(N1,P,R).
```

Exercício do Exame de Recorrência de 1998: Problema 27. Sobre operações com Listas em Prolog, responda às alíneas seguintes:

a): Nas aulas teórico-práticas, a quando da implementação do ficheiro *math.pl*, implementou-se o predicado `inverte/2` da maneira que se segue:

```
concatena([],L,L).
concatena([X|R],L,[X|C]) :-
  concatena(R,L,C).

inverte([],[]).
inverte([X|L],I) :-
  inverte(L,LI),
  concatena(LI,[X],I).
```

O predicado `inverte` devolve-nos, no segundo argumento, a lista invertida que é fornecida como primeiro parâmetro. Assim, por exemplo,

```
?- inverte([1,2,3],L).
   L = [3,2,1]

?- inverte([1,[2,3],4],L).
   L = [4,[2,3],1]
```

Pretende-se que implemente agora um predicado `inverte_tudo` que faz a inversão a todos os níveis (caso a lista contenha listas de entre os seus elementos, essas listas também são invertidas). Assim, para os exemplos atrás considerados, teríamos:

```
?- inverte_tudo([1,2,3],L).
   L = [3,2,1]

?- inverte_tudo([1,[2,3],4],L).
   L = [4,[3,2],1]
```

b): Ainda em relação ao predicado `concatena/3` da alínea anterior, mostre, usando a *Árvore de Prova* que o átomo

```
concatena([a,b], [c], [a,b,c]).
```

é verdadeiro.

c): Ainda em relação ao predicado `inverte/2` da alínea anterior, mostre, usando a *Árvore de Procura* que a resposta do interpretador de Prolog à questão

```
inverte([a,b,c], [c,a,b]).
```

é "`no`".

d): Pretende-se que implemente agora um predicado denominado `merge`, que serve para fundir duas listas ordenadas produzindo uma terceira lista, também ela ordenada. *Exemplo:*

```
?- merge([2,5,6,6,8],[1,3,5,9],L).
   L = [1,2,3,5,5,6,6,8,9]
```

Pressuponha a existência de um predicado `antes(X,Y)` que resulta verdadeiro quando `X` vem antes de `Y`. Para o exemplo apresentado, pode pensar nos termos da seguinte definição:

```
antes(X,Y) :- X =< Y.
```

Possível resolução.

```
% a)
      inverte_tudo([],[]).
      inverte_tudo([[X|R]|L],I) :-
        inverte_tudo([X|R],XRI),
        inverte_tudo(L,LI),
        concatena(LI,[XRI],I).
      inverte_tudo([X|L],I) :-
        inverte_tudo(L,LI),
        concatena(LI,[X],I).
% d)
      merge([],L,L).
      merge(L,[],L).
      merge([X1|R1],[X2|R2],[X1|R]) :-
        antes(X1,X2),
        merge(R1,[X2|R2],R).
      merge(L,[X2|R2],[X2|R]) :-
        merge(L,R2,R).
```

Exercício do Exame de Recurso de 1998: Problema 28. Implemente em `Prolog` o predicado `separa/4`, que recebe no primeiro argumento um objecto, no segundo uma lista, e nos devolve no terceiro argumento todos os elementos da lista antes da primeira ocorrência do objecto e no último parâmetro os elementos que sucedem na lista ao objecto em questão. Exemplo:

```
?- separa(*,[1,2,3,4,*,a,b,c,d],A,D).
   A = [1,2,3,4]
   D = [a,b,c,d]
```

4
Introdução ao controlo

1. O corte

Introduzimos um novo átomo, denotado por `!`, e chamado *corte* (ou *corta-escolhas*).

O corte não tem significado lógico, mas apenas um significado operacional determinístico, ligado ao percurso standard da árvore de procura.

Para o definir, é cómodo usar a terminologia procedimental. O corte é um procedimento *pré-definido*, quer dizer, apenas poderá figurar no corpo de uma cláusula, ou numa questão (alvo).

A chamada ao procedimento `!` tem imediatamente êxito, como se o corte fosse definido pela única cláusula (facto)

```
!.
```

Mas o que é essencial é "o efeito lateral" que acompanha o êxito dessa chamada e que modifica a árvore de procura corrente (modifica o percurso-construção). Para descrever esse efeito lateral consideremos um nó `N2` da árvore de procura onde o corte é o átomo escolhido. Seja `N1` o nó pai do nó onde esse corte aparece: o átomo escolhido em `N1` é unificado com a cabeça da cláusula cujo corpo contém esse corte. O efeito do corte é o de suprimir todas as escolhas em espera depois do nó `N1` (incluído): corta os ramos da árvore de procura ainda não percorridos e situados à direita da descida de `N1` a `N2`. Estes ramos sendo suprimidos, não serão portanto percorridos por ocasião dos retrocessos.

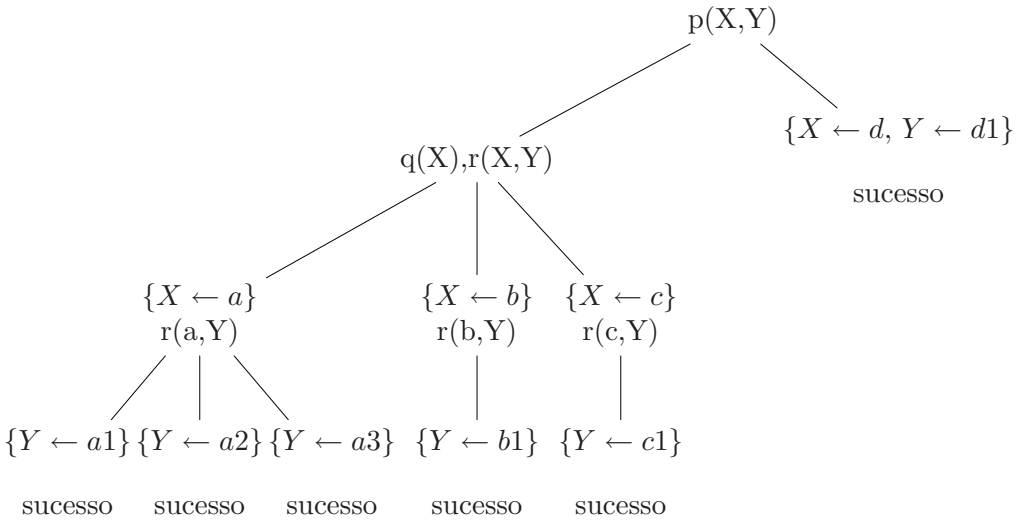
Vejamos alguns exemplos que ilustram o mecanismo. Para isso consideremos o seguinte programa:

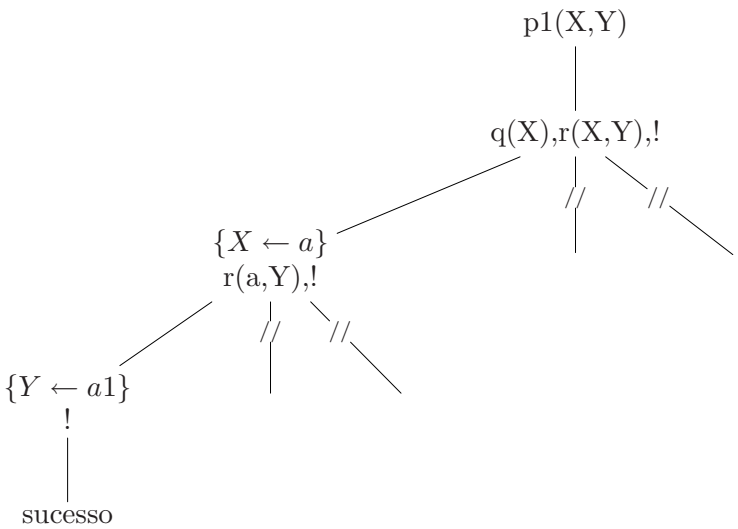
```
q(a).
q(b).
q(c).
r(b,b1).
r(c,c1).
r(a,a1).
r(a,a2).
r(a,a3).

p(X,Y) :- q(X), r(X,Y).
p(d,d1).

p1(X,Y) :- q(X), r(X,Y), !.

p2(X,Y) :- q(X), !, r(X,Y).

p3(X,Y) :- !, q(X), r(X,Y).
p3(d,d1).
```

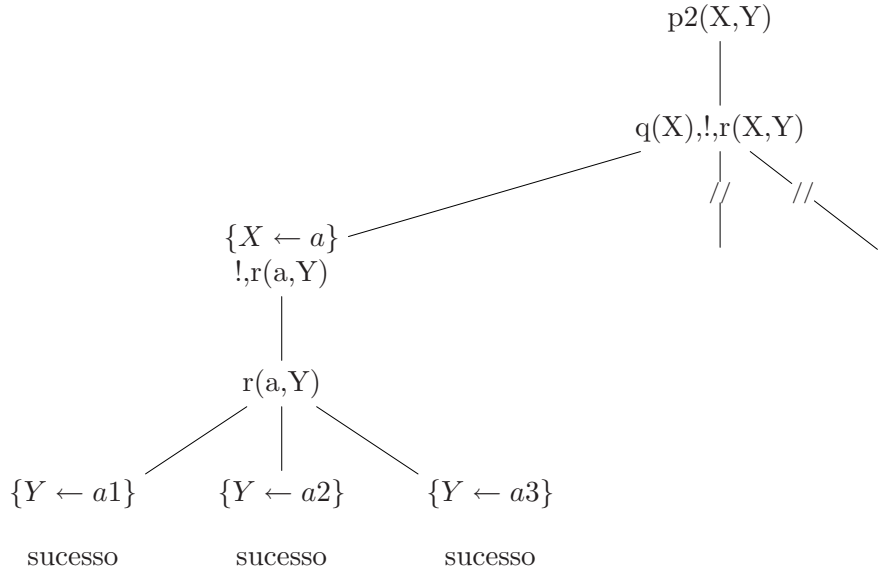
e as questões:

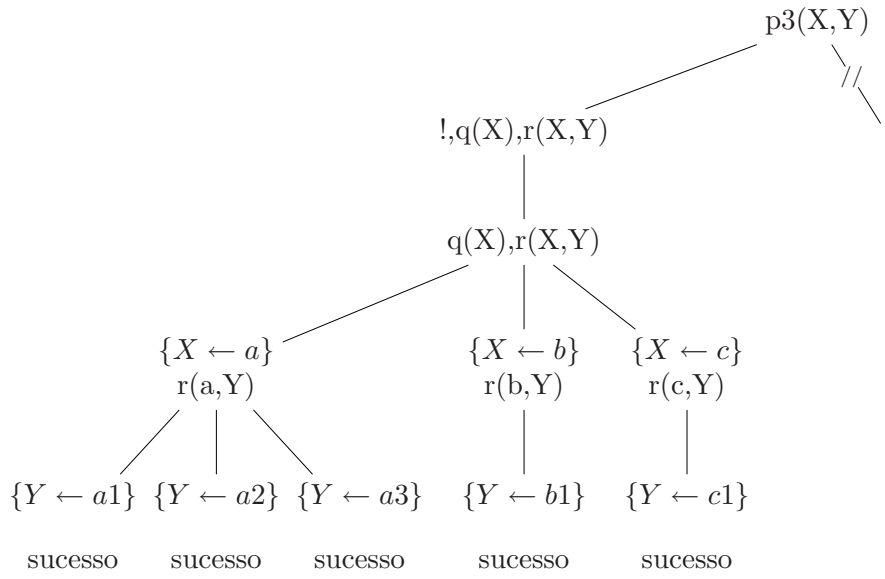
```
?- p(X,Y).
?- p1(X,Y).
?- p2(X,Y).
?- p3(X,Y).
```

Eis as quatro árvores de procura (simplificadas). Os ramos cortados são marcados com `//`









2. Aplicações do corte

O corte pode servir para fazer com que a procura pare a partir do momento que uma primeira resposta é encontrada. Por exemplo para o nosso programa de relações de parentesco a questão

```
?- avo(rosa,N), !.
```

faz com que obtenhamos apenas a resposta $\{N \leftarrow ana\}$.

O corte pode também ser usado para *"optimizar"* o programa, evitando procuras inúteis.

3. Para além do formalismo lógico

Vamos agora enriquecer a linguagem, construindo certos predicados que, tal como o corte, têm uma semântica operacional que ultrapassa o formalismo da lógica.

Introduzimos um novo símbolo de predicado de aridade um, denotado por `call`. Tal como o corte, trata-se de um predicado pré-definido, isto é, que pode apenas figurar no corpo de uma cláusula ou num alvo-questão.

Consideremos um átomo da forma `call(A)` onde `A` pode ser uma variável ou um átomo.

No decorrer do percurso-construção da árvore de procura, quando o átomo escolhido for `call(A)`, `A` deve estar instanciado por um átomo (senão erro). Então tudo se passa como se o programa comportasse (nessa etapa apenas) a cláusula

```
call(A) :- A.
```

Na terminologia procedimental, `call(A)` é equivalente à chamada do procedimento `A` e tem, por conseguinte, o efeito de tentar construir uma árvore de prova tendo por raiz uma instância de `A`.

Vamos ver de seguida alguns exemplos de utilização do corte e do `call`.

3.1. Condicional. Definimos o predicado `ifthenelse` pelas duas cláusulas seguintes (pela ordem indicada ...):

```
ifthenelse(X,Y,Z) :- call(X), !, call(Y).
ifthenelse(X,Y,Z) :- call(Z).
```

Consideremos agora um programa `P` ao qual juntamos estas duas cláusulas.[1]

Consideremos uma chamada ao procedimento `ifthenelse(X,Y,Z)` onde `X, Y, Z` estão respectivamente instanciados pelos átomos (= chamadas de procedimentos) `A, B, C`. (De facto não é necessariamente verdade que `X, Y, Z` estejam instanciados por `A, B, C` no momento da chamada a `ifthenelse`. É suficiente que assim seja no momento das chamadas correspondentes a `call`.)

Examinando a árvore de procura, vemos que o controlo é primeiro passado a `A` e que em seguida, se `A` tiver êxito, o controlo é passado a `B` não havendo retrocesso; se `A` falhar em tempo finito, o controlo é passado a `C`.

Uma leitura natural para `ifthenelse(X,Y,Z)` é então:

Se `X` Então `Y` Senão `Z`.

3.2. Negação como falha. Um programa `P` não pode produzir informação negativa no sentido da negação lógica.

De seguida assumimos a *"Hipótese do Mundo Fechado"* (teoria das bases de dados) e vamos produzir informação negativa mas num sentido diferente: substituímos

(i): "`not(A)` é consequência lógica de `P`" (negação lógica)

por

(ii): "`A` não é consequência lógica de `P`" (negação como falha)

Para apreciar a diferença entre (i) e (ii), podemos observar que:

- (ii) é verdade para certos `A` e falso para outros;
- (i) nunca é verdade: a negação de um átomo `A`, `not(A)`, não é consequência lógica de um programa. (Na prática os interpretadores Prolog têm `not` como uma palavra *reservada*, para que nunca possamos colocar no nosso programa o facto `not(A)`.)

Podemos acentuar a diferença entre (i) e (ii) notando que (ii) pode ser escrito como

(ii): não(`A` é consequência lógica de `P`)

Temos usado a designação `falha` quando um átomo não é unificável com nenhuma cabeça de cláusula.

A *negação como falha* (denotada por `not(X)` é então definida pelas duas cláusulas seguintes:

```
not(X) :- call(X), !, falha.
not(X).
```

Uma chamada a `not(X)`, onde `X` está instanciada por um átomo `A`, faz com que: se `A` tiver êxito, então `not(A)` falha; se `A` falha em tempo finito, `not(A)` tem êxito.

Na prática temos de assegurar que não existe nenhum facto

```
falha.
```

no nosso programa! Com essa finalidade os interpretadores Prolog têm a constante `fail` como palavra reservada.

Como geralmente os interpretadores de Prolog não efectuam, automaticamente, os retrocessos, para encontrar todas as respostas a uma questão, podemos "forçar" esses retrocessos substituindo uma questão

```
?- A.
```

por uma questão da forma:

```
?- A, "write resposta", fail.
```

[1] O predicado `ifthenelse` já se encontra disponível na maior parte das implementações Prolog. Nesse caso, deverá usar outro nome para o predicado.

Há todo um conjunto de questões de 'Fundamentos da Lógica' que esta negação como falha levanta. Aqui vamos apenas dar alguns exemplos que mostram que o seu uso é delicado.

Consideremos um programa cuja única cláusula é

```
igual(X,X).
```

e sejam `a` e `b` duas constantes. Notemos que, segundo a visão procedimental, `igual` é o procedimento de unificação, uma vez que a chamada `igual(A1,A2)` tem por efeito unificar `A1` a `A2`.

A chamada a `not(igual(a,b))` tem êxito:

```
?- not(igual(a,b)).
   yes
```

enquanto a chamada `not(igual(a,a))` falha:

```
?- not(igual(a,a)).
   no
```

Consideremos agora os alvos seguintes:

```
?- igual(X,a), not(igual(X,b)).
   X = a
   yes
?- not(igual(X,b)), igual(X,a).
   no
```

A segunda questão falha, porque `igual(X,b)` tem êxito com $\{X \leftarrow b\}$.

Juntemos agora ao programa que fizemos para o Problema 1, das relações de parentesco, as seguintes cláusulas:

```
masculino(paulo).
masculino(gustavo).
masculino(luis).
masculino(eusebio).

igual(X,X).

irmao(X,Y) :-
  not(igual(X,Y)),
  masculino(X),
  progenitores(P,M,X),
  progenitores(P,M,Y).
```

na esperança de definir a relação "é irmão de". À questão

```
?- irmao(eusebio,ana).
```

recebemos uma confirmação

```
yes
```

mas já a questão

```
?- irmao(X,ana).
```

falha (`no`) e portanto não nos dá a resposta

```
X = eusebio
```

Isto acontece porque `igual(X,ana)` tem êxito com `X = ana`.

O mesmo fenómeno acontece com a questão

```
?- irmao(X,Y).
   no
```

Mas se definirmos "é irmão de" por:

```
irmao(X,Y) :-
  masculino(X),
  progenitores(P,M,X),
  progenitores(P,M,Y),
  not(igual(X,Y)).
```

então teremos

```
?- irmao(eusebio,ana).
   yes

?- irmao(X,ana).
   X = eusebio
   yes

?- irmao(X,Y).
   X = eusebio, Y = ana
   yes
```

3.3. Repete. Introduzimos agora um novo símbolo de predicado de aridade zero, denotado por `repeat`, e que definimos pelas duas cláusulas seguintes:

```
repeat.
repeat :- repeat.
```

Resulta claro que no que diz respeito à *semântica lógica*, isto equivale à única cláusula

```
repeat.
```

e é portanto a sua *semântica operacional* que faz a diferença (e o seu interesse).
Para ilustrarmos a utilidade de tal construção, consideremos a questão

```
?- repeat, write('Qual a opcao desejada?'),
           read(O), faz(O), !.
```

Um exame à árvore de procura mostra que desencadeamos uma iteração, "sem fim", de chamadas ao `write` e ao `read`, cuja condição de saída é o êxito do predicado `faz`.

4. Exercícios

Exercício do Exame de Recorrência de 1998: Problema 29. Analise atentamente a seguinte Base de Conhecimento que o Gabinete de Assuntos Económicos da CE criou para determinar automaticamente se um País está apto a aderir ao projecto de introdução do EURO

```
%percentagem da Divida Publica em relacao ao PIB
percDivPub(belgica,135.0).
percDivPub(portugal,69.6).
............

%taxa de Inflaccao
taxaInfl(belgica,1.4).
taxaInfl(portugal,3.8).
............

menorTaxaInfl(finlandia,1.0).

%percentagem do Deficite Orcamental em relacao ao PIB
percDefOrc(belgica,5.1).
percDefOrc(portugal,2.8).
............

%criterios de aptidao
criterio1(P) :- percDivPub(P, D), D=< 75, !,
                write(P), write('Passou criterio1'),nl.
criterio1(P) :- write(P), write('NAO Passou criterio1'),nl.
criterio2(P) :- taxaInfl(P, T), menorTaxaInfl(_,M), T=< 2.5 * M.
criterio3(P) :- percDefOrc(P, D), D=< 3.

%verificacao da aptidao
apto(Pais) :- criterio1(Pais), criterio2(Pais), criterio3(Pais).
```

Responda, então, às alíneas seguintes:

a): Diga por palavras suas e com base na BC acima, quais são as condições em que um País pode aderir ao projecto.

b): Diga que alterações deveria fazer às condições que indicou na alínea anterior, se o predicado `apto` fosse definido da seguinte maneira:

```
%verificacao da aptidao
apto(Pais) :- verificaDivida(Pais), verificaInflaccao(Pais).
verificaDivida(Pais) :- criterio1(Pais), criterio3(Pais).
verificaInflaccao(Pais) :- criterio2(Pais).
```

c): Essa BC permite-lhe calcular automaticamente todos os Países aptos a aderirem, ou só serve para saber se um País está apto? Em caso afirmativo, diga como interrogava a BC para obter essa resposta.

d): Para que será que `criterio1` está definido de uma forma diferente de `criterio2` e `criterio3`? Era mesmo necessário, ou é um complemento?

e): Explique as razões lógicas e operacionais para se usar o predicado `cut, '!'` na primeira cláusula relativa a `criterio1`.
Que aconteceria se se retirasse o `cut` e o segundo critério falhasse (obviamente, depois do primeiro ter sido testado com sucesso)?

5
Cálculo do valor de expressões matemáticas

Um dos conceitos mais úteis em ciências da computação, é o conceito de *pilha*. Uma *pilha* é uma colecção ordenada de dados, na qual os dados podem ser inseridos e retirados apenas de uma das extremidades. A essa extremidade chamamos *topo da pilha*. Neste capítulo fazemos uso desta estrutura de dados que, em Prolog, é trivialmente implementada por uma lista.

1. Notação infixa, prefixa e posfixa

Consideremos a soma de A e B, que representamos usualmente por $A + B$. Esta representação particular é chamada de *infixa*. Existem contudo duas notações alternativas. Estas são

$$\begin{array}{ll} +A\,B & \text{prefixa,} \\ A\,B\,+ & \text{posfixa.} \end{array}$$

Na notação prefixa o operador precede os operandos enquanto na posfixa o operador vem depois dos operandos. As notações prefixa e posfixa não são tão inusuais como podem parecer à primeira vista. Por exemplo, se estivermos a usar um predicado Prolog para a soma de dois argumentos A e B, invocamos qualquer coisa como $soma(A, B, R)$. O operador *soma* precede os operandos A e B!

O cálculo da expressão $A + B * C$, escrita na notação usual, infixa, requer conhecimento de qual das duas operações, $+$ ou $*$, deve ser realizada em primeiro lugar: "sabemos" que a multiplicação deve ser feita antes da adição. Assim $A+B*C$ é interpretado como $A+(B*C)$. Dizemos que a multiplicação tem *precedência* sobre a adição. Suponhamos que queremos escrever $A + B * C$ na notação posfixa. Aplicando as regras de precedência, convertemos em primeiro lugar a porção da expressão que é calculada em primeiro lugar: a multiplicação. Fazendo esta conversão por etapas obtemos:

$$\begin{array}{cc} A + (B * C) & \text{parêntesis para dar ênfase} \\ A + (B\,C*) & \text{conversão da multiplicação} \\ A\ (B\,C*)\,+ & \text{conversão da soma} \\ A\,B\,C\,*\,+ & \text{notação posfixa.} \end{array}$$

As únicas regras a lembrar, durante o processo de conversão, é que as operações com maior precedência são convertidas em primeiro lugar; e que depois de uma porção da expressão ter sido convertida para a notação posfixa, ela é tratada como um único operando. Consideremos o mesmo exemplo com a precedência dos operadores invertida pelo uso deliberado de parêntesis:

$(A+B)*C$	forma infixa
$(A\,B+)*C$	conversão da adição
$(A\,B+)\;C*$	conversão da soma
$A\,B+C*$	notação posfixa.

As regras para a conversão da notação infixa para posfixa são simples, desde que saibamos à priori a ordem de precedência dos vários operadores.

Consideramos, nos exemplos que se seguem, cinco operações binárias: adição, subtracção, multiplicação, divisão e exponenciação que representaremos respectivamente pelos símbolos $+$, $-$, $*$, $/$ e $\hat{}$. O valor da expressão $A\hat{}B$ é A levantado a B, de tal modo que $3\hat{}2$ é 9. Para estas operações temos a seguinte ordem de precedência (maior para menor):

exponenciação
multiplicação e divisão
adição e subtracção.

Usando parêntesis, podemos mudar as precedências por defeito. Damos os seguintes exemplos adicionais de conversão infixa para posfixa. Avancem apenas quando os conseguirem fazer por vocês próprios. Notar que, na ausência de parêntesis, os operadores da mesma precedência são considerados da esquerda para a direita, com a única excepção da exponenciação onde a ordem é assumida como da direita para a esquerda. Assim, $A+B+C$ significa $(A+B)+C$ enquanto $A\hat{}B\hat{}C$ significa $A\hat{}\,(B\hat{}C)$.

Infixa	Posfixa
$A+B$	$A\,B+$
$A+B-C$	$A\,B+C-$
$(A+B)*(C-D)$	$A\,B+C\,D-*$
$A\hat{}B*C-D+E/F/(G+H)$	$A\,B\hat{}C*D-E\,F/G\,H+/+$
$((A+B)*C-(D-E))\,\hat{}\,(F+G)$	$A\,B+C*D\,E--F\,G+\hat{}$
$A-B/\,(C*D\hat{}E)$	$A\,B\,C\,D\,E\hat{}*/-$

Notar que na notação posfixa de uma expressão não há parêntesis: a ordem dos operadores na forma posfixa determina a ordem das operações no cálculo do valor das expressões. Perdemos assim a capacidade de notar de imediato que operandos estão associados com um particular operador, mas ganhamos uma forma não ambígua sem o uso incómodo de parêntesis.

2. Cálculo do valor de uma expressão posfixa

Pretendemos ter um método que nos permita concluir, por exemplo, que $3\,4\,5*+=23$ e $3\,4+5*=35$. Por outras palavras, desejamos um algoritmo para o cálculo de expressões posfixas.

Cada operador numa "string" posfixa deve ser aplicado aos operandos que imediatamente lhe antecedem. Esses operandos podem por sua vez resultar da aplicação de um ou mais operadores. Suponhamos que, de cada vez que lemos um operando, fazemos *push* desse operando para uma pilha. Quando chegarmos a um operador, os seus operandos estarão no topo da pilha: fazemos um número de *pop's* correspondente ao número de argumentos do operador em questão. Depois de efectuada a operação, fazemos *push* do resultado para a pilha, de modo a que ele esteja disponível para uso como operando do próximo operador. Vamos ilustrar este algoritmo com o exemplo $6\,2\,3+-3\,8\,2/+*2\hat{}3+$

Símbolo corrente	**Descrição**	**"Pilha"**
6	Fazer "push" do 6	[6]
2	Fazer "push" do 2	[2, 6]
3	Fazer "push" do 3	[3, 2, 6]
+	Retirar os 2 elementos do topo: 3 e 2	[6]
	Fazer "push" do valor de $2+3$	[5, 6]
−	Retirar os 2 elementos do topo: 5 e 6	[ ]
	Fazer "push" do valor de $6-5$	[1]
3	Fazer "push" do 3	[3, 1]
8	Fazer "push" do 8	[8, 3, 1]
2	Fazer "push" do 2	[2, 8, 3, 1]
/	Retirar os 2 elementos do topo: 2 e 8	[3, 1]
	Fazer "push" do valor de $8/2$	[4, 3, 1]
+	Retirar os 2 elementos do topo: 4 e 3	[1]
	Fazer "push" do valor de $3+4$	[7, 1]
*	Retirar os 2 elementos do topo: 7 e 1	[ ]
	Fazer "push" do valor de $1*7$	[7]
2	Fazer "push" do 2	[2, 7]
^	Retirar os 2 elementos do topo: 2 e 7	[ ]
	Fazer "push" do valor de 7^2	[49]
3	Fazer "push" do 3	[3, 49]
+	Retirar os 2 elementos do topo: 3 e 49	[ ]
	Fazer "push" do valor de $49+3$	[52]

O resultado da nossa expressão é então 52.

3. Conversão de uma expressão infixa para posfixa

Vamos agora apresentar um algoritmo para converter uma expressão infixa numa posfixa.

Já vimos que as subexpressões dentro dos parêntesis mais interiores, devem ser convertidas para a notação posfixa em primeiro lugar, de modo a puderem ser tratadas como operandos. Deste modo, os parêntesis podem ser sucessivamente eliminados até toda a expressão ser convertida. O último par de parêntesis a ser aberto dentro de um grupo de parêntesis, contém a primeira subexpressão dentro desse grupo a ser transformada. Este comportamento sugere imediatamente o uso de uma pilha.

Consideremos as duas expressões infixas $A+B*C$ e $(A+B)*C$ e as correspondentes versões posfixas $A\,B\,C\,*\,+$ e $A\,B\,+\,C*$. Em cada um dos casos a ordem dos operandos é a mesma que na expressão original infixa. Ao "varrermos" a primeira expressão, $A+B*C$, o primeiro operando A pode ser imediatamente inserido na expressão posfixa. Obviamente, o símbolo + não pode ser inserido até ao seu segundo operando, que ainda não foi lido, ser inserido. Por conseguinte, temos de o armazenar à parte (numa pilha) para mais tarde inseri-lo na sua posição correcta. Quando o operando B é lido, ele é inserido imediatamente após o A. Agora, contudo, dois operandos foram lidos. O que impede o símbolo +, armazenado na pilha, de ser inserido? A resposta é, como resulta claro, o facto do símbolo $*$ que se segue ter precedência sobre o +. (No caso da segunda expressão os parêntesis indicam que a operação + deve ser realizada em primeiro lugar.) Lembrar que na notação posfixa o operador que aparece mais cedo na string é a que a deve ser aplicada em primeiro lugar.

Vamos assumir a existência de um predicado *prcd(Op1,Op2)* que nos indica que Op1 tem precedência sobre Op2 quando Op1 nos aparece à esquerda de Op2 numa expressão infixa sem parêntesis. Por exemplo *prcd(*,+)* e *prcd(+,+)* devem constar da BC enquanto *prcd(+,*)* não! Assim, na ausência de parêntesis, o nosso algoritmo de conversão infixa->posfixa pode ser descrito nos seguintes termos: *vamos lendo os símbolos da expressão infixa. Sempre que o símbolo é um operando colocamo-lo de imediato na expressão posfixa. No caso de um operador das duas uma: ou existe um operador no topo da pilha (que no início está vazia) com precedência sobre ele: caso em que retiramos o operador do topo da pilha para a expressão posfixa; ou não: neste caso não podemos colocar o operador na expressão posfixa até termos lido o próximo operador, que poderá ter precedência. Colocamo-lo então na pilha.*

Quando acabarmos de ler toda a expressão infixa, resta-nos retirar todos os operadores que restam na pilha para a expressão posfixa.

Propomos que simulem o algoritmo acima para as expressões infixas 2∗3+4∗5 e 1+2∗2ˆ3ˆ2 (onde o símbolo ˆ representa a exponenciação e $prcd(\hat{}, \hat{})$ não deve constar da base de conhecimento!) para se convencerem que ele está correcto. Notar que em cada ponto da simulação, um operador na Pilha tem uma precedência inferior em relação a todos os operadores abaixo dele. Isto é verdade porque a pilha inicialmente está vazia (o que trivialmente satisfaz a condição) e sempre que é feito "push" de um operador, isso significa que o operador correntemente no topo da pilha tem precedência inferior em relação ao operador a vir para a pilha.

Falta-nos responder à seguinte questão: que modificações devem ser feitas ao algoritmo para podermos lidar com parêntesis? A resposta é animadora: pouco! Sempre que um parêntesis é aberto, ele deve ser colocado no topo da pilha. Isto pode ser conseguido de um modo automático pelo nosso algoritmo se encararmos '(' como um operador.

Quando um parêntesis ')' é lido, todos os operadores até ao primeiro '(' devem ser retirados do topo da pilha para a expressão posfixa. Isto pode ser conseguido colocando na base de conhecimento *prcd(Op,')')* para todos os operadores Op que não o '('. Quando estes operadores tiverem sido retirados da pilha devemos ter o cuidado de retirar o parêntesis '(' e desprezá-lo conjuntamente com o seu par ')' (não devemos colocá-los na expressão posfixa!).

Exemplos.

Ex. 1: $A + B * C$

Símbolo lido	**Expressão posfixa**	**Pilha**
A	A	[ ]
$+$	A	$[+]$
B	$A\,B$	$[+]$
$*$	$A\,B$	$[*, +]$
C	$A\,B\,C$	$[*, +]$
	$A\,B\,C*$	$[+]$
	$A\,B\,C*+$	[ ]

Ex. 2: $(A + B) * C$

Símbolo lido	**Expressão posfixa**	**Pilha**
$($		$[($
A	A	$[($
$+$	A	$[+, ($
B	$A\,B$	$[+, ($
$)$	$A\,B+$	[ ]
$*$	$A\,B+$	$[*]$
C	$A\,B+C$	$[*]$
	$A\,B+C*$	[ ]

Ex. 3: $((A-(B+C))*D)\,\hat{}\,(E+F)$

Símbolo lido	**Expressão posfixa**	**Pilha**
(		[(]
(		[(, (]
A	A	[(, (]
$-$	A	[$-$, (, (]
(	A	[(, $-$, (, (]
B	AB	[(, $-$, (, (]
$+$	AB	[$+$, (, $-$, (, (]
C	ABC	[$+$, (, $-$, (, (]
)	$ABC+$	[$-$, (, (]
)	$ABC+-$	[(]
$*$	$ABC+-$	[$*$, (]
D	$ABC+-D$	[$*$, (]
)	$ABC+-D*$	[]
$\hat{}$	$ABC+-D*$	[$\hat{}$]
(	$ABC+-D*$	[(, $\hat{}$]
E	$ABC+-D*E$	[(, $\hat{}$]
$+$	$ABC+-D*E$	[$+$, (, $\hat{}$]
F	$ABC+-D*EF$	[$+$, (, $\hat{}$]
)	$ABC+-D*EF+$	[$\hat{}$]
	$ABC+-D*EF+\hat{}$	[]

4. Programa em Prolog para o cálculo de expressões matemáticas

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Delfim F. Marado Torres
% Ultima Alteracao: 21/Abril/1999
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Opera,c~oes sobre listas
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% primeiros(N,Lista,Primeiros_N_da_Lista,Resto_da_Lista)

primeiros(0,L,[],L).
primeiros(N,[X|L],[X|R],Sobra) :-
  N1 is N - 1,
  primeiros(N1,L,R,Sobra).

concatena([],L,L).
concatena([X|R],L,[X|C]) :-
  concatena(R,L,C).

inverte([],[]).
inverte([X|L],I) :-
  inverte(L,LI),
  concatena(LI,[X],I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Opera,c~oes matem'aticas. O primeiro argumento
% corresponde sempre ao resultado
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%mais/3
        mais(R,X,Y) :- R is X + Y.

%menos/3

        menos(R,X,Y) :- R is X - Y.

%vezes/3

        vezes(R,X,Y) :- R is X * Y.

%dividir/3

        dividir(R,X,Y) :- R is X / Y.

%pow/3

        pow(1,_,0).
        pow(R,X,N) :-
          N1 is N - 1,
          pow(R1,X,N1),
          R is X * R1.

%factorial/2

    factorial(1,0).
        factorial(F,N) :-
      N1 is N - 1,
          factorial(F1,N1),
          F is N * F1.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tabela de correspond^encia s'imbolos -> predicados
%
% tabela(Simbolo,Nome_Predicado,Aridade_Predicado)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tabela(+,mais,3).
tabela(-,menos,3).
tabela(*,vezes,3).
tabela(/,dividir,3).
tabela(^,pow,3).
tabela(!,factorial,2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tabela de preced^encias
%
% prcd(Op1,Op2) significa que Op1 tem preced^encia sobre Op2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prcd(+,+).
prcd(-,-).
prcd(*,*).
prcd(/,/).
prcd(*,+).
prcd(/,+).
```

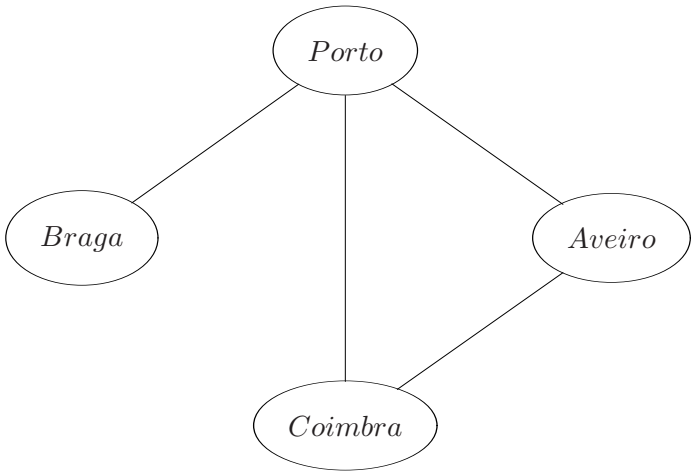
```
prcd(*,-).
prcd(/,-).
prcd(^,*).
prcd(^,/).
prcd(^,+).
prcd(^,-).
prcd(!,*).
prcd(!,/).
prcd(!,+).
prcd(!,-).
prcd(!,^).

prcd(Op,')') :-
  tabela(Op,_,_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% C'alculo do valor de express~oes num'ericas na nota,c~ao postfix
%
%  Exemplos:
%              ?- postfix([2,3,5,*,+],Valor).
%              Valor = 17
%
%              ?- postfix([6,2,3,+,-,3,8,2,/,+,*,2,^,3,+],Valor).
%              Valor = 52
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

postfix(L,V) :-
  calcula_postfix(L,V,[]).      % No in'icio a pilha est'a vazia

calcula_postfix([],V,[V]).      % Quando a express~ao acabou, o resultado
                                % 'e o 'unico elemento da pilha.
calcula_postfix([S|R],V,P) :-
  number(S),
  calcula_postfix(R,V,[S|P]).   % Faz push para a pilha

calcula_postfix([S|R],V,P) :-
  tabela(S,Op,Ar),              % Se n~ao 'e n'umero 'e operador
  N is Ar - 1,
  primeiros(N,P,Nums,RP),
  inverte(Nums,Numeros),
  C =.. [Op,Res|Numeros],
  call(C),
  calcula_postfix(R,V,[Res|RP]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Convers~ao de uma express~ao em infix para postfix
% Exemplos:
%   ?- infix_to_postfix([1,+,2,*,3],P).
%   P = [1,2,3,*,+]
%
%   ?- infix_to_postfix(['(',1,+,2,')',*,3],P).
%   P = [1,2,+,3,*]
%
%   ?- infix_to_postfix(
%       ['(','(',1,-,'(',2,+,3,')',')',*,4,')',^,'(',5,+,6,')'],P).
%   P = [1,2,3,+,-,4,*,5,6,+,^]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
infix_to_postfix(I,PF) :-
  inf_to_post(I,[],[],PF).

inf_to_post([],S,P,PF) :-
  inverte(S,SI),
  concatena(SI,P,PF).


inf_to_post([X|I],S,P,PF) :-
  number(X),
  inf_to_post(I,[X|S],P,PF).
inf_to_post(I,S,[')'|P],PF) :-
  inf_to_post(I,S,P,PF).
inf_to_post([X|I],S,[Y|P],PF) :-
  prcd(Y,X),
  inf_to_post([X|I],[Y|S],P,PF).
inf_to_post([')'|I],S,[_|P],PF) :-
  inf_to_post(I,S,P,PF).
inf_to_post([X|I],S,P,PF) :-
  inf_to_post(I,S,[X|P],PF).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Convers~ao de uma string para uma lista
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
str_list(S,L) :-
  name(S,LA),
  converte(LA,L).

converte([],[]).
converte([32|CA],C) :-  % despreza espa,cos
  converte(CA,C).
converte([XA|CA],[X|C]) :-
  name(X,[XA]),
  converte(CA,C).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%             PREDICADO PRINCIPAL                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% C'alculo do valor de express~oes
%
%  Exemplos:
%          ?- valor('((1 - (2+3))*4)^5 + 3!',V).
%          V =  -1048570
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

valor(String,Val) :-
  str_list(String,Exp),
  infix_to_postfix(Exp,ExpPost),
  postfix(ExpPost,Val), !.
```

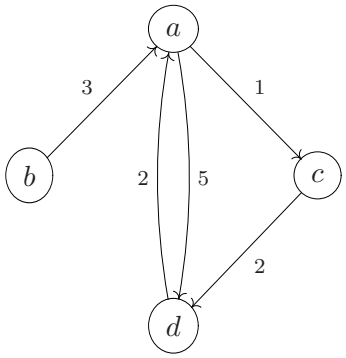
6
Grafos

Os grafos são estruturas matemáticas muito úteis em aplicações. Um *grafo* $G = (V, A)$ é definido por um conjunto finito de *nós* V (também chamados de *nodos* ou *vértices*) e por um conjunto de *arestas* ou *arcos* A. Cada aresta estabelece uma relação entre dois nós. Quando a relação entre os nós é descrita por pares *ordenados*, as arestas são dirigidas e diz-se que o grafo é *dirigido* ou *orientado*. Se as arestas são pares não ordenados, o grafo é dito *não dirigido* ou *não orientado*. Se existir um arco entre o vértice a e o vértice b dizemos que b é *adjacente* a a. Às arestas podem ser associados *pesos*, ou outro tipo de *etiquetas*, dependendo da aplicação. Segue-se um exemplo de um grafo não dirigido (G_1) e de um grafo dirigido com pesos associados às arestas (G_2):

G_1



G_2



1. Representação de grafos em Prolog

Podemos representar grafos em Prolog de várias maneiras. Um método será representar cada aresta separadamente como uma cláusula (um facto). Vamos considerar um predicado `arco` em que o primeiro argumento consiste no nome do grafo e o segundo e terceiros argumentos especificam o par de vértices que definem esse arco. Dependendo do caso, esse par pode ser encarado como um par ordenado (grafo dirigido) ou não. Se quisermos trabalhar em simultâneo com grafos dirigidos e não dirigidos, podemos convencionar que os arcos são sempre dirigidos, substituindo os arcos não dirigidos por dois dirigidos: um em cada sentido. No caso de *grafos pesados*, acrescentamos um quarto argumento ao predicado `arco` com o peso. Os grafos acima podem então ser representados em Prolog como se segue:

```
arco(g1,porto,braga).
arco(g1,braga,porto)
arco(g1,porto,coimbra).
arco(g1,coimbra,porto).
arco(g1,porto,aveiro).
arco(g1,aveiro,porto).
arco(g1,aveiro,coimbra).
arco(g1,coimbra,aveiro).

arco(g2,a,c,1).
arco(g2,a,d,5).
arco(g2,b,a,3).
arco(g2,c,d,2).
arco(g2,d,a,2).
```

Nesta representação, os vértices estão implícitos. `X` será um vértice do grafo `G` se existir no grafo pelo menos um arco a sair de `X` ou então a entrar:

```
vertice(X,G) :- arco(G,X,_).
vertice(X,G) :- arco(G,_,X).
```

Este facto levanta o problema da representação de grafos com *nós isolados*: nós que não estão ligados a nenhum outro nó do grafo. A maneira mais elegante de resolver este problema, consiste em arranjar uma outra representação alternativa, representação essa que tenha os vértices dados de forma explícita. Por exemplo, um grafo pode ser representado por um único facto com três argumentos: o primeiro com o nome do grafo, o segundo argumento com a lista de vértices e o último com a lista dos arcos. Assim sendo, podemos também representar os grafos G_1 e G_2 da seguinte maneira:

```
% Os arcos de g1 devem ser interpretados
% como pares nao ordenados.

grafo( g1,
       [aveiro,coimbra,porto,braga],
       [ [braga,porto], [coimbra,porto],
         [aveiro,porto], [aveiro,coimbra],
       ]
     ).

% Os arcos de g2 devem ser interpretados
% como pares ordenados.

grafo(g2,[a,b,c,d],[arco(b,a,3),arco(a,d,5),arco(d,a,2),
                    arco(a,c,1),arco(c,d,2)]).
```

Só para mostrar que existem de facto várias maneiras e filosofias de representar grafos em Prolog, vamos ainda considerar uma terceira alternativa em que associamos a cada nó uma lista dos nós que lhe são adjacentes. Desta maneira podemos representar grafos com nós isolados sem explicitar os vértices: um nó isolado terá como lista de nós adjacentes uma lista vazia e não pertencerá a nenhuma das listas de adjacência. Por exemplo:

```
grafo(g1,[ [aveiro,[coimbra,porto]],
           [braga,[porto]],
           [coimbra,[aveiro,porto]],
           [porto,[aveiro,coimbra,braga]]
         ]
     ).

grafo(g1,[ [a,[ [c,1],[d,5] ],
           [b,[ [a,3] ],
           [c,[ [d,2] ],
           [d,[ [a,2] ]
         ]
     ).
```

Nesta representação, podemos definir o predicado `vertice` como se segue:

```
vertice(X,G) :-
  grafo(G,L),
  member([X,_],L).
```

A escolha da melhor representação dependerá, sempre, da aplicação que tivermos em mente e das operações que desejarmos realizar sobre os grafos. Nas seguintes secções consideramos algumas operações típicas, que serão definidas em Prolog para a primeira representação que considerámos. O leitor é encorajado a defini-las paras as outras duas representações consideradas.

Um outro tipo de exercícios interessante, consiste em construir predicados que nos convertam grafos de uma representação para outra. A título de exemplo, vamos definir o predicado `convGrafo/1` que converte um grafo `G`, dado na primeira representação, para a segunda representação considerada acima:

```
convGrafo(G) :-
  findall(V,vertice(V,G),LV),
  findall(arco(G,X,Y,P),arco(G,X,Y,P),LA),
  asserta(grafo(G,LV,LA)),
  apagaTudo(arco(G,A,B)).

apagaTudo(A) :- retract(A), fail.
apagaTudo(_).
```

2. Encontrar um caminho

Seja `G` um grafo orientado e pesado, `A` e `Z` dois vértices de G. Vamos definir o predicado

```
caminho(G,A,Z,Cam,P)
```

onde `Cam` é um caminho acíclico, em `G`, entre `A` e `Z` e com peso `P`. O caminho `Cam` irá ser representado como uma lista de nós. Para o grafo G_2 que considerámos acima temos:

```
?- caminho(g2,a,d,C,P).
   C = [a,d]
   P = 5;

   C = [a,c,d]
   P = 3;

   no
```

Uma vez que estamos interessados em caminhos sem ciclos, um vértice do grafo pode aparecer no caminho quando muito uma vez. O predicado `caminho/5` é definido simplesmente como

```
caminho(G,A,Z,C,P) :-
  caminho1(G,A,Z,C,P,[]).
```

A relação `caminho1(G,A,Z,C,P,V)` encontra um caminho evitando os nós na lista `V` (os vértices já visitados). No início ainda não visitámos nenhum nó e é por isso que `caminho` "chama" `caminho1` com a lista de nós visitados vazia. Segue-se a definição de `caminho1/6`:

```
caminho1(G,A,Z,[A,Z],P,_) :-
  arco(G,A,Z,P).
caminho1(G,A,Z,[A|R],P,V) :-
  arco(G,A,X,P1),
  not member(X,V),
  caminho1(G,X,Z,R,P2,[X|V]),
  P is P1 + P2.
```

Recorrendo ao predicado `caminho1` podemos fazer questões do tipo:

"Quais os caminhos de G_2 entre `a` e `d` que não passam por `c`?"

Para isso basta inicializar a lista de nós visitados adequadamente:

```
?- caminho1(g2,a,d,C,_,[c]).
   C = [a,d];
   no
```

3. Caminho de Hamilton

Um problema clássico em grafos é o de encontrar o *caminho de Hamilton*, isto é, um caminho acíclico contendo todos os vértices do grafo. Usando o predicado `caminho` podemos obter um caminho de Hamilton `CH` do grafo `G` como se segue:

```
hamilton(CH,G) :-
  caminho(G,_,_,CH,_),
  not ( vertice(V,G), not member(V,CH) ).
```

4. Encontrar o caminho de custo mínimo e o de custo máximo

Quando existem pesos associados aos grafos, é, do ponto de vista das aplicações, importante saber qual o caminho de peso mínimo (quando os pesos representam "custos" e é pois importante *minimizar* esses custos) ou então o caminho de peso máximo (quando os pesos representam "lucros" e o que pretendemos é pois *maximizá-los*). De seguida implementamos os predicados `cmc`, *"Caminho Mais Curto"*, e `cml`, *Caminho Mais Longo"*:

```
cmc(G,A,Z,CMC) :-
  caminho(G,A,Z,CMC,P),
  not ( caminho(G,A,Z,_,P1), P1 < P ).

cml(G,A,Z,CML) :-
  caminho(G,A,Z,CML,P),
  not ( caminho(G,A,Z,_,P1), P1 > P ).
```

5. Árvores

A partir deste momento consideramos grafos não orientados: se for possível ir de `A` para `B` em `G`, também será possível ir de `B` para `A` em `G`.

Diz-se que um grafo é *conexo* se existir um caminho de qualquer nó para qualquer outro. Um grafo G diz-se uma *arvore@árvore* se e somente se:

1. G é conexo; e
2. não existem ciclos em G.

Vamos definir em Prolog o predicado `arvore(G)` que resulta verdadeiro se e somente se `G` for uma árvore:

```
arvore(G) :-
  conexo(G),
  not temCiclos(G).

conexo(G) :-
  not (vertice(V1,G), vertice(V2,G), not (caminho(G,V1,V2,_,_))).
```

```
temCiclos(G) :-
  arco(G,A,B,_),
  caminho(G,A,B,[A,_,_|_],_). % caminho com mais de 1 arco
```

5.1. Árvore geradora de um grafo. Diz-se que $A_{ger} = (V, A')$ é uma *arvore geradora@árvore geradora* do grafo $G = (V, A)$, se A_{ger} for uma árvore e A' for um subconjunto de A. De seguida definimos o predicado `arvoreGeradora(Ager,G)` que resulta verdadeiro quando e apenas quando `Ager` for uma árvore geradora de `G`:

```
arvoreGeradora(Ager,G) :-
  arvore(Ager),
  findall(arco(X,Y,P),arco(G,X,Y,P),LAG),
  findall(arco(X,Y,P),arco(Ager,X,Y,P),LAAger),
  subconjunto(LAAger,LAG),
  not ( vertice(V,G), not vertice(V,Ager) ). % mesmos vertices

subconjunto([],_).
subconjunto([X|R],L) :-
  member(X,L),
  subconjunto(R,L).
```

5.2. Árvores Binárias. Uma *arvore binária@árvore binária* é uma árvore em que cada vértice tem, no máximo, dois vértices adjacentes, chamados de *filho esquerdo* e *filho direito*. O vértice da árvore binária que não é filho de nenhum nó, é designado por *raiz* da árvore. A subárvore cuja raiz é o filho esquerdo (respectivamente direito) de um vértice v é chamada *subárvore esquerda* (respectivamente direita) de v. Os vértices sem nenhum filho são designados por *folhas*.

Uma definição recursiva de *arvore binária@árvore binária* é:

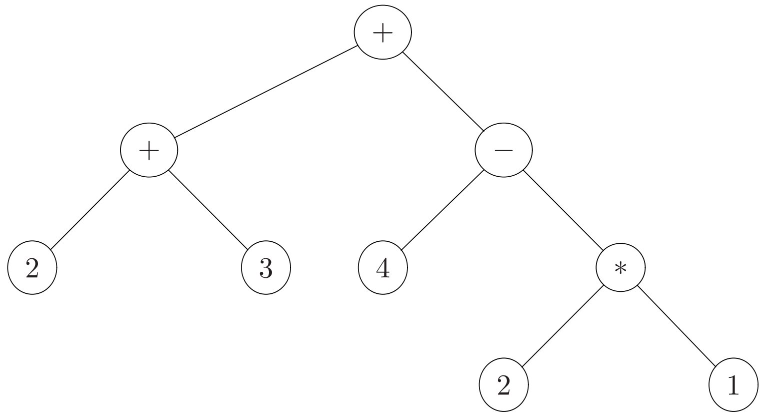
> Uma árvore binária ou é vazia (o conjunto de vértices é vazio) ou é formada por 3 partes:
>
> - uma raiz;
> - uma subárvore esquerda (também árvore binária); e
> - uma subárvore direita (também árvore binária).

A raiz pode ser qualquer coisa (pode conter qualquer tipo de informação), mas as subárvores têm de ser outra vez árvores binárias (daí ser uma definição recursiva).

É muito frequente em aplicações ser necessário, a partir de um dado vértice, aceder à subárvore esquerda ou direita. É nesse contexto que a definição recursiva acima nos dá uma representação de árvores binárias em Prolog adequada. Para isso precisamos de um átomo especial para representar a árvore vazia e precisamos de um símbolo de predicado para construir uma árvore binária não vazia a partir das suas três componentes (a raiz, a subárvore esquerda e a subárvore direita). Faremos as escolhas como se segue.

- A constante `vazia` representa a árvore vazia.
- Escolhemos o símbolo de predicado `arvBin` de tal modo que uma árvore binária de raiz `X`, subárvore esquerda `E` e subárvore direita `D` é representada por `arvBin(X,E,D)`.

Vejamos um exemplo.



será representada em Prolog como se segue:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% arvBin(Vertice,SubArvEsq,SubArvDir)
%
% Quando uma subarvore e' vazia usamos a constante
% 'vazia'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

arvBin( + ,
        arvBin( + ,
                arvBin(2,vazia,vazia),
                arvBin(3,vazia,vazia)
              ),
        arvBin( - ,
                arvBin(4,vazia,vazia),
                arvBin( * ,
                        arvBin(2,vazia,vazia),
                        arvBin(1,vazia,vazia)
                      )
              )
      ).
```

5.2.1. *Relação de pertença.* Vamos agora definir um predicado que defina a relação de pertença de um objecto numa árvore binária. Queremos que

```
pertence(X,AB)
```

seja verdadeiro apenas quando `X` for um dos vértices da árvore binária `AB`. O predicado `pertence/2` pode ser facilmente definido em Prolog se atendermos a que `X` pertence a `AB` se e só se umas das seguintes três alternativas ocorrer:

- `X` é a raiz de `AB`.
- `X` pertence à subárvore esquerda de `AB`.
- `X` pertence à subárvore direita de `AB`.

Segue-se a respectiva definição em Prolog:

```
pertence(X,arvBin(X,_,_)).
pertence(X,arvBin(_,E,_)) :-
  pertence(X,E).
pertence(X,arvBin(_,_,D)) :-
  pertence(X,D).
```

Obviamente que com esta definição a questão

```
?- pertence(X,vazia).
```

falha, independentemente do valor de `X`.

5.2.2. *Travessias.* Pretendemos agora visitar cada um dos vértices de uma árvore binária uma e uma só vez. Muitos algoritmos que utilizam árvores percorrem-nas numa ordem específica. Vamos definir as três travessias mais comuns.

A *travessia prefixa* consiste em visitar primeiro a raiz e depois visitar (recursivamente) a subárvore esquerda e a subárvore direita (nesta ordem com ambas as travessias prefixas). Numa *travessia posfixa*, as subárvores esquerda e direita são visitadas, nesta ordem, recursivamente com travessia posfixa, e só depois a raiz é visitada. Por fim a *travessia infixa* consiste em fazer (recursivamente) a travessia infixa da subárvore esquerda, visitar de seguida a raiz e finalmente usar a travessia infixa na subárvore direita.

Podemos definir estas travessias em Prolog como se segue:

```
travPrefixa(vazia,[]).
travPrefixa(arvBin(R,ABE,ABD),[R|L]) :-
  travPrefixa(ABE,LE),
  travPrefixa(ABD,LD),
  append(LE,LD,L).

travPosfixa(vazia,[]).
travPosfixa(arvBin(R,ABE,ABD),L) :-
  travPosfixa(ABE,LE),
  travPosfixa(ABD,LD),
  append(LE,LD,L1),
  append(L1,[R],L).

travInfixa(vazia,[]).
travInfixa(arvBin(R,ABE,ABD),L) :-
  travInfixa(ABE,LE),
  append(LE,[R],L1),
  travInfixa(ABD,LD),
  append(L1,LD,L).
```

Para o exemplo que considerámos atrás, a lista de nós visitados vem:

Travessia Prefixa: `[+,+,2,3,-,4,*,2,1]`
Travessia Posfixa: `[2,3,+,4,2,1,*,-,+]`
Travessia Infixa: `[2,+,3,+,4,-,2,*,1]`

Reparar que as travessias prefixa e posfixa dão-nos, respectivamente, a expressão

$$(2+3)+(4-(2*1))$$

na notação prefixa e posfixa.

Se representarmos expressões matemáticas (apenas com operadores binários) por meio de uma árvore binária como no nosso exemplo, podemos definir facilmente um predicado para calcular o valor numérico da expressão. Basta usar, por exemplo, a travessia posfixa e o algoritmo que vimos no capítulo anterior (veja-se o predicado `postfix/2` do Cap. 5).

6. Exercícios

Exame Final 1997/1998. Para árvores binárias cujos vértices contêm números, considere-se a seguinte definição:

> Uma árvore diz-se `boa` se para qualquer vértice, excepto as folhas, o número do vértice é igual à soma dos números nos dois vértices descendentes.

Usando esta definição, e a representação de árvores binárias que foi usada nas aulas teóricas e teórico-práticas, escreva um predicado `boa/1` que recebe como parâmetro uma árvore binária, e que toma o valor verdadeiro sse a árvore é boa. *Exemplos:*

```
?- boa(arv(7,vazio,vazio)).
   yes

?- boa(arv(5,arv(2,vazio,vazio),arv(2,vazio,vazio))).
   no

?-boa(arv(5,vazio,arv(5,arv(3,vazio,vazio),arv(2,vazio,vazio)))).
  yes
```

Possível resolução.

```
boa(vazio).
boa(arv(_,vazio,vazio)).
boa(arv(N,E,D)) :-
  boa(E),
  boa(D),
  vertice(E,NE),
  vertice(D,ND),
  N is NE + ND.

vertice(vazio,0).
vertice(arv(N,_,_),N).
```

Exame de Recorrência 1997/1998. Escreva um predicado `equivalente(A1,A2)` que permita relacionar duas representações de uma mesma árvore binária. A primeira representação é a dada nas aulas; a segunda representação é baseada em listas: uma árvore ou é vazia (símbolo `v`) ou é uma lista com 3 elementos —o conteúdo do vértice, a árvore esquerda e a árvore direita.
Exemplo:

```
?- equivalente(arv(2,arv(3,arv(4,v,v),v),arv(5,arv(6,v,v),arv(7,v,v))),L).
   L = [2,[3,[4,v,v],v],[5,[6,v,v],[7,v,v]]]
```

Possível resolução.

```
equivalente(v,v).
equivalente(arv(N,E,D),[N,EE,ED]) :-
  equivalente(E,EE),
  equivalente(D,ED).
```

Exame de Recurso 1997/1998. Recordando o que sabe sobre árvores binárias e seu manuseamento, responda às alíneas seguintes.

a): Defina o predicado `altura/2` que devolva no segundo argumento a altura da árvore binária, representada da forma dada nas aulas, passada no primeiro parâmetro do predicado. Assuma que a altura da árvore vazia é 0 e que a altura de uma árvore com apenas um vértice é 1.
Exemplos:

```
?- altura(arv(1,arv(2,v,v),v),N).
   N = 2
?- altura(arv(1,arv(2,v,v),arv(1,v,arv(8,v,arv(8,v,v)))),N).
   N = 4
```

b): Atente agora à seguinte definição:

> A árvore vazia é balanceada. Se não for vazia, uma árvore diz-se balanceada quando a altura da sua árvore esquerda não diferir da altura da sua árvore direita por mais de 1 unidade e as suas árvores esquerda e direita forem também balanceadas.

Implemente em `Prolog` o predicado `balanceada/1` que receba uma árvore binária e retorne `yes` caso a árvore binária seja balanceada e `no` na situação contrária.
Exemplos:

```
?- balanceada(arv(1,v,arv(2,arv(2,v,v),v))).
   no
?- balanceada(arv(1,v,arv(2,v,v))).
   yes
```

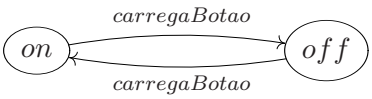
7
Autómatos Finitos

1. Introdução

Um autómato funciona como um reconhecedor de uma determinada linguagem e serve para modelar uma máquina ou, se quiserem, um computador simples. É usado, por exemplo, em editores de texto para reconhecer padrões.

Um conceito fundamental nos autómatos é o conceito de *estado*. Este conceito é, de facto, aplicado a qualquer sistema, por exemplo, à nossa televisão. As noções de estado e sistema são tão omnipresentes que foi desenvolvido uma campo de conhecimento chamado "Teoria dos Sistemas".

Uma televisão pode estar *ligada*(on) ou *desligada* (off). Temos então um sistema com dois estados que podemos representar pelo seguinte diagrama:



A um nível mais detalhado, podemos desejar diferenciar os canais, caso em que podemos ter centenas de estados: um para "desligada" e os restantes significando "ligada no canal i". De modo semelhante, uma máquina de lavar pratos pode estar num estado "desligada" ou "ligada num determinado programa".

Em qualquer dos exemplos acima, existe um número *finito* de estados. Neste capítulo estaremos sempre a "lidar com máquinas" com um número finito de estados.

Dada uma televisão, ela não está apenas num dos estados possíveis: somos capazes de fazer mudar a televisão de estado. Esta opção, tal como na figura acima, será indicada através de arcos dirigidos e suas etiquetas.

A nossa figura funciona assim como um *diagrama de estados* e como um *diagrama de transições*.

Uma sequência de acções `carregaBotao` faz com que a televisão acabe no estado `on`, se começarmos com ela ligada e carregarmos no botão um número par de vezes ou então se ela no início estiver desligada e houver um número ímpar de `carregaBotao`.

Se o *estado inicial* for o `off` e o *estado final* desejado for o `on`, então todas as sequências finitas de `carregaBotao` com comprimento ímpar têm como resultado o pretendido.

2. Autómatos Finitos Deterministas

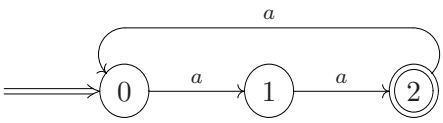
Um *autómato finito determinista* (AFD) é definido como um quíntuplo

$$AFD = (E, S, \delta, i, F)$$

onde E é o conjunto (finito) de *estados* admissíveis; S o conjunto dos símbolos de *entrada*; $\delta : E \times S \to E$ a *função de transição*; $i \in E$ o *estado inicial*; e $F \subseteq E$ o conjunto dos *estados finais* (não vazio).

No exemplo da televisão dado acima,
$E = \{on, off\}$;
$S = \{carregaBotao\}$;
$\delta\,(off, carregaBotao) = on$, $\delta\,(on, carregaBotao) = off$;
$i = off$; e $F = \{on\}$.

Consideremos agora um exemplo formal. Seja $AFD_1 = (E, S, \delta, i, F)$ definido por $E = \{0, 1, 2\}$; $S = \{a\}$; $\delta\,(0, a) = 1$, $\delta\,(1, a) = 2$, $\delta\,(2, a) = 0$; $i = 0$; e $F = \{2\}$. AFD_1 pode ser representado pelo seguinte diagrama:



onde convencionamos identificar o estado inicial com uma seta dupla e os estados finais com um círculo duplo.

Em Prolog basta-nos descrever este diagrama:

```
estadoInicial(afd1,0).
estadoFinal(afd1,2).
delta(afd1,0,a,1).
delta(afd1,1,a,2).
delta(afd1,2,a,0).
```

Nos dois exemplos que vimos, as funções de transição são ambas totais. Como função que é, δ pode, no entanto, ser parcial. Discutiremos isto mais à frente.

Os AFDs podem ser usados para ler *palavras* de entrada e terminar com a sua aceitação ou rejeição: eles são *reconhecedores*. Para compreender como os AFDs fazem este reconhecimento, convém olhá-los como sendo máquinas (computadores primitivos).

Seja (E, S, δ, i, F) um AFD e consideremos uma *fita de entrada* (ou *ficheiro de entrada*), uma cabeça de leitura e um *estado corrente*. A função de transição será o análogo de um *programa*; e o estado corrente o análogo da instrução corrente a ser executada. A fita de entrada contém uma palavra, um símbolo em cada célula da fita. Teremos tantas células na fita quantos os símbolos na palavra de entrada.

Dada uma palavra x inicializamos a "máquina" da seguinte maneira:

(i): Colocamos x na fita de entrada –um símbolo por célula.
(ii): A cabeça de leitura é posicionada na célula mais à esquerda (no primeiro símbolo da palavra de entrada).
(iii): Colocamos i como estado corrente.

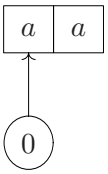
A máquina começa então a trabalhar. Como qualquer computador, ela tem um modo de funcionamento. Este computador funciona assim:

(i): O símbolo que se encontra na célula apontada pela cabeça de leitura é lido (*símbolo corrente*). Se a fita já tiver acabado a máquina pára (o que acontece quando já tivermos lido toda a palavra).
(ii): O *próximo estado* é calculo a partir do estado corrente e do símbolo corrente, usando a função de transição do autómato:

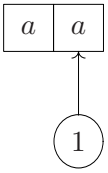
$$\delta\,(estadoCorrente, simboloCorrente) = proximoEstado$$

(iii): A cabeça de leitura move-se uma célula para a direita.
(iv): O *proximoEstado* torna-se no *estadoCorrente*, e torna-se ao passo (i).

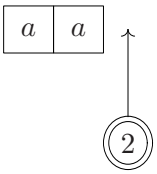
Para clarificar o que acabámos de dizer, voltemos ao autómato AFD_1 e consideremos, como input, a palavra aa. No início, $estadoCorrente = 0$ e cabeça de leitura está a apontar para o primeiro a:



Uma vez que $delta(0, a) = 1$, o próximo estado passa a ser 1 e a cabeça de leitura passa a apontar para o segundo a:



Atendendo que $\delta(1, a) = 2$ e que a palavra de entrada só possui dois símbolos, o autómato termina na seguinte situação:



Como o autómato terminou num estado final, dizemos que a palavra *aa* é *aceite* (ou *reconhecida*) por AFD_1. O conjunto de todas as palavras aceites por um AFD é designado por *linguagem aceite*, *linguagem definida* ou *linguagem reconhecida* por esse autómato.

As palavras *aa* e *aaaaa* são aceites por AFD_1 enquanto a palavra vazia, a palavra *a* e *aaaa* são rejeitadas. Não é muito difícil provar que a linguagem aceite por AFD_1 é dada pelo conjunto

$$\left\{aa(aaa)^i : i \geq 0\right\}$$

Em Prolog o processo de reconhecimento de palavras por um AFD, que acabámos de explicar, pode ser descrito da seguinte maneira:

```
aceita(AFD,Palavra) :-
  estadoInicial(AFD,I),
  reconhece(AFD,Palavra,I).       % estadoCorrente = estadoInicial

reconhece(AFD,[],EstCor) :-
  estadoFinal(AFD,EstCor).
reconhece(AFD,[S|R],EstCor) :-
  delta(AFD,EstCor,S,ProxEst), !, % reparar no cut
  reconhece(AFD,R,ProxEst).       % (delta 'e uma funcao...)
```

Com este programa poderemos perguntar ao interpretador de Prolog se uma palavra pertence, ou não, à *linguagem definida* por um determinado autómato. Por exemplo:

```
?- aceita(afd1,[a,a]).
   yes
?- aceita(afd1,[a,a,a,a,a]).
   yes
?- aceita(afd1,[]).
   no
?- aceita(afd1,[a]).
   no
?- aceita(afd1,[a,a,a,a]).
   no
```

Quando a função de transição é total, qualquer palavra de entrada será lida na totalidade antes do AFD parar. Se a função de transição não for total isso não é assim.

Esta classificação das funções de transição, conduz à correspondente classificação dos AFDs: se a função de transição de um AFD é total dizemos que o autómato é *completo* senão dizemos que ele é *incompleto*.

Não é muito difícil pensarmos num método que permita transformar qualquer AFD incompleto num completo, ao mesmo tempo que preservamos a sua linguagem. Seja $AFD = (E, S, \delta, i, F)$ um AFD incompleto, isto é, existe um par $(e, s) \in E \times S$ para o qual $\delta(e, s)$ não está definido. Vamos construir AFD', que é um AFD completo com a mesma linguagem de AFD, da seguinte maneira. Adicionamos um novo estado c a E e definimos δ' de tal modo que para todo o $e \in E$ e para todo o $s \in S$, $\delta'(e, s) = \delta(e, s)$ se $\delta(e, s)$ está definido e $\delta'(e, s) = c$ caso contrário. Para completar a função de transição δ', fazemos $\delta'(c, s) = c$ para todo o $s \in S$. $AFD' = (E \cup \{c\}, S, \delta', i, F)$ é um autómato com as características desejadas.

No nosso próximo exemplo vamos considerar uma linguagem na qual certas combinações de letras são proibidas. Pretendemos construir um AFD que reconheça essa linguagem.

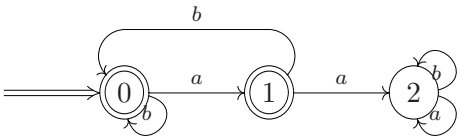
Consideramos $S = \{a, b\}$ e a linguagem L formada por todas as palavras com símbolos em S que não contêm dois a's consecutivos. Qualquer palavra que não contenha o símbolo a está trivialmente em L: a palavra vazia, b, bb, ... , b^i, ... A palavra a também está em L, mas as palavras

$$aa,\ aaa,\ a^i,\ \ldots$$

concerteza que não. Exemplos de mais palavras que estão em L são:

$$ababab,\ abb,\ abba,\ abbbababbabb$$

O autómato seguinte define a linguagem L.



3. AFDs com acções semânticas

Uma das limitações dos AFDs consiste na limitação do resultado do reconhecimento, formado apenas pela indicação *aceita/não aceita*.

Para tornar estes autómatos mais potentes, foram propostas extensões que permitissem a execução de acções arbitrárias durante o reconhecimento de frases de entrada. O mecanismo que vamos agora estudar é designado por *AFD reactivo* ou *AFD com acções semânticas*.

Um AFD reactivo é definido como um sêxtuplo

$$AFDR = (E, S, \delta, i, F, R)$$

onde E é o conjunto (finito) de *estados* admissíveis; S o conjunto dos símbolos de *entrada*; $\delta : E \times S \to E \times R$ a *função de transição*; $i \in E$ o *estado inicial*; $F \subseteq E$ o conjunto dos *estados finais* (não vazio); e R um conjunto de acções semânticas.

Os AFD reactivos podem, tal como vimos para os AFDs, ser representados graficamente por um grafo etiquetado. Só que agora uma transição $\delta(e1, s) = (e2, r)$ deve ler-se "o autómato transita do estado $e1$ para o estado $e2$ e executa a acção r, através do reconhecimento do símbolo s".

3.1. Simulação do funcionamento de uma máquina de café.

Vamos agora tentar modelar, por meio de um AFD reactivo, uma máquina de vender café. Vamos considerar que cada café custa 80$00; que se aceitam moedas de 10$00, 20$00 e 50$00; que a máquina é capaz de dar o troco; e por fim produzir um café saboroso. Outras moedas, que não as aceites, devem ser devolvidas.

Neste caso S é o conjunto das moedas; a palavra de entrada será uma sequência de moedas que chega à máquina; os estados corresponderão ao total de dinheiro inserido na máquina até ao momento. O estado inicial será então o estado 0$00 e os estados finais todos aqueles que correspondem a uma quantidade D de dinheiro igual ou superior ao preço do café: $D \geq 80\$00$. Quando se alcança um estado final D, serão despoletadas as seguintes duas acções (acções

semânticas): dar o troco $D - 80\$00$; preparar e dar o café. As acções semânticas são indicadas nos arcos do diagrama do autómato entre { }. O autómato `cafe` encontra-se descrito pelo seguinte programa Prolog: (não deixe de fazer o diagrama correspondente)

```
estadoInicial(cafe,0).

estadoFinal(cafe,D) :-
  D >= 80,
  actua(D).

 actua(80) :- fazCafe.
 actua(D)  :- daTroco(D), fazCafe.

 daTroco(D) :-
   T is D - 80,
   write('Troco = '), write(T), write('Esc.'), nl.

 fazCafe :- write('Espero que goste do cafe...').

delta(cafe,EC,10,PE) :- PE is EC + 10.
delta(cafe,EC,20,PE) :- PE is EC + 20.
delta(cafe,EC,50,PE) :- PE is EC + 50.
delta(cafe,EC,M,_) :- % M e' uma moeda na~o aceite pela m'aquina
  D is EC + M,        % D e' o dinheiro a devolver
  erro(D),!,          % devolve moedas
  fail.               % a sequencia de moedas na~o e' reconhecida

 erro(D) :-
   write('Fica com o teu dinheiro = '),
   write(D), write('Esc.'), nl.
```

Eis alguns exemplos de "utilização da máquina de café":

```
?- aceita(cafe,[50,50]).
   Troco = 20Esc.
   Espero que goste do cafe...
   Yes

?- aceita(cafe,[20,10,20,20,10]).
   Espero que goste do cafe...
   Yes

?- aceita(cafe,[20,100]).
   Fica com o teu dinheiro = 120Esc.
   No
```

3.2. Simulação do funcionamento de um multi-banco.

Pretende-se criar um modelo, muito simplificado, que descreva o funcionamento de um multi-banco. Tal modelo deve permitir uma interacção como a que se segue.

```
?- aceita(mb,[cartao,codigo,levantar,5,aceitar]).
   Retire os seus 5000$00
   Retire o talao
   Retire o seu cartao
   Volte sempre
   Yes
?- aceita(mb,[cartao,codigo,consultar,movimentos,aceitar]).
   Retire o talao de movimentos
   Retire o seu cartao
   Volte sempre
   Yes
```

```
?- aceita(mb,[cartao,codigo,consultar,interromper]).
   Retire o seu cartao
   Volte sempre
   Yes
?- aceita(mb,[cartao,codigo,consultar]).
   No
?- aceita(mb,[cartao,codigo,consultar,aceitar]).
   No
```

Eis o programa Prolog.

```
% simulacao do funcionamento de um multibanco por um automato reactivo
% versao simplificada: 19/Maio/1999

estadoInicial(mb,0).
estadoFinal(mb,0).

delta(mb,0,cartao,1).
delta(mb,1,codigo,2).
delta(mb,2,levantar,3).
delta(mb,3,15,4).       % 15 contos
delta(mb,4,aceitar,0) :-
  msg(15),
  msg.
delta(mb,3,10,5).       % 10 contos
delta(mb,5,aceitar,0) :-
  msg(10),
  msg.
delta(mb,3,5,6).        %  5 contos
delta(mb,6,aceitar,0) :-
  msg(5),
  msg.
delta(mb,2,consultar,7).
delta(mb,7,saldo,8).
delta(mb,8,aceitar,0) :-
  write('Retire o talao'), nl,
  msg.
delta(mb,7,movimentos,9).
delta(mb,9,aceitar,0) :-
  write('Retire o talao de movimentos'), nl,
  msg.
delta(mb,_,interromper,0) :- % em qualquer estado,
  msg.                       % podemos sempre cancelar

msg(D) :-
  write('Retire os seus '),
  write(D), write('000$00'), nl,
  write('Retire o talao'), nl.
msg :-
  write('Retire o seu cartao'), nl,
  write('Volte sempre'), nl.
```

4. Autómatos Finitos Não-Deterministas

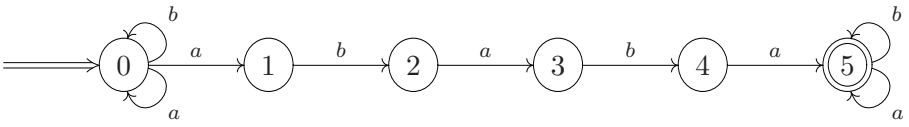
Vamos agora introduzir a noção de *escolha*, isto é, introduzir *não-determinismo*. A definição de *AFND* (*Autómato Finito Não-Determinista*) coincide com a do AFD com a única excepção que δ é agora encarada como uma *relação* (em vez de uma função):

$$\delta \subseteq E \times S \times E.$$

Significa então que dado um estado e um símbolo poderemos ter vários estados possíveis para onde ir. Isto corresponde, se quiserem, à noção de *universos múltiplos* que aparece nas histórias de ficção científica ou na interpretação dos *"muitos mundos"* da mecânica quântica. Claro que muitos argumentam que o universo é um sistema determinista, não obstante complexo, e por conseguinte cada acção ou "escolha" que fazemos é pré-determinada ou pré-destinada. Contudo, pelo que me toca, gosto sempre de pensar que temos escolhas e liberdade de escolha ...

Mas a verdade é que podemos sempre construir um AFD, a partir de um AFND dado, que aceite exactamente a mesma linguagem. Trata-se de um teorema importante que não é muito difícil de perceber se pensarmos que estamos sempre a lidar com um número finito de estados. Não obstante esta equivalência, muitas vezes é muito mais fácil definir um AFND do que um determinista equivalente. Vejamos um exemplo.

Uma tarefa típica de um editor de texto é procurar uma dada sub palavra ou padrão no texto. Vamos assumir, por simplicidade, que o texto é constituído por palavras formadas apenas por a's e b's; e que o padrão que andamos à procura é *ababa*. O AFND



aceita todos os textos que contêm a sub palavra *ababa*. Definir um AFD que reconheça a mesma linguagem é talvez um pouco mais complicado (deve-o fazer, no entanto, como exercício –uma versão não determinista e uma versão determinista).

Havendo várias transições possíveis para o mesmo símbolo de entrada, como é que sabemos para onde ir? Uma solução consiste em seguir por um dos estados e, em caso de falha, efectuar um retrocesso ("backtracking"): o Prolog irá voltar ao ponto de ramificação e "remover" os símbolos aceites desde essa ramificação.

O AFND que construímos é descrito em Prolog como se segue:

```
% definicao do AFND ababa
estadoInicial(ababa,0).
estadoFinal(ababa,5).
delta(ababa,0,a,0).
delta(ababa,0,a,1).
delta(ababa,0,b,0).
delta(ababa,1,b,2).
delta(ababa,2,a,3).
delta(ababa,3,b,4).
delta(ababa,4,a,5).
delta(ababa,5,a,5).
delta(ababa,5,b,5).
```

Temos agora que alterar o nosso reconhecedor de palavras para passar a funcionar com AFNDs. A versão que se segue funciona quer com autómatos deterministas quer com autómatos não deterministas (a única alteração foi a remoção do cut).

```
aceita(AF,Palavra) :-
  estadoInicial(AF,I),
  reconhece(AF,Palavra,I).        % estadoCorrente = estadoInicial

reconhece(AF,[],EstCor) :-
  estadoFinal(AF,EstCor).
reconhece(AF,[S|R],EstCor) :-
  delta(AF,EstCor,S,ProxEst),
  reconhece(AF,R,ProxEst).
```

Com estas definições obtemos:

```
?- aceita(ababa,[b,a,b,a,b,b,a,b,a,b,b,a,b,a,b,b,a]).
   no
?- aceita(ababa,[b,a,b,a,b,b,a,b,a,b,a,b,a,b,a,b,b,a]).
   yes
```

5. Autómatos Finitos com transições vazias

Nos autómatos que vimos até agora, exigimos, em cada passo, que a cabeça de leitura se mova uma célula para a direita. Vamos agora relaxar esta condição permitindo que a cabeça permaneça na mesma célula durante uma transição. A estas transições, às quais não correspondem nenhum símbolo lido, chamamos *transições vazias*. Num AFND com transições vazias temos

$$\delta \subseteq E \times (S \cup \{\ \}) \times E.$$

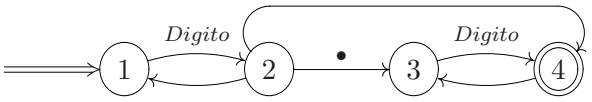
Para que o nosso programa Prolog aceite estes autómatos, basta acrescentar uma terceira cláusula `reconhece/3`:

```
reconhece(AF,T,EstCor) :-
  transicaoVazia(AF,EstCor,ProxEst),
  reconhece(AF,T,ProxEst).
```

O poder expressivo de um AFND com transições vazias não é maior que um AFND (que por sua vez é igual, como dissemos, a um AFD). Existem situações, no entanto, em que as transições vazias são úteis. Vejamos alguns exemplos.

Dados dois autómatos AF_1 e AF_2, pretende-se construir um terceiro autómato AF_{tv} que defina a linguagem dada pela união da linguagem reconhecida por AF_1 com a linguagem reconhecida por AF_2. O autómato AF_{tv} pretendido pode ser definido com um novo estado inicial que, via transições vazias, conduz aos estados iniciais de AF_1 e AF_2. Com esta construção vê-se facilmente que se x é uma palavra aceite por AF_1, então x é aceite por AF_{tv} e que se x é aceite por AF_2, então é aceite por AF_3. O inverso também é verdade: se x é aceite por AF_3 é aceite por AF_1 ou por AF_2.

Vejamos, para terminar, um autómato finito com transições vazias que define um número decimal sem sinal



e a respectiva definição em Prolog

```
digitos([0,1,2,3,4,5,6,7,8,9]).

estadoInicial(af_tv,1).
estadoFinal(af_tv,4).

delta(af_tv,1,D,2) :- digitos(L), member(D,L).
delta(af_tv,2,.,3).
delta(af_tv,3,D,4) :- digitos(L), member(D,L).

transicaoVazia(af_tv,2,1).
transicaoVazia(af_tv,2,4).
transicaoVazia(af_tv,4,3).
```

que nos permite fazer perguntas como as que se seguem.

```
?- aceita(af_tv,[1,1,.,2,3]).  % 11.23
   yes
```

```
?- aceita(af_tv,[.,2,3]).  % .23
   no

?- aceita(af_tv,[2,3]).  % 23
   yes
```

6. Exercícios

Exercício do Exame Final de 1998. Os modernos telefones prestam já serviços diversos, para além da simples ligação a outro telefone. Esses serviços atingem-se começando por digitar o símbolo "*", ou "#", seguido do código do serviço.
Supondo que de momento o operador telefónico nacional disponibiliza:

- a ligação a outro posto, quando for escrito um número telefónico (começa e só contém dígitos);
- informações referentes aos resultados do totoloto, através do código "*10";
- informações referentes ao gravador de chamadas, usando o código "*20"seguido de 3 dígitos que correspondem à senha;
- teste do sinal de toque, usando o símbolo "#"

pretende-se que modele o funcionamento do telefone usando um Autómato Determinista Reactivo.
Represente o autómato em causa numa BC em Prolog.

Exercício do Exame de Recurso de 1998. Pretende-se construir um programa que seja capaz de ler palavras (sequências de letras separadas por 1 ou mais espaços, ou new-line (NL)) e reconhecer conceitos, que podem ser expressos por palavras diferentes.
Considere-se por exemplo a seguinte lista (reduzida):

```
giro = lindo = belo --> conceito(bonito)
bera = fraco        --> conceito(mau)
ricaco = riquinho   --> conceito(rico)
```

Uma forma fácil e eficiente de implementar tal programa é concebê-lo como um ciclo standard guiado por um autómato determinista, que vá lendo caracteres e transitando de estado por cada caracter lido, até chegar ao fim duma palavra. Assim sendo, pede-se-lhe que desenhe o referido autómato determinista para reconhecer os conceitos e as palavras indicadas acima (todas as outras palavras devem ser rejeitadas, como erro).
Represente o autómato em causa numa BC em Prolog.

8
Gramáticas

Muita da investigação feita na construção de sistemas com processamento de língua natural, permitiu concluir que a inferência lógica é um meio poderoso pela qual uma linguagem pode ser analisada e gerada. A lógica das cláusulas definidas — um subconjunto da lógica na qual o Prolog é baseado — provou ser muito útil na construção de analisadores e geradores lexicais. Neste capítulo descrevemos o formalismo das gramáticas em Prolog: as *gramáticas de cláusulas definidas*.

1. A estrutura de uma linguagem

As gramáticas de cláusulas definidas — *DCG's*, do inglês "Definite Clause Grammar's" — constituem uma classe de gramáticas, úteis na construção de sistemas de língua natural. Uma gramática é definida como um quádruplo

$$G = (N, T, R, i)$$

onde N é o conjunto dos *símbolos não terminais* da gramática; T o conjunto dos *símbolos terminais* (N e T são disjuntos); R o conjunto das *regras de produção (ou derivação)*; e $i \in N$ o *símbolo inicial* da gramática.

As gramáticas são particularmente úteis por dois motivos. Por um lado elas servem para definir/descrever uma linguagem: a linguagem formada por todas as sucessões de símbolos terminais que podem ser obtidas por aplicação das regras de produção da gramática; por outro dão-nos um processo de análise/reconhecimento dessa mesma linguagem.

De maneira a compreendermos, completamente, uma linguagem, seja ela uma linguagem natural ou uma linguagem de programação, temos de compreender a sua gramática. As frases de uma linguagem são formadas por sequências de *palavras*, palavras essas ordenadas de acordo com regras específicas. Essas regras fiscalizam o acordo verbal, de género, voz, etc.

2. As gramáticas independentes do contexto

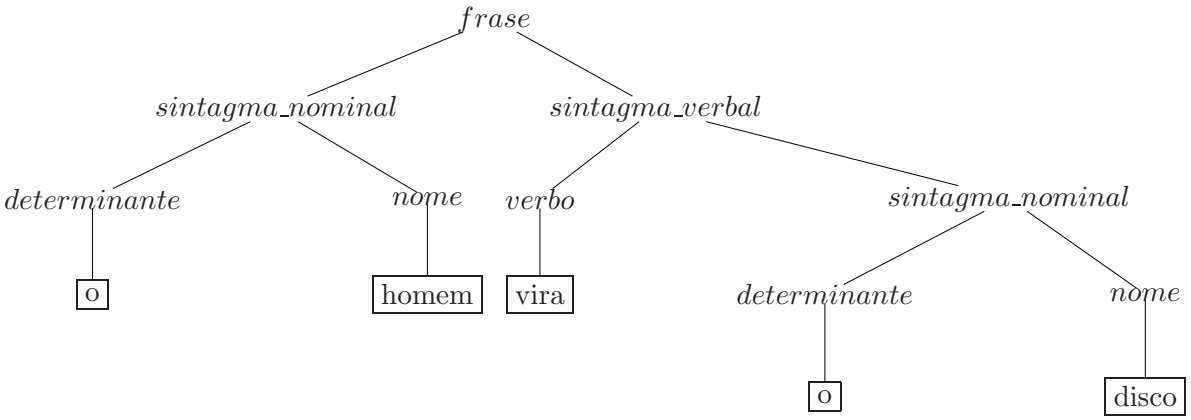
Existem muitas classes de gramáticas. Cada uma delas tem a sua própria notação. Uma classe familiar de gramáticas são as *gramáticas independentes do contexto*. As linguagens de programação são muitas vezes especificadas com estas gramáticas, geralmente usando a notação BNF (Backus-Naur Form).

O seguinte exemplo mostra uma gramática independente do contexto, na forma BNF, que aceita a frase "o homem vira o disco":

```
<frase> ::= <sintagma_nominal> <sintagma_verbal>
<sintagma_nominal> ::= <determinante> <nome>
<sintagma_verbal> ::= <verbo> <sintagma_nominal>
<determinante> ::= o
<nome> ::= homem | disco
<verbo> ::= vira
```

As palavras que podem aparecer nas frases são os símbolos *terminais* da gramática. Os outros símbolos, que aparecem nas regras de produção da gramática entre < e >, são os símbolos *não terminais*. Para o exemplo acima, os símbolos terminais são: `o`, `homem`, `vira`, e `disco`; enquanto os símbolos não terminais são: `frase`, `sintagma_nominal`, `sintagma_verbal`, `determinante`, `nome` e `verbo`.

Uma *arvore de derivação@árvore de derivação* é uma representação gráfica, em forma de árvore, da sequência de derivação de uma frase a partir de uma gramática independente do contexto. Essa árvore mostra a estrutura da frase e as relações hierárquicas entre as várias frases possíveis. A um programa que constrói árvores de derivação chamamos *analisador sintáctico* ("parser"). A seguinte figura ilustra uma árvore de derivação para uma frase simples, como "o homem vira o disco".



Numa árvore de derivação os nós interiores estão associados a símbolos não terminais e as folhas estão associadas a símbolos terminais.

3. As gramáticas de cláusulas definidas

As DCGs constituem uma classe de gramáticas mais poderosas do que as independentes de contexto: os símbolos não terminais podem ter argumentos! O problema de analisar sintacticamente uma string irá ser encarado como o problema de provar que um teorema é conclusão lógica da linguagem definida pelos axiomas das cláusulas definidas (pelas regras de produção da gramática).

Na notação das DCGs, isto é, em Prolog, se quisermos dizer que uma frase *pode tomar a forma de* um sintagma nominal *seguido de* um sintagma verbal, introduzimos a seguinte regra:

```
frase --> sintagma_nominal, sintagma_verbal.
```

Nas regras das gramáticas de cláusulas definidas, lê-mos o símbolo `->` como "pode tomar a forma de" e a vírgula como "seguido de".

A DCG que se segue, gera uma linguagem que tem como frase particular "o homem vira o disco".

```
frase --> sintagma_nominal, sintagma_verbal.
sintagma_nominal --> determinante, nome.
sintagma_verbal --> verbo, sintagma_nominal.
determinante --> [o].
nome --> [homem].
nome --> [disco].
verbo --> [vira].
```

4. As DCGs como analisadores sintácticos de uma língua natural

Muitas linguagens de programação não são apropriadas para transformar gramáticas em analisadores sintácticos de língua natural. O Prolog, contudo, é uma linguagem de programação que se presta a isso. Os programas escritos na notação DCG são automaticamente convertidos em cláusulas Prolog quando a base de conhecimento é consultada. O interpretador de Prolog traduz cada regra DCG numa cláusula que ele pode compreender. O operador `-->` é transformado em `:-`. Uma solução seria transformar uma regra da DCG como

```
frase --> sintagma_nominal, sintagma_verbal.
```

na cláusula

```
frase(F) :- sintagma_nominal(SN),
            sintagma_verbal(SV),
            append(SN,SV,F).
```

É esta de facto a ideia mas, por razões de eficiência, os interpretadores não a implementam assim. Os símbolos não terminais são transformados em predicados com *dois* argumentos adicionais. O primeiro argumento irá corresponder à lista de palavras de entrada e o argumento final à lista com os elementos que restam depois da chamada ao predicado ser feita. Assim a regra da DCG acima será transformada internamente na cláusula

```
frase(A,B) :-
  sintagma_nominal(A,C), sintagma_verbal(C,B).
```

A DCG acima (que aceita a frase "o homem vira o disco") é equivalente ao seguinte programa Prolog:

```
frase(L1,L3) :-
  sintagma_nominal(L1,L2), sintagma_verbal(L2,L3).
sintagma_nominal(L1,L3) :-
  determinante(L1,L2), nome(L2,L3).
sintagma_verbal(L1,L3) :-
  verbo(L1,L2), sintagma_nominal(L2,L3).
determinante([o|R],R).
nome([homem|R],R).
nome([disco|R],R).
verbo([vira|R],R).
```

4.1. A sintaxe das DCGs.

A sintaxe das DCGs é semelhante à sintaxe do Prolog que já conhecemos. Listamos de seguida as regras específicas (ver exemplos à frente):

- Os símbolos não terminais são escritos como qualquer átomo em Prolog.
- Para distinguir os símbolos terminais dos não terminais, uma sequência de símbolos terminais é escrita como uma lista. A lista vazia `[]` corresponde à sequência vazia.
- A parte esquerda das regras de uma gramática (antes de `-->`) contêm apenas símbolos não terminais.
- A parte direita das regras (a seguir a `-->`) podem conter símbolos não terminais e símbolos terminais.
- As alternativas podem ser explicitadas directamente na parte direita de uma regra da gramática, usando o operador de disjunção `;` ou então a barra vertical `|`.
- Podem ser incluídas acções semânticas na parte direita de uma regra. Essas acções semânticas são colocados entre { }. O processo de conversão automático acima referido não altera o que estiver no interior destes parêntesis.

- O cut poder ser incluído na parte direita de uma regra da gramática. Não é necessário colocar o cut no interior de { }.

4.2. Invocar uma DCG.

Depois de construída uma DCG e consultada a base de conhecimento correspondente, podemos invocá-la através do predicado correspondente ao símbolo não terminal inicial da gramática. Para a DCG que construímos podemos chamar o predicado `frase` da seguinte maneira:

```
?- frase([o,homem,vira,o,disco],[]).
   yes
```

4.3. Acrescentar argumentos.

Podemos adicionar um qualquer número de argumentos aos símbolos não terminais que estamos a definir numa DCG. Podemos querer acrescentar argumentos, por exemplo, para conseguirmos acordo de número. Caso contrário o analisador sintáctico pode terminar com sucesso com uma frase gramaticalmente incorrecta. Por exemplo todas as frases seguintes estão gramaticalmente incorrectas:

```
Os homem vira o disco
O homem vira os disco
Os homens vira o disco
```

No exemplo que se segue vamos adicionar o argumento `Numero`, que pode estar instanciado quer com `singular` quer com `plural`, de modo a sermos capazes de garantir acordo de número. Adicionamos também um outro argumento que nos permitirá distinguir pessoas de coisas. Deste modo, seremos capazes de diferenciar entre o `nome homem` (pessoa) e o `nome disco` (coisa) e fazer com que a frase "o homem vira o disco" seja aceite enquanto que a frase"o disco vira o homem" não ...

```
frase -->
  sintagma_nominal(Numero,pessoa),
  sintagma_verbal(Numero).
sintagma_nominal(Numero,Tipo) -->
  determinante(Numero),
  nome(Numero,Tipo).
sintagma_verbal(Numero) -->
  verbo(Numero),
  sintagma_nominal(Numero,coisa).
determinante(singular) --> [o].
determinante(plural) --> [os].
nome(singular,pessoa) --> [homem].
nome(plural,pessoa) --> [homens].
nome(singular,coisa) --> [disco].
verbo(singular) --> [vira].
verbo(singular) --> [viram].
```

Também é útil acrescentar argumentos quando pretendemos ver o resultado de um cálculo que ocorre durante o processo de análise sintáctica. Vejamos um exemplo.

Queremos uma gramática que sirva para definir expressões aritméticas simples (envolvendo números de 0 a 9) e que calcule o valor das expressões. Considere-se

```
-2+3*5+1
```

como um exemplo de uma expressão aritmética. No nosso programa lógico `Z` é o argumento adicional:

```
expr(Z) --> termo(X), [+], expr(Y), {Z is X + Y}.
expr(Z) --> termo(X), [-], expr(Y), {Z is X - Y}.
expr(Z) --> termo(Z).

termo(Z) --> numero(X), [*], termo(Y), {Z is X * Y}.
termo(Z) --> numero(X), [/], termo(Y), {Z is X / Y}.
termo(Z) --> numero(Z).
```

```
numero(N) --> [+], numero(N).
numero(N) --> [-], numero(X), {N is 0 - X}.
numero(N) --> [N], {number(N), 0 =< N, N =< 9}.
```

Uma possível execução desta gramática será:

```
?- expr(Z,[-,2,+,3,*,5,+,1],[]).
   Z = 14
```

Vamos agora imaginar que estamos interessados em saber, dada uma frase de entrada, qual a árvore de derivação que representa a estrutura de tal frase. Para isso vamos usar argumentos extra ... De maneira a devolvermos a árvore de derivação, vamos adicionar um argumento extra a cada predicado. Este argumento indica que estamos interessados em ver a árvore de derivação construída a partir das árvores das sub frases. Se o interpretador de Prolog poder encontrar um sintagma nominal seguido por um sintagma verbal, as duas árvores de derivação serão combinadas de modo a formar uma árvore de derivação maior, que usa o nome de predicado `frase`, e que constituirá a árvore da frase completa. Na DCG que se segue `Num` representa o argumento para o acordo de número; enquanto `SN`, `SV`, ... representam os argumentos para a árvore de derivação.

```
frase(frase(SN,SV)) -->
  sintagma_nominal(Num,SN),
  sintagma_verbal(Num,SV).


sintagma_nominal(Num,sintagma_nominal(D,N)) -->
  determinante(Num,D),
  nome(Num,N).
sintagma_verbal(Num,sintagma_verbal(V,SN)) -->
  verbo(Num,V),
  sintagma_nominal(Num,SN).
determinante(singular,determinante(o)) --> [o].
determinante(singular,determinante(a)) --> [a].
determinante(singular,determinante(um)) --> [um].
determinante(plural,determinante(os)) --> [os].
nome(singular,nome(homem)) --> [homem].
nome(singular,nome(mulher)) --> [mulher].
nome(plural,nome(homens)) --> [homens].
nome(singular,nome(disco)) --> [disco].
nome(singular,nome(livro)) --> [livro].
verbo(singular,verbo(vira)) --> [vira].
verbo(singular,verbo(viram)) --> [viram].
verbo(singular,verbo(escreve)) --> [escreve].
```

O que se segue é a árvore de derivação (escrita "toda bonitinha") para a frase "a mulher escreve um livro":

```
?- frase(A,[a,mulher,escreve,um,livro],[]).
   A = frase(
          sintagma_nominal(
                     determinante(a),
                     nome(mulher)
          ),
          sintagma_verbal(
                     verbo(escreve),
                     sintagma_nominal(
                        determinante(um),
                        nome(livro)
                     )
          )
       )
```

5. Mais exemplos

5.1. DCG "email".

Pretendemos construir uma gramática que defina o que é uma mensagem de correio electrónico (vulgo email). Vamos começar com uma versão muito simples e, incrementalmente, acrescentando "novas funcionalidades".

```
%DCG 'email'. Versao 1

email      --> cabec, corpo.
cabec      --> [from], [:], [nome], dest.
dest       --> [to], [nome], [cc], listanomes.
listanomes --> [].
listanomes --> [nome], listanomes.
corpo      -->[texto].
```

Esta versão permite reconhecer, como emails válidos, mensagens do género:

```
?- email([from,:,nome,to,nome,cc,nome,texto],[]).
   yes
```

Acrescentemos agora um argumento que permita enviar mensagens de conteúdo arbitrário (e não apenas mensagens com conteúdo `texto`):

```
%DCG 'email'. Versao 2

email      --> cabec, corpo(TXT).
cabec      --> [from], [:], [nome], dest.
dest       --> [to], [nome], [cc], listanomes.
listanomes --> [].
listanomes --> [nome], listanomes.
corpo(TXT) --> [TXT].
```

que nos permitirá reconhecer mensagens como

```
?- email([from,:,nome,to,nome,cc,'Ola'],[]).
   yes
```

Vamos agora imaginar que, dada uma mensagem, estamos não só interessados em saber se ela é, ou não, válida, mas também, no caso de válida, saber a quantas pessoas ela se destina. Para isso vamos usar mais um argumento (`Conta`):

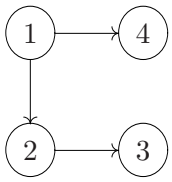
```
% DCG 'email'. Versao 3

email(Conta)   --> cabec(Conta), corpo(TXT).
cabec(Conta)   --> [from], [:], [nome], dest(Conta).
dest(Conta)    -->
  [to], [nome], [cc], listanomes(C), {Conta is C+1}.
listanomes(0)  --> [].
listanomes(N)  --> [nome], listanomes(N1), {N is N1+1}.
corpo(TXT)     --> [TXT].
```

Podemos agora 'enviar um email a 4 amigos':

```
?- email(C,[from,:,nome,to,nome,cc,nome,nome,nome,'ola'],[]).
   C = 4
   yes
```

5.2. DCG "grafo".

Pretendemos construir um programa em Prolog, usando uma DCG, que permita descrever um grafo, dirigido não pesado, por meio de frases em português. Desta maneira, não teremos de introduzir os factos `arco` correspondentes directamente na Base de Conhecimento (BC). Por exemplo, se quisermos descrever o grafo

G1 

dizemos "grafo g1: 1 ligado a 2, 4; 2 ligado a 3." e o programa encarregar-se-á de acrescentar à BC os factos

```
arco(g1,1,2).
arco(g1,1,4).
arco(g1,2,3).
```

Segue-se uma solução.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Gramatica 'Grafo'
%%
%% Implementacao por Delfim F. Marado Torres
%% 25/Maio/1999
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gramGrafo -->
  nome(G),
  listaVertices(G).

nome(NomeGrafo) -->
  [grafo, NomeGrafo, :].

listaVertices(G) -->
  vertices(G),
  outros(G).

vertices(G) -->
  [N, ligado, a],
  lista(G,N).



lista(G,N1) -->
  [N2],
  { acrescentaBC(G,N1,N2) },
  resto(G,N1).

resto(_,_) -->
  [].
resto(G,N) -->
  [','],
  lista(G,N).

outros(_) -->
  ['.'].
outros(G) -->
  [';'],
  listaVertices(G).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Accoes Semanticas da Gramatica
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
acrescentaBC(G,N1,N2) :-
  assertz(arco(G,N1,N2)),
  nl,
  write('A clausula '),
  write(arco(G,N1,N2)),
  write(' foi adicionada com sucesso.'),
  nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicado Principal: grafo/1
%% Recebe uma string em vez de uma lista.
%%
%% EXEMPLO:
%%   ?- grafo('grafo g1: 1 ligado a 2,4; 2 ligado a 3.').
%%
%%   A clausula arco(g1,1,2) foi adicionada com sucesso.
%%
%%   A clausula arco(g1,1,4) foi adicionada com sucesso.
%%
%%   A clausula arco(g1,2,3) foi adicionada com sucesso.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

grafo(String) :-
  converte(String,Lista),!,
  gramGrafo(Lista,[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Transforma uma frase em lingua natural,
%% dada como uma string de palavras separadas
%% por espacos, numa lista cujos elementos sao
%% as palavras.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

converte(S,L) :-
  name(S,LA1),
  adicionaEspacos(LA1,LA2),
  conv(LA2,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Os caracteres ':' ';' ',' '.' '!' '?' podem
% estar escritos juntos a uma palavra, mas contam
% como uma palavra separada... (simbolos terminais
% na gramatica).
%
% Desta maneira, escrever
%    'grafo g1: 1 ligado a 2,4; 2 ligado a 3.'
% sera equivalente a escrever
%    'grafo g1 : 1 ligado a 2 , 4 ; 2 ligado a 3 .'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 32 = codigo ASCII do espaco
%
% 58 = codigo ASCII ':'
% 59 = codigo ASCII ';'
% 44 = codigo ASCII ','
% 46 = codigo ASCII '.'
```

```
% 33 = codigo ASCII '!'
% 63 = codigo ASCII '?'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

adicionaEspacos([],[]).
adicionaEspacos([X|R1],[32,X,32|R2]) :-
  member(X,[58,59,44,46,33,63]),
  adicionaEspacos(R1,R2).
adicionaEspacos([X|R1],[X|R2]) :-
  adicionaEspacos(R1,R2).

conv([],[]).
conv(LA,[P|R]) :-
  palavra(LA,PA,RA), name(P,PA), conv(RA,R).

palavra([],[],[]).
palavra([32|R],[],L) :-
  desprezaEspacos(R,L).
palavra([X|R1],[X|R2],L) :-
  palavra(R1,R2,L).

desprezaEspacos([32|R1],R) :-
  desprezaEspacos(R1,R).
desprezaEspacos(L,L).
```

5.3. DCG "Pascal".

Damos agora um 'exemplo real', mais complexo. O objectivo é construir uma gramática que defina/descreva uma linguagem de programação do tipo Pascal. De seguida encontra a DCG juntamente com alguns exemplos de programas válidos (`teste1`, ... , `teste4`).

```
teste1(P) :-
  parse([program,teste1,';',begin,write,x,'+','y','-',z,'/',2,end],P), !.

teste2(P) :-
  parse(
   [program,teste2,';',begin,if,a,'>',b,then,max,':=',a,else,max,':=',b,end],
   P
  ), !.

teste3(P) :-
  parse(
    [program,repeat,';',begin,i,':=',1,';',repeat,begin,write,i,';',i,':=',
     i,'+',1,end,until,i,'=',11,end],P), !.

teste4(P) :-
  parse([program,ciclofor,';',begin,for,i,'=',1,to,11,step,1,do,write,i,end],
        P), !.

% Parser

parse(Source,Structure) :-
  pl_program(Structure,Source,[]).

pl_program(S) -->
  [program],
  identifier(X),
  [';'],
  statement(S).
```

```
statement((S;Ss)) -->
  [begin],
  statement(S),
  rest_statements(Ss).
statement(assign(X,V)) -->
  identifier(X),
  [':='],
  expression(V).
statement(if(T,S1,S2)) -->
  [if],
  test(T),
  [then],
  statement(S1),
  [else],
  statement(S2).
statement(while(T,S)) -->
  [while],
  test(T),
  [do],
  statement(S).

statement(repeat(S,T)) -->
  [repeat],
  statement(S),
  [until],
  test(T).

statement(for(ID,C1,C2,C3,S)) -->
  [for],
  identifier(ID),
  ['='],
  expression(C1),
  [to],
  expression(C2),
  [step],
  expression(C3),
  [do],
  statement(S).

statement(read(X)) -->
  [read],
  identifier(X).
statement(write(X)) -->
  [write],
  expression(X).

rest_statements((S;Ss)) -->
  [';'],
  statement(S),
  rest_statements(Ss).
rest_statements(void) -->
  [end].

expression(X) -->
  pl_constant(X).
```

```
expression(expr(Op,X,Y)) -->
  pl_constant(X),
  arithmetic_op(Op),
  expression(Y).

arithmetic_op('+') -->
  ['+'].
arithmetic_op('-') -->
  ['-'].
aritmetic_op('*') -->
  ['*'].
arithmetic_op('/') -->
  ['/'].

pl_constant(name(X)) -->
  identifier(X).
pl_constant(number(X)) -->
  pl_integer(X).

identifier(X) -->
  [X],
  {atom(X)}.

pl_integer(X) -->
  [X],
  {integer(X)}.

test(compare(Op,X,Y)) -->
  expression(X),
  comparison_op(Op),
  expression(Y).

comparison_op('=') -->
  ['='].
comparison_op('<>') -->
  ['<>'].
comparison_op('>') -->
  ['>'].
comparison_op('<') -->
  ['<'].
comparison_op('>=') -->
  ['>='].
comparison_op('<=') -->
  ['<='].
```

6. Exercícios

Exercício do Exame Final de 1997. Analise atentamente a seguinte gramática escrita na notação lógica DCG

```
frase     -->  exp, [';'].
exp       -->  equac.
exp       -->  interroga.
equac     -->  expressao, ['='], expressao.
interroga -->  ['?'], ['='], expressao.
expressao -->  operando, resto.
operando  -->  [num(N)].
operando  -->  ['('], expressao, [')'].
resto     -->  operador, expressao.
```

```
resto     -->  [].
operador  -->  ['+']  |  ['-']  |  ['/']  |  ['*'].
```

Responda, então, às alíneas seguintes:

a): Diga qual a resposta de um Interpretador de Prolog (IP), à questão

```
frase( [num(5),'+',num(3),'=',num(8),';'], [] ).
```

b): Dê 1 exemplo de uma frase válida da linguagem definida pela DCG acima.

c): Modifique a gramática supra de modo a permitir que cada frase tenha mais do que uma equação ou interrogação (mais do que uma `exp`).

Exercício do Exame de Recorrência de 1997. Considere as frases seguintes, que são estruturalmente (isto é, sintacticamente) idênticas:

- O Delfim escreveu um programa com um computador
- O Pedro encontrou um colega com um problema
- A Joana comeu uma maçã com os dentes

e responda às questões:

a): defina um analisador sintáctico, usando o formalismo DCG do Prolog, que reconheça essas frases e outras com a mesma estrutura.

b): Modifique a sua gramática (DCG) de modo a aceitar também frases do tipo
- um homem disse um poema

c): Acrescente à sua gramática (DCG) os argumentos e acções necessários de modo a validar a *concordância*, em *género* e em *número*, entre os determinantes e os nomes.

d): Diga que questão (átomo lógico) deve colocar ao Interpretador de Prolog para fazer o reconhecimento da 1ª frase acima com a DCG que desenvolveu (suponha que a DCG já foi consultada (carregada para a memória do Interpretador)).

Exercício do Exame de Recurso de 1997. Relativamente a uma Central de Reservas, desenhou-se a gramática abaixo (apresentada na notação lógica DCG) para definir uma linguagem que permita descrever viaturas.

```
viaturas  -->  marca, modelo, matriculas.
marca     -->  [pal(M)].
modelo    -->  [pal(M)], cilindrada, lugares.
cilindrada-->  numero.
lugares   -->  numero.
numero    -->  [num(N)].
matriculas-->  matricula, outras.
outras    -->  [','], matriculas.
outras    -->  [].
matricula -->  [mat(X)].
```

Responda, então, às alíneas seguintes:

a): Diga qual a resposta de um Interpretador de Prolog (IP), à questões:

a1):

```
?- viaturas([pal(ferrari),pal(f40),num(300),num(8),
             mat(45-66-AX),mat(66-68-EI)],[]).
```

a2):

```
?- viaturas([pal('Mercedes'),pal(c220D),num(2155),num(5)],[]).
```

b): Modifique a gramática supra de modo a: permitir que cada frase contenha a descrição de viaturas de marcas e/ou modelos diferentes; e que não seja obrigatório escrever o número de lugares.

c): Acrescente à DCG supra as *acções semânticas* e os *argumentos* que achar necessários de modo a:

c1): validar a cilindrada que terá de ser um número superior, ou igual, a 1000 e validar o número de lugares que deverá estar no intervalo [2,10[;

c2): validar a marca do automóvel que terá de ser uma de entre as marcas existentes numa lista que se supõe existir na BC na forma do predicado `marcas([peugeot, fiat, ferrari, mercedes, ...])`.

Exercício do Exame Final de 1998. Pretende-se criar uma linguagem que permita identificar um Arqueólogo (indicando o *código, nome e instituição* e indicar todos os Objectos (referidos pelo respectivo *identificador*) por ele encontrados numa determinada escavação (que também terá de ser identificada).
O que se lhe pede é que escreva uma DCG para definir a dita linguagem, começando por dar exemplos de frases válidas da linguagem que vai desenvolver.
Enriqueça posteriormente a sua DCG com atributos (argumentos dos símbolos gramaticais) para retirar de cada frase, reconhecida pelo programa derivado da DCG, os dados relativos ao arqueólogo descrito.

Exercício do Exame de Recorrência de 1998. Analise atentamente a seguinte gramática escrita na notação lógica DCG

```
comando   -->  operSimp, args, [';'].
comando   -->  operComp, [de], arg, [para], arg, [';'].
args      -->  arg, resto.
resto     -->  [].
resto     -->  arg, resto.
arg       -->  [NomeFich].
operSimp  -->  [lista].
operSimp  -->  [tamanho].
operSimp  -->  [apaga].
operSimp  -->  [renomeia].
operComp  -->  [copia].
operComp  -->  [move].
```

Responda, então, às alíneas seguintes:

- **a):** Dê exemplos de 2 frases válidas da linguagem definida pela DCG acima e de 2 frases incorrectas.
- **b):** Modifique a gramática supra de modo a permitir que cada frase da linguagem em causa possa ter mais do que um `comando`.
- **c):** Acrescente atributos aos símbolos não-terminais (i. é, argumentos aos predicados) da gramática de modo a que a DCG force que o comando `tamanho` e o comando `apaga` tenham precisamente 1 argumento e o comando `renomeia` tenha sempre 2 argumentos.
- **d):** Junte acções semânticas à DCG (e associe atributos aos símbolos, quando achar necessário) para que o programa gerado a partir da DCG acrescente à Base de Conhecimento (BC) o predicado `comando/2` que tenha como argumentos o *nome* e a *aridade* do *comando* reconhecido.

Exercício do Exame de Recurso de 1998. Para estudos na área da demografia das populações, construiu-se uma base de conhecimento (BC) com os dados que se puderam recolher sobre cada *individuo*, sobre os *casais* e sobre os *descendentes* (note-se que tanto os membros dos casais, como os filhos, têm de ser indivíduos). Para manipular essa BC foram, também, desenvolvidos alguns predicados operacionais, tais como `validaFam/1` e `filho/2`.
O extracto de Prolog seguinte mostra essa BC.

```
%indiv( C,N,LNasc,DNasc ) :- C 'e o codigo do individuo de nome N,
%                            nascido no local LNasc na data DNasc.
indiv( p001,joao,porto,1983 ).
indiv( p002,nuno,lisboa,1955 ).
indiv( p003,maria,braga,1956 ).
indiv( p004,joana,aveiro,1986 ).

%casal( C,CMu,CMa,D ) :-
%    C 'e o codigo que identifica o casal cuja mulher
%    'e o individuo de codigo CMu, o marido 'e o individuo
%    de codigo CMa, casados na data D
casal( cp001,p003,p002,1980 ).
```

```
%descendente( CC,CI)  :-
%  CI 'e o codigo de um individuo que 'e filho do casal CC
descendente( cp001,p001 ).
descendente( cp001,p004 ).

validaFam(Cod) :- casal(Cod, Mu, Ma, DC),
                  indiv(Mu, _, DNMu, _),
                  TolMu is DNMu+15, DC >= TolMu,
                  indiv(Ma, _, DNMa, _),
                  TolMa is DNMa+15, DC >= TolMa.



filho(F,Prog)  :- indiv(CodF, F, _, _), indiv(CodP, Prog, _, _),
                  descendente(CodFam, CodF),
                  ( casal(CodFam, CodP, _, _) ;
                    casal(CodFam, _, CodP, _)
                  ).
```

Responda, então, às alíneas seguintes:

a): Defina o predicado `descendeDe/1`, que recebe como argumento o nome de um indivíduo e escreve os dados conhecidos sobre o casal de quem ele é filho(a).

b): Defina o predicado `numeroFilhos/2`, que dá no segundo argumento o número total de filhos do casal cujo código é passado no primeiro argumento.

c): Defina o predicado `nascidosEmEntre/4`, que recebe o nome de um local e duas datas (nos 3 primeiros argumentos) e dá no quarto argumento a lista de todos os indivíduos nascidos nesse local, no período compreendido entre as duas datas especificadas.

d): Interprete o predicado `validaFam/1`, dizendo que lógica é que ele exprime e indicando para que serve e como se utiliza.

e): Estenda o predicado anterior, `validaFam/1`, para um novo predicado `validaFams/0`, que aplica o primeiro a todas as famílias.

f): Usando a árvore de procura, mostre a diferença entre a prova dos objectivos

```
?- validaFam(cp001).
?- validaFam(X).
```

g): Recorrendo à árvore de prova, mostre que `filho(joao,maria)` é verdadeiro.

h): Crie uma DCG para definir uma linguagem que sirva para descrever famílias de modo a carregar automaticamente a BC a partir das frases dessa linguagem, que deverão ser do género:

```
CASAL <cod> (Codmul,Codmar,data)
FILHOS <(CodF1,Data1),...>
```

i): Acrescente à DCG da alínea os atributos e as acções semânticas necessários para verificar que os códigos, tanto da mulher como do marido, existam na BC como indivíduos.

j): Com a linguagem que se lhe pediu atrás só é possível extrair informação para juntar à BC factos `casal` e `descendente`, mas não pode retirar dados para juntar factos `indiv`. Estenda, então, a gramática para que junto com a descrição dos filhos se incluam dados suficientes para acrescentar os ditos factos `indiv` à BC. Acrescente acções semânticas a essa extensão de modo a produzir mesmo esse resultado, durante o reconhecimento das novas frases.

9
Sistemas Periciais

1. Introdução e generalidades sobre os Sistemas Periciais

Os *Sistemas Periciais* (SP) são um dos primeiros sucessos comerciais da Inteligência Artificial. O seu uso na indústria, educação, ciência, medicina, ... tem aumentado durante os últimos anos.

Os SP são sistemas baseados em conhecimento, conhecimento esse a maior parte das vezes não procedimental, essencialmente declarativo. Isto faz com que o Prolog seja um candidato à construção de tais sistemas.

Várias áreas de trabalho, e investigação, encontram-se associadas ao desenvolvimento de SP. Algumas delas são:

- Estudo e implementação dos mecanismos de *raciocínio*.
- Aquisição e *representação de conhecimento*.
- *Verificação* e *validação* do conhecimento.
- Estudo e implementação de mecanismos para lidar com a *incerteza*.
- Estudo e implementação de vários paradigmas de *aprendizagem*.

Um SP é formado por um conjunto de programas que manipulam conhecimento, com a finalidade de resolver problemas num domínio especializado que normalmente requer peritos humanos.

Os SP diferem dos programas de computador convencionais, em vários aspectos importantes. Alguns deles são:

- Os SP usam conhecimento em vez de dados para controlar o processo de solução. Muito desse conhecimento não é algorítmico.
- O conhecimento é representado e mantido como uma entidade separada do programa de controlo. Isto permite a adição incremental de conhecimento, assim como a modificação (refinamento) da BC sem 'recompilar' o programa de controlo. É possível usar o mesmo programa de controlo com diferentes BC para produzir diferentes SP. Esses programas de controlo são conhecidos por *conchas* (do inglês *expert systems shells*).
- Os SP são capazes de *explicar* como uma dada conclusão foi alcançada ('*como?*') e porque é que determinada informação é precisa durante o processo de resolução ('*porquê?*'). Isto é importante, pois permite ao utilizador aceder e compreender o raciocínio do sistema, aumentando assim a confiança do utilizador no sistema.

Os SP surgiram em laboratórios de investigação, de umas poucas universidades, durante os anos 60 e 70. O primeiro SP a ser finalizado foi o *DENDRAL*, desenvolvido na Universidade de Stanford em finais dos anos 60. Este sistema era capaz de determinar a estrutura de componentes químicas, a partir de uma especificação dos elementos constituintes e da massa espectral do composto. Durante os testes, o DENDRAL descobriu um número de estruturas anteriormente desconhecidas dos peritos químicos! Este sistema conduziu depois ao

Meta-DENDRAL, um módulo de aprendizagem para o DENDRAL, que era capaz de aprender novo conhecimento (novas regras) a partir de exemplos positivos: uma forma de *aprendizagem indutiva*. Depois do sistema DENDRAL estar completo, começou o desenvolvimento do *MYCIN* na Universidade de Stanford. O MYCIN diagnosticava doenças infecciosas do sangue e recomendava uma lista de terapias para o doente.

Outro SP pioneiro foi o *PROSPECTOR*, um sistema que assistia os geólogos na descoberta de depósitos minerais.

Desde a introdução destes SP pioneiros, a vastidão de aplicações aumentou dramaticamente. Os métodos e as teorias de suporte aos sistemas modernos são também mais complexas, integrando vários paradigmas. Assim, para além dos métodos lógicos tradicionais, encontramos abordagens *conexionistas* (e.g., *redes neuronais*) e *paradigmas evolucionários* (e.g., *paradigmas genéticos*).

Aplicações podem hoje ser encontradas em quase todas as áreas do conhecimento. Para uma lista e breve descrição de aproximadamente 200 SP veja-se [18, Cap. 24]

2. Implementação de um Sistema Pericial em Prolog

Iremos aqui dar apenas "um cheirinho" das potencialidades e características de um SP.

Podemos dizer que a arquitectura mais comum usada na construção de SP, aliás como em outros tipos de sistemas baseados em conhecimento, são os sistemas baseados em regras. Estes sistemas usam o conhecimento codificado sobre a forma de regras de produção:

```
Se
  Cond1 e Cond2 e ... e CondN
Entao
  Conclusao.
```

Cada regra representa um pedacinho de conhecimento, relacionado com o domínio de peritagem escolhido. Em Prolog estas regras são escritas naturalmente como cláusulas.

A "concha" que iremos desenvolver irá usar este formalismo de *representação de conhecimento*, e o mecanismo de *raciocínio* (processo de inferência) lógico já disponibilizado pelo Prolog.

Tal como foi dito,

$$\text{Sistema Pericial} = \text{Conhecimento} + \text{Inferência}$$

e existe uma separação explícita entre as duas componentes.

Não entraremos aqui nas questões associadas à *aquisição*, *verificação* e *validação* de conhecimento, nem no estudo (e implementação) de mecanismos de *aprendizagem*.

Quanto ao tratamento de conhecimento *incerto*, iremos apenas dar (veja-se a secção 3) uma pequena introdução a uma das muitas abordagens existentes: o uso de conjuntos difusos.

2.1. Conhecimento. A Base de Conhecimento contém conhecimento específico de um determinado domínio (factos, regras, métodos, heurísticas, ...) e a indicação dos *predicados questionáveis* que fazem com que certos dados sobre o problema sejam solicitados ao utilizador quando necessário, de modo atingir o *objectivo*.

Segue-se um exemplo, em que o domínio de peritagem é a classificação de aves da América do Norte.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Base de Conhecimento (BC) 'aves.pl'
%% que conjuntamente com 'perito.pl'
%% forma o 'Sistema Pericial aves'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Para uma BC funcionar com o 'perito.pl'
% deve possuir:
%
```

```
%
%  - conhecimento;
%  - identificacao dos atributos questionaveis
%    (recorrendo aos predicados 'questiona');
%  - identificacao da especialidade da BC
%    (recorrendo ao predicado 'objectivo/1').
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Conhecimento.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ave(albatrozLaysan) :-
  familia(albatroz),
  cor(branca).
ave(albatrozPePreto) :-
  familia(albatroz),
  cor(cinzenta).
ave(cisneSibilante) :-
  familia(cisne),
  voz(silvoMusicalAbafado).
ave(cisneCorneteiro) :-
  familia(cisne),
  voz(corneteiroEspalhafatoso).
ave(gansoCanadiano) :-
  familia(ganso),
  estacao(inverno),
  pais(estadosUnidos),
  cabeca(preta),
  face(branca).
ave(gansoCanadiano) :-
  familia(ganso),
  estacao(verao),
  pais(canada),
  cabeca(preta),
  face(branca).
ave(patoBravo) :-
  familia(pato),
  voz(grasno),
  cabeca(verde).
ave(patoBravo) :-
  familia(pato),
  voz(grasno),
  cabeca(manchasCastanhas).

ordem(bicoTubular) :-
  narinas(tubularExterna),
  vive(mar),
  bico(curvo).
ordem(aquatico) :-
  patas(membranaInterdigital),
  bico(espalmado).

familia(albatroz) :-
  ordem(bicoTubular),
  dimensao(grande),
  asas(longaEstreita).
```

```
familia(cisne) :-
  ordem(bicoTubular),
  pescoco(comprido),
  cor(branca),
  voo(lento).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% E' necessario, para funcionar com o
%% 'perito.pl', que especifiquemos quais
%% os atributos questionaveis, i.e., quais
%% os factos primitivos.
%%
%% Para cada caso, usar um dos predicados:
%%   questiona/2
%%   questiona/3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cor(X) :-
  questiona(cor,X).
voz(X) :-
  questiona(voz,X,[silvoMusicalAbafado,corneteiroEspalhafatoso,grasno]).
estacao(X) :-
  questiona(estacao,X,[inverno,primavera,verao,outono]).
pais(X) :-
  questiona(pais,X).
cabeca(X) :-
  questiona(cabeca,X).
face(X) :-
  questiona(face,X).
narinas(X) :-
  questiona(narinas,X).
vive(X) :-
  questiona(vive,X).
bico(X) :-
  questiona(bico,X).
patas(X) :-
  questiona(patas,X).
dimensao(X) :-
  questiona(dimensao,X,[grande,pequena,media]).
asas(X) :-
  questiona(asas,X).
pescoco(X) :-
  questiona(pescoco,X).
voo(X) :-
  questiona(voo,X).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Identifica a 'especialidade' desta Base
%% de Conhecimento: identificar aves.
%%
%% Para isso devemos usar o predicado
%% 'objectivo/1'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
objectivo(X) :- ave(X).
```

2.2. Concha. A concha a desenvolver é formada por um *interface* (`perito/0`) que controla a comunicação entre o SP e o utilizador; por uma *memória de trabalho* que evita, por

memorização das respostas (`conhece/3`), que haja perguntas repetidas; e por um motor de inferência (`soluciona/0`).

A concha permite recorrer a várias BCs (uma de cada vez que consultamos o `perito.pl`, para evitar conflitos entre cláusulas de BCs diferentes) permitindo assim termos vários SPs para diferentes domínios de peritagem. A concha irá resolver o problema definido pelo predicado `objectivo/1` incluído na BC em questão.

Exemplo de uma sessão com o perito.pl:

```
?- consult(perito).
Yes

?- perito.
Concha simples de Sistema Pericial
Versao de 27/Maio/1999

Comandos disponiveis (introduza o numero ‘1.‘, ‘2.‘ ou ‘3.‘):
1 - Consultar uma Base de Conhecimento (BC)
2 - Solucionar
3 - Sair
> 1.
Nome da BC: aves.

BC consultada com sucesso.

Comandos disponiveis (introduza o numero 2 ou 3):
2 - Solucionar
3 - Sair
> 2.
narinas:tubularExterna? (sim/nao) sim.
vive:mar? (sim/nao) sim.
bico:curvo? (sim/nao) sim.
Qual o valor para dimensao?
[grande, pequena, media]
|: grande.
asas:longaEstreita? (sim/nao) sim.
cor:branca? (sim/nao) nao.
cor:cinzenta? (sim/nao) sim.

Resposta encontrada: albatrozPePreto

Comandos disponiveis (introduza o numero 2 ou 3):
2 - Solucionar
3 - Sair
> 3.
Volte Sempre!
Qualquer tecla para sair.
```

perito.pl:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% perito.pl, 27/Maio/1999
%%
%% Expert System Shell:
%%   Permite carregar a base de conhecimento desejada.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicado principal: perito/0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

perito :-
  write('Concha simples de Sistema Pericial'), nl,
  write('Versao de 27/Maio/1999'), nl, nl,
  esperaOrdens(123).

esperaOrdens(MC) :-
  mostraComandos(MC),
  write('> '),
  read(Comando),
  executa(MC,Comando).

mostraComandos(123) :-
  write('Comandos disponiveis (introduza o numero `1.`, `2.` ou `3.`):'),
  nl,
  write('1 - Consultar uma Base de Conhecimento (BC)'), nl,
  write('2 - Solucionar'), nl,
  write('3 - Sair'), nl.
mostraComandos(23) :-
  write('Comandos disponiveis (introduza o numero 2 ou 3):'), nl,
  write('2 - Solucionar'), nl,
  write('3 - Sair'), nl.

executa(_,1) :-
  write('Nome da BC: '),
  read(F),
  consult(F),
  write('BC consultada com sucesso.'), nl, nl,
  continua.
executa(_,2) :-
  soluciona,
  esperaOrdens(23).
executa(_,3) :-
  nl,
  write('Volte Sempre!'), nl,
  write('Qualquer tecla para sair.'),
  get0(_),
  halt.
executa(MC,X) :-
  write(X),
  write(' nao e um comando valido!'), nl,
  esperaOrdens(MC).

continua :-
  retract( executa(_,1) :- _ ), % Ja carrega'mos uma BC.
  esperaOrdens(23). % De futuro ja nao temos essa opcao.


%%%%%%%%%%%%%%
% soluciona/0
%%%%%%%%%%%%%%
```

```
soluciona :-
  abolish(conhece,3),
  asserta(conhece(def,def,def)), % apenas para o predicado
  objectivo(X),                  % conhece/3 estar definido...
  nl, nl, write('Resposta encontrada: '),
  write(X),
  nl, nl.
soluciona :-
  nl, nl, write('Nao foi encontrada resposta :-('), nl.


%%%%%%%%%%%%%%
% questiona/2
%%%%%%%%%%%%%%

questiona(Atributo,Valor) :-
  conhece(sim,Atributo,Valor).
questiona(Atributo,Valor) :-
  conhece(_,Atributo,Valor), !, fail.
questiona(Atributo,Valor) :-
  write(Atributo:Valor),
  write('? (sim/nao) '),
  read(R),
  processa(R,Atributo,Valor).

processa(sim,Atributo,Valor) :-
  asserta(conhece(sim,Atributo,Valor)).
processa(R,Atributo,Valor) :-
  asserta(conhece(R,Atributo,Valor)),!,
  fail.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% questiona/3
%
% Recurso a Menus:
%   sao apresentados ao utilizador os valores
%   que cada atributo pode assumir.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

questiona(Atr,Val,_) :-
  conhece(sim,Atr,Val).
questiona(Atr,_,_) :-
  conhece(sim,Atr,_), !, fail.
questiona(Atr,Val,ListaOpcoes) :-
  write('Qual o valor para '),
  write(Atr),
  write('? '), nl,
  write(ListaOpcoes), nl,
  read(X),
  processa(X,Atr,Val,ListaOpcoes).

processa(Val,Atr,Val,_) :-
  asserta(conhece(sim,Atr,Val)).
processa(X,Atr,_,ListaOpcoes) :-
  member(X,ListaOpcoes),
  asserta(conhece(sim,Atr,X)), !, fail.
```

```
processa(X,Atr,Val,ListaOpcoes) :-
  write(X),
  write(' nao e valor aceite!'), nl,
  questiona(Atr,Val,ListaOpcoes).
```

3. Incerteza

3.1. Motivação.

O conhecimento dos peritos, e os seus processos de raciocínio, nem sempre podem ser modelados usando a lógica booleana e as suas técnicas de inferência: muitas vezes o conhecimento é incerto. A *incerteza* pode-se manifestar de diversas formas:

- informação incompleta;
- informação conflituosa;
- imprecisão.

Muitos domínios são, por inerência, imprecisos e vagos. Em tais domínios, dados conflituosos podem surgir devido à presença de vários agentes com opiniões diferentes ou mesmo conflituosas. Nestas circunstâncias, o desempenho de um SP depende muito da maneira como a incerteza é gerida pelo sistema. Por esse motivo, um número de teorias alternativas foram desenvolvidas para gerir a incerteza. A maioria dessas teorias são quantitativas: é proposto um esquema de introdução de medidas que quantificam, numericamente, a incerteza e especificam como propagar e combinar essas medidas numéricas de incerteza durante o raciocínio. Como exemplo consideremos a seguinte regra que descreve uma heurística descrita por um perito em investimentos:

> *Se* o investidor é de meia idade e tem um salário baixo *Então* o investidor encaixa no perfil de um investidor de baixo risco.

O perito pode não confiar em absoluto nesta regra (usualmente os investidores com estas características são investidores de baixo risco, mas nem sempre ...). Um factor de confiança de 90% pode ser associada à regra que pode ser interpretada como significando que se o investidor satisfaz as condições da regra então ele é classificado como "investidor de baixo risco" com uma certeza de 90%. Mas também pode haver incerteza em saber quando as condições da regra são satisfeitas ou não. Podemos só estar 80% certos que o ordenado do investidor é baixo. Nesse caso, qual a certeza de o perfil do investidor ser um perfil de baixo risco? Deve ser menor do que 90%, mas como combinamos as incertezas? Outro factor que contribui para a incerteza é que os termos "meia idade" ou "salário baixo" são inerentemente difusos e vagos. Como tratar estes termos linguísticos imprecisos? Se fixarmos um intervalo para definir o termo "meia idade", por exemplo [45, 55], então o que dizer de uma pessoa que tem 44 anos, 11 meses e alguns dias? A regra não será aplicada a este investidor. O predicado `idade` tem um intervalo de definição contínuo. Tornar discreto este intervalo, isto é, considerar subintervalos correspondentes aos conceitos de novo, meia idade, etc., introduz o problema de lidar com os pontos fronteiros.

3.2. Conjuntos vagos e Lógica Vaga. Introdução

A *lógica difusa* (*Fuzzy logic*) é uma tentativa de lidar com conhecimento vago. A teoria dos conjuntos vagos (ou conjuntos difusos) foi desenvolvida nos anos 60 e generaliza as lógicas de n-valores. Numa lógica de n-valores, o conjunto de valores *verdade* foi estendido do 0 ou 1 (verdadeiro ou falso) para valores no conjunto de verdade

$$T_n = \left\{0, \frac{1}{n-1}, \frac{2}{n-1}, \cdots, \frac{n-2}{n-1}, 1\right\}$$

Por exemplo, numa lógica de 3-valores os valores de verdade permitidos são 0, 1/2 e 1, correspondendo ao falso, desconhecido e verdadeiro. A lógica difusa é uma lógica com infinitos valores, onde os valores de verdade estão no intervalo contínuo [0, 1].

Os peritos incorporam muitas vezes, nos seus processos de raciocínio, conceitos qualitativos tais como *alto*, *baixo*, *calor*, etc. e quantificadores como *muito*, *pouco*, *normalmente*, *algumas vezes*, etc. A imprecisão inerente nestes termos, é naturalmente implementada com a lógica difusa. O raciocínio é depois feito usando *inferência difusa*. Algumas das aplicações da lógica difusa são:

- Controlo de comboios (usado no Japão).
- Diagnóstico médico.
- Análise de risco.
- Focagem automática, exposição automática e estabilização de imagem em câmaras de vídeo e máquinas fotográficas digitais.
- Compressão de dados áudio e vídeo (HDTV).

Teoria dos conjuntos vagos

Na teoria dos conjuntos tradicional, um conjunto A pode ser definido em termos de uma função

$$\mu_A : U \to \{0, 1\},$$

do universo de discurso U no conjunto discreto $\{0, 1\}$, onde $\mu_A(x) = 1$ se x pertence ao conjunto A e 0 caso contrário.

Um *conjunto difuso* A é definido por uma função

$$\mu_A : U \to [0, 1].$$

O domínio desta função pode ser discreto ou contínuo. A imagem de um elemento x, $\mu_A(x)$, é um número real no intervalo fechado [0, 1] que indica a *grau de pertença* de x ao conjunto A. Note-se que este "grau de pertença" de um elemento no conjunto, não é a probabilidade do elemento pertencer ao conjunto: os graus de pertença de todos os elementos de U, não têm necessariamente de totalizar um.

Como exemplo, considere-se o conceito *alto* no universo de todos os estudantes, docentes e funcionários, da Universidade de Aveiro. Na lógica booleana, uma pessoa ou é *alta* ou não. Assim sendo, teríamos de escolher um limiar. Todos os elementos (pessoas) do universo (conjunto U) cuja altura fosse maior que esse limiar seriam classificadas como altas e aquelas com altura inferior ao limiar de *não altas*. Em contraste, na teoria dos conjuntos vagos um valor de pertença é associado a cada altura, indicando a quantidade de confiança em ser alto. A uma altura de 1m e 90cm podemos associar o valor 1.0 (pessoa sem dúvida alta no universo da Univ. de Aveiro); enquanto a uma altura de 1m e 80cm podemos associar um valor de pertença $\mu_{alto}(1.80) = 0.90$ e a 1m e 55cm $\mu_{alto}(1.55) = 0.02$.

A construção *apropriada* da função de pertença μ, é uma das principais dificuldades no uso da lógica e inferência difusa. Os métodos, que até agora foram propostos, são experimentais e *ad hoc*.

O *suporte* de um conjunto vago A, $sup(A)$, é definido como o conjunto de elementos do universo cujo grau de pertença é estritamente positivo: $sup(A) = \{u \in U : \mu_A(u) > 0\}$.

Um conjunto vago A de suporte finito, pode ser denotado por

$$A = \sum_{i=1}^{n} x_i/\mu_i$$

onde $sup(A) = \{x_1, x_2, \ldots, x_n\}$ e μ_i é o grau de pertença de x_i.

Termos linguísticos

Os graus de certeza de uma frase são representados por termos linguísticos como *possível*, *muito possível*, etc. Por exemplo, podemos dizer que: "É muito provável que o Sr. António seja rico". Aqui *rico* é um conjunto vago (por exemplo definido pelo rendimento anual bruto da pessoa) e *muito provável* um termo linguístico que associa um valor de verdade à afirmação de que o Sr. António é rico.

Quando a pertença de um elemento num conjunto vago é expressa por tais termos linguísticos, temos de definir as funções de pertença para esses quantificadores, de modo a exprimir o seu significado. Por exemplo, se A é um conjunto vago podemos definir a função de

pertença dos quantificadores *muito*, *mais ou menos* e *não* como se segue:

$$\mu_{\text{muito}(A)}(x) = \mu_A^2(x-\delta)$$
$$\mu_{\text{mais\_ou\_menos}(A)}(x) = \sqrt{\mu}_A(x+\delta)$$
$$\mu_{\text{nao}(A)}(x) = 1 - \mu_A(x)$$

onde δ é uma constante.

Operações com conjuntos vagos

Existem, na literatura especializada, muitas definições alternativas para as operações de união, intersecção, ... de conjuntos difusos. As mais usadas são as seguintes:

$$\mu_{A\cap B}(x) = min\,\{\mu_A(x),\, \mu_B(x)\}$$
$$\mu_{A\cup B}(x) = max\,\{\mu_A(x),\, \mu_B(x)\}$$
$$\mu_{A\Rightarrow B}(x) = \mu_{\neg A\vee B}(x) = max\,\{1-\mu_A(x),\, \mu_B(x)\}$$

Lógica difusa em Prolog

De seguida segue-se uma possível implementação da lógica difusa em Prolog. Testamos o programa num problema de *Tomada de Decisão*. O objectivo é construir um "sistema pericial" para apoio a hotéis. Concretamente pretendemos ajudar na decisão: "ligo ou não ligo o aquecedor?"

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Implementacao de Logica Difusa em Prolog.
% Delfim F. Marado Torres, 8/Junho/1999
%
% Exemplificacao:
%
%   ?- ligaAquecedor(C).
%
% devolve em C um valor entre 0 e 1.
% Os conselhos peremptorios sao:
%   C = 1 <=> aquecedor deve estar ligado;
%   C = 0 <=> aquecedor deve estar desligado.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Um conjunto difuso e' representado pelo facto
%
%  conjuntoDifuso(NomeConjunto,[ [X1,Y1], ..., [Xn,Yn] ]).
%
% O segundo argumento e' uma lista de listas [Xi,Yi]
% onde Xi e' um elemento do universo e Yi o respectivo
% grau de pertenca:  mu(Xi) = Yi.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

conjuntoDifuso(temperaturasAltas,[[0,0.0],[5,0.2],[10,0.4],[20,0.8]]).

conjuntoDifuso(pertoDe(0), [[0,1.0],[5,0.8],[10,0.5],[20,0.0]]).
conjuntoDifuso(pertoDe(5), [[0,0.8],[5,1.0],[10,0.6],[20,0.2]]).
conjuntoDifuso(pertoDe(10),[[0,0.5],[5,0.6],[10,1.0],[20,0.9]]).
conjuntoDifuso(pertoDe(20),[[0,0.0],[5,0.2],[10,0.9],[20,1.0]]).


%%%%%%%%%%%%%%%%%%%%%%%%%
% 'Funcao de pertenca'
%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
pertence(X,C,GP) :-
  conjuntoDifuso(C,L),
  pertence([X,GP],L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Quantificadores vagos, tal como o 'muito', sao
% definidos por regras do seguinte tipo:
%   Se [X,P] esta' na lista do conjunto A
%   Entao [X,P^2] esta' na lista muito(A)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

conjuntoDifuso(muito(A),LMA) :-
  conjuntoDifuso(A,LA),
  mu_muito(LA,LMA).

mu_muito([],[]).
mu_muito([[X,Y]|R],[[X,Y2]|RM]) :-
  Y2 is Y * Y,
  mu_muito(R,RM).

conjuntoDifuso(mais_ou_menos(A),LMMA) :-
  conjuntoDifuso(A,LA),
  mu_mm(LA,LMMA).

mu_mm([],[]).
mu_mm([[X,Y]|R],[[X,RY]|RM]) :-
  RY is sqrt(Y),
  mu_mm(R,RM).

conjuntoDifuso(nao(A),LNA) :-
  conjuntoDifuso(A,LA),
  mu_nao(LA,LNA).

mu_nao([],[]).
mu_nao([[X,Y]|R],[[X,CY]|RM]) :-
  CY is 1 - Y,
  mu_nao(R,RM).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Operacoes sobre conjuntos vagos
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

conjuntoDifuso(e([A]),LA) :-
  conjuntoDifuso(A,LA).
conjuntoDifuso(e([A,B]),LAB) :-
  conjuntoDifuso(A,LA),
  conjuntoDifuso(B,LB),
  mu_e(LA,LB,LAB).
conjuntoDifuso(e([A|R]),L) :-
  conjuntoDifuso(e([A,e(R)]),L).

mu_e([],_,[]).
mu_e([[X,Y1]|R],L2,[[X,Y]|RE]) :-
  pertence([X,Y2],L2),
  min(Y1,Y2,Y),
  mu_e(R,L2,RE).
```

```
mu_e([_|R],L2,RE) :-
  mu_e(R,L2,RE).

conjuntoDifuso(ou([A]),LA) :-
  conjuntoDifuso(A,LA).
conjuntoDifuso(ou([A,B]),LAB) :-
  conjuntoDifuso(A,LA),
  conjuntoDifuso(B,LB),
  mu_ou(LA,LB,LAB).
conjuntoDifuso(ou([A|R]),L) :-
  conjuntoDifuso(ou([A,ou(R)]),L).

mu_ou([],L,L).
mu_ou([[X,Y1]|R],L2,[[X,Y]|RE]) :-
  pertence([X,Y2],L2),
  max(Y1,Y2,Y),
  tira([X,Y2],L2,NL2),
  mu_ou(R,NL2,RE).
mu_ou([[X,Y1]|R],L2,[[X,Y1]|RE]) :-
  mu_ou(R,L2,RE).


conjuntoDifuso(implica(Ant,Cons),L) :-
  conjuntoDifuso(ou([nao(Ant),Cons]),L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Regra difusa (vaga):
%
%  Se
%    a Temperatura Exterior nao e' alta e
%    a Temperatura Interior esta' proxima
%    da Exterior
%  Entao
%    o Aquecedor deve estar ligado.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
aquecedorLigado(GP) :-
  temperaturaExterior(TE),
  temperaturaInterior(TI),
  pertence(TE,e([nao(temperaturasAltas),pertoDe(TI)]),GP), !.

%%%%%%%%%%%%%%%%%
% Interface
%%%%%%%%%%%%%%%%%

temperaturaExterior(TE) :-
  temperatura(exterior,TE).

temperaturaInterior(TI) :-
  temperatura(interior,TI).

temperatura(M,T) :-
  repeat,
  nl,
  write('Valor aproximado (0, 5, 10 ou 20) da temperatura '),
  write(M), write(' = '),
  read(T),
  pertence(T,[0,5,10,20]), nl.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicados Auxiliares
%%%%%%%%%%%%%%%%%%%%%%%%%%

pertence(X,[X|_]).
pertence(X,[_|R]) :-
  pertence(X,R).

tira(X,[X|R],R).
tira(X,[Y|R],[Y|T]) :-
  tira(X,R,T).

min(X,Y,X) :- X =< Y.
min(_,Y,Y).

max(X,Y,X) :- X >= Y.
max(_,Y,Y).
```

4. Exercícios

Geração de Explicações. Pense numa maneira, e implemente-a, de alterar o `perito.pl` de modo a permitir dar resposta a perguntas do utilizador do tipo "Porque é que quer saber isto?"; "Como é que chegou a esta conclusão?".

Tratamento de conhecimento incerto. Altere o `perito.pl` de modo a permitir lidar com conhecimento incerto, por meio da abordagem difusa atrás considerada.

Aulas Teórico-Práticas

10
Guião para a Primeira Aula

1. Objectivos da aula

- Aprender a representar conhecimento em Prolog e a "questionar" a base de conhecimento. Exemplificação com uma pequena base de conhecimento sobre vídeo gravadores, disponível em

  `g:\prolog\ipl9899\videos.pl`

  Nota: ficheiros com exemplos, informação variada sobre a cadeira e ficheiros de instalação dos interpretadores podem ser encontrados em rede, na directoria

  `g:\prolog`

  O repositório "oficial" da cadeira será, no entanto, o URL

  `http://www.mat.ua.pt/delfim/cadeiras/ipl99/ipl.htm`
- Introdução ao uso do SWI-Prolog e outros interpretadores, em ambiente Windows, MS-DOS ou Linux.
- Aprender a introduzir/alterar conhecimento usando um editor de texto em ambiente Windows, MS-DOS ou Linux.
- Aprender a chamar o interpretador e a consultar uma base de conhecimento.
- Uso do ficheiro "videos.pl". Colocação de várias questões (recorrendo já ao conceito de *variável*).

2. Comandos do Prolog

O texto de um programa em Prolog é normalmente criado num ficheiro, ou conjunto de ficheiros, usando um dos editores de texto standard (por exemplo o *edit* que vem com o MS-DOS ou o notepad do Windows). O interpretador de Prolog pode depois ser instruído a ler os programas ou bases de conhecimento destes ficheiros – a este processo chamamos *consultar*. Embora esses ficheiros possam ter uma extensão arbitrária, optamos por usar a extensão *.pl*, que é a extensão por defeito usada pelo SICStus e pelo SWI-Prolog.

2.1. Iniciar o interpretador de Prolog. Existe um icon para o interpretador SWI-Prolog no Windows. Depois de clicar duas vezes nesse icon, o interpretador de Prolog fica à espera de "ordens", aparecendo o prompt

```
?-
```

2.2. Consultar bases de conhecimento. Para consultar uma base de conhecimento que se encontra no ficheiro `info.pl`, na directoria

```
c:\prolog\ipl\
```

introduzir

```
?-consult('c:/prolog/ipl/info.pl').
```

2.3. Mais Comandos.

- `halt.` (Sair do interpretador de Prolog)
- `listing.` (Mostrar a informação carregada em memória. Experimentar após o `consult` de um ficheiro.)

3. Exemplo (videos.pl)

```
% Pequena Base de Conhecimento sobre video gravadores
% usa-se o predicado video de aridade 8:
% video(Marca,
%       Modelo,
% Tipo_de_video,
% Mono_ou_stereo,
% Uma_ou_2_velocidades,
% Insert_ou_nao,
% Pip_ou_nao,
% Dobragem_audio_ou_nao).

video(jvc,d540,vhs,mono,uma_velocidade,n_insert,n_pip,n_dobragem_audio).
video(jvc,fc100,vhs,mono,dupla_velocidade,n_insert,n_pip,n_dobragem_audio).
video(jvc,d830,vhs,stereo,dupla_velocidade,n_insert,n_pip,dobragem_audio).
video(jvc,d960,vhs,stereo,dupla_velocidade,insert,n_pip,dobragem_audio).
video(jvc,s5800,s_vhs,stereo,dupla_velocidade,insert,n_pip,dobragem_audio).

video(sanyo,vhr8100,vhs,mono,uma_velocidade,n_insert,n_pip,n_dobragem_audio).
video(sanyo,vhr8500,vhs,mono,dupla_velocidade,n_insert,n_pip,n_dobragem_audio).
video(sanyo,vhr8700,vhs,stereo,dupla_velocidade,n_insert,n_pip,n_dobragem_audio).
video(sanyo,vhr4890,vhs,stereo,dupla_velocidade,insert,pip,dobragem_audio).

video(grundig,vs630,vhs,mono,uma_velocidade,n_insert,n_pip,n_dobragem_audio).
video(grundig,vs810,vhs,mono,uma_velocidade,insert,n_pip,n_dobragem_audio).
video(grundig,vs660,vhs,stereo,dupla_velocidade,n_insert,n_pip,n_dobragem_audio).
video(grundig,vs680,s_vhs,stereo,dupla_velocidade,insert,pip,dobragem_audio).
```

3.1. Algumas perguntas para colocar ao Prolog.

- Que marcas têm vídeos super vhs?
- Quais os modelos, e respectivas marcas, vhs, stereo, com dupla velocidade e insert?

4. Exercício: criar ficheiro socios.pl

- O João, a Susana, o Basílio e a Elvira são sócios da Associação Académica.
- O João é casado com a Susana.
- A Elvira é irmã do Basílio.

4.1. Algumas perguntas. Depois de criada a base de conhecimento e feita a respectiva consulta, coloque as seguintes questões ao interpretador de Prolog.

- Quem é sócio da Associação Académica?
- O Basílio é irmão da Elvira?
- A Miquelina é sócia da Associação Académica?
- Quem é casado com a Susana?

5. Mais ideias a reter

- Tudo o que se segue a % é considerado pelo interpretador de Prolog como um comentário.
- As variáveis em Prolog começam por maiúsculas.
- No decurso de uma computação, uma variável pode ser substituída por um objecto concreto. Dizemos então que a variável foi *instanciada*.

11
Guião para a Segunda Aula

1. Objectivos da aula

- Algumas considerações sobre os primeiros trabalhos práticos.
- Consolidação, através de exercícios, dos seguintes conceitos:
  - regras (implicação e conjunção);
  - recursão (definição de regras recursivas);
  - retrocesso ("backtracking").
- Como é que o Prolog responde às questões?

2. Exercícios

2.1. Exercício (regras, recursividade, como o Prolog responde às questões). Considere a seguinte situação:

- Um objecto A está sobre uma mesa;
- Um objecto B está sobre o objecto A.

a): Instrua o interpretador de Prolog sobre esta situação, usando para o efeito o predicado *sobre*.

b): Exprima por meio de fórmulas lógicas (e de seguida na notação do Prolog) as seguintes regras de conhecimento:

i): Se um objecto está sobre outro, então está acima dele.

ii): O predicado *acima_de* é transitivo.

c): Coloque a seguinte questão ao interpretador de Prolog: "B está acima da mesa?".

2.2. Exercício (regras, retrocesso e recursividade). Numa empresa fabricante de equipamento eléctrico para cozinhas, encontram-se as seguintes proposições sobre a constituição de um **fogão**:

- Um fogão é composto por uma estrutura e um cordão eléctrico.
- Umas das componentes da estrutura é uma resistência de aquecimento.
- A resistência de aquecimento é em metal.
- Outra parte da estrutura é o painel do fogão.
- O painel tem um botão.
- Os botões são sempre feitos em plástico.
- O cordão eléctrico é composto de fio metálico.
- Parte do cordão eléctrico é um isolador.
- O isolador é feito de fibra plástica.

Responda, então, às alíneas seguintes:

a): Recorrendo apenas aos predicados `parte-de` e `feito-em` (note que uma palavra diferente não corresponde necessariamente a um predicado ou argumento distinto), escreva um conjunto de cláusulas Prolog que descrevam precisamente o conhecimento contido nas frases acima.

b): Tomando em consideração a Base de Conhecimento criada na alínea anterior, diga como procederia, face a um Interpretador de Prolog, para obter resposta para às seguintes questões:

- Que objectos (simples) são de metal ?
- Que objectos (simples) não são de plástico ?

c): Tomando em consideração a Base de Conhecimento criada na alínea a), escreva um predicado que permita determinar se um dado objecto faz ou não parte do fogão.

Solução:

```
% a)

parte_de(estrutura,fogao).
parte_de(cordao_electrico,fogao).
parte_de(resistencia,estrutura).
parte_de(painel,estrutura).
parte_de(botao,painel).
parte_de(fio,cordao_electrico).
parte_de(isolador,cordao_electrico).


feito_em(resistencia,metal).
feito_em(botao,plastico).
feito_em(fio,metal).
feito_em(isolador,plastico).

% b):  feito_em(X,metal).
%           OU (analisar diferen,cas!!)
%      feito_em(X,metal), write(X), nl, fail.


%      feito_em(X,M), M \= plastico.
%           OU (analisar diferen,cas!!)
%      feito_em(X,M), M \= plastico, write(X), nl, fail.

% c):  parte_fogao(X) :- parte_de(X,fogao).
%      parte_fogao(X) :- parte_de(X,Y), parte_fogao(Y).
```

3. Mais ideias a reter

- Em Prolog existem cláusulas de três tipos: *factos*, *regras* e *questões*.
- O Prolog usa um conceito de "mundo fechado": tudo o que não está explicitado na base de conhecimento, ou dela pode ser deduzido por aplicação de regras, é considerado falso.

12

Guião para a Terceira Aula

1. Objectivos da aula

- Aritmética em Prolog: $+$, $-$, $*$, /, *mod*, //; e o operador **is**
- Operadores de comparação: $>$, $<$, $>=$, $=<$, $=:=$, $=\backslash=$
- Resolução do "Problema do macaco e das bananas"
- Ida aos computadores:
  - teste *in loco* da resolução adoptada para o problema do macaco e das bananas
  - esclarecimento de dúvidas acerca dos trabalhos

2. Aritmética - Exemplos

```
?- X = 1+2.
X = 1+2

?- X is 1+2.
X = 3

?- X is 3/2, Y is 3 // 2.
X = 1.5
Y = 1

?- 277 * 37 > 10000.
yes

?- 1+2 = 2+1.
no

?- 1+2 =:= 2+1.
yes

?- 1+A = B+2.
A=2
B=1
```

3. Problema do macaco e das bananas

Um macaco está à porta duma sala. No centro da sala está uma banana pendurada no tecto. O macaco está com fome e quer a banana, mas não consegue chegar à altura a que esta está. Junto à janela da sala está uma caixa que pode ser usada pelo macaco e que o pode ajudar a chegar à altura a que está a banana.

O macaco pode realizar as seguintes acções: andar no chão; subir para cima da caixa; empurrar a caixa para qualquer ponto da sala; apanhar a banana (se subir para cima da caixa estando no centro da sala).

Pergunta: o macaco pode apanhar a banana, ou estará condenado a passar fome?

Resolução:

Em cada instante, vamos descrever a *situação* do nosso "jogo" através:

- do ponto da sala onde está o macaco;
- se o macaco está no chão ou em cima da caixa;
- do ponto da sala onde está a caixa;
- se o macaco tem ou não a banana.

O objectivo do jogo será:

$$\text{situacao}\,(\_,\_,\_,\mathit{tem\_banana})\,.$$

As *acções* permitidas, que fazem passar de uma situação a outra, são:

- apanhar a banana;
- subir a caixa;
- empurrar a caixa;
- andar

No entanto, cada acção só é permitida em certas situações.

```
% ficheiro macaco.pl
accao(situacao(centro,cima_caixa,centro,nao_tem),
      apanhar,
      situacao(centro,cima_caixa,centro,tem)).
accao(situacao(P,chao,P,TN),
      subir,
      situacao(P,cima_caixa,P,TN)).
accao(situacao(P1,chao,P1,TN),
      empurrar(P1,P2),
      situacao(P2,chao,P2,TN)).
accao(situacao(P1,chao,C,TN),
      andar(P1,P2),
      situacao(P2,chao,C,TN)).

% cumpre_obj(S) <=> macaco pode obter banana partindo de S
cumpre_obj(situacao(_,_,_,tem)).
cumpre_obj(E) :-
  accao(E,A,S),
  cumpre_obj(S).
```

Depois de fazer o consult do ficheiro, basta perguntar ao interpretador:

```
?- cumpre_obj(situacao(porta,chao,janela,nao_tem)).
yes
```

Exercício: Justifique, usando uma árvore de prova, a resposta dada pelo interpretador.

Questão: que se passa se a ordem por que aparecem as acções possíveis for alterada?

Conforme fizemos, o macaco prefere apanhar a banana, a subir a caixa, a empurrar a caixa, a movimentar-se. Se colocássemos a acção *andar* em primeiro lugar, o Prolog nunca encontraria uma solução para a questão posta. Experimente! Note, no entanto, que a árvore de prova neste caso é precisamente a mesma!

4. Mais ideias a reter

- Se nos argumentos das operações de comparação ou no argumento direito de **is** aparecerem variáveis, elas têm de estar instanciadas com números no momento de cálculo.
- Um programa pode estar declarativamente correcto, mas procedimentalmente incorrecto (poderá ser necessário reordenar as cláusulas do programa).

13
Guião para a Quarta Aula

1. Objectivos da aula

Manipulação de listas:

- Definição (recursiva) de lista (introduzida na aula teórica);
- Representação de listas em Prolog (introduzida na aula teórica);
- Consolidação do conceito de lista, à custa da resolução de vários exercícios: `adiciona/3`, `tira/3`, `concatena/3`, `adiciona_fim/3`, `inverte/2`. Faz-se notar a versatilidade dos predicados implementados, os quais podem ser usados com fins não pensados à priori (uso de várias configurações possíveis para *inputs* e *outputs*).

2. Alguns exercícios a resolver na aula

```
concatena([],L,L).
concatena([X|R],L,[X|C]) :-
  concatena(R,L,C).

adiciona(X,L,[X|L]).

tira(_,[],[]).
tira(X,[X|R],R).
tira(X,[Y|R],[Y|R1]) :-
  tira(X,R,R1).

adiciona_fim(X,L,NL) :-
  concatena(L,[X],NL).
```

3. Mais Exercícios

Implemente os predicados `tira_todos/3`, `pertence/2`, `ultimo/2`, `sublista/2`, `permutacao/2` e `ordena/2` (fazer uma versão para ordenar uma lista de números por ordem crescente, e uma outra para ordenar por ordem decrescente).

```
tira_todos(_,[],[]).
tira_todos(X,[X|R],NR) :- tira_todos(X,R,NR).
tira_todos(X,[Y|R],[Y|R1]) :- tira_todos(X,R,R1).

pertence(X,[X|_]).
pertence(X,[_|R]) :- pertence(X,R).

ultimo([U],U).
ultimo([_|R],U) :- ultimo(R,U).
```

```
sublista(S,L) :-
  concatena(_,S,L1),
  concatena(L1,_,L).

permutacao([],[]).
permutacao(L,[X|P]) :-
  tira(X,L,L1),
  permutacao(L1,P).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados de ordena,c~ao.
%% Recebem sempre dois argumentos. Retornam
%% ''yes'' se o primeiro e'
%% ''menor'' que o segundo
%% e ''no'' caso contra'ario.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
antes_c(X,Y) :-  % serve para ordenar uma lista de nu'meros
  X =< Y.        % por ordem crescente

antes_d(X,Y) :-  % serve para ordenar uma lista de nu'meros
  X >= Y.        % por ordem decrescente
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ordenada(_,[]).
ordenada(_,[_]).
ordenada(P,[X,Y|R]) :-
  ordenada(P,[Y|R]),
  C =..[P,X,Y],
  call(C).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ORDENA usando um algoritmo do tipo Bubble Sort
%%    bubble_sort/3
%%    ordena_c/2
%%    ordena_d/2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bubble(_,[X],[X]).
bubble(P,[X,Y|R],[X|O]) :-
  C =.. [P,X,Y],
  call(C),
  bubble(P,[Y|R],O).
bubble(P,[X,Y|R],[Y|O]) :-
  bubble(P,[X|R],O).

bubble_sort(P,L,L) :-
  ordenada(P,L).
bubble_sort(P,L,LO) :-
  bubble(P,L,L1),
  bubble_sort(P,L1,LO).

ordena_c(L,LO) :-
  bubble_sort(antes_c,L,LO).
ordena_d(L,LO) :-
  bubble_sort(antes_d,L,LO).
```

14
Guião para a Quinta Aula

1. Objectivos da aula

Resolução dos seguintes problemas:

- *"Élle-é-érre"*, proposto no CeNPL'99 (Concurso/encontro Nacional de Programação em Lógica de 1999);
- *"Os dez degraus do Miguel"*, proposto no CNPL'98.

2. Élle-é-érre: Ler sequências de algarismos

Considere a seguinte cadeia de algarismos: 118. Se a ler, em voz alta agrupando os algarismos homónimos, leria: *dois uns um oito*, que pode por sua vez ser representada por uma cadeia de algarismos: 2118. Esta cadeia pode agora por sua vez ser lida, obtendo-se: 122118.

2.1. Tarefa. Escreva, em Prolog, um predicado binário `eleiturade(S,E)` em que a lista de algarismos `S` é a leitura da lista de algarismos `E`.

2.2. Os Resultados. Como exemplos, tem que os seguintes predicados são verdadeiros:

```
eleiturade([2,1,1,8],[1,1,8])
eleiturade([1,2,2,1,1,8],[2,1,1,8])
eleiturade([1,1,1,2,1,3],[1,2,3])
eleiturade([3,1,1,2,1,1,1,3],[1,1,1,2,1,3])
eleiturade([1,3,2,1,1,2,3,1,1,3],[3,1,1,2,1,1,1,3])
```

Como estamos a usar o Prolog, e queremos que cada algarismo seja um elemento da lista, temos que:

```
eleiturade([1,0,1],[1,1,1,1,1,1,1,1,1,1])
```

2.3. Uma solução.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Realizado pelos alunos de IPL'99 na aula teorico-pratica
%% -- 16/Abril/1999 -- partindo de uma ideia da Dacha.
%%
%% Introduzido em computador por Cristina.
%%
%% Testado por Delfim.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ?- desmembra(123,L).
%    L = [1,2,3]
%
% ?- desmembra(7,L).
%    L = [7]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

desmembra(N,L):- name(N,LA),
                 conv_ascii(LA,L).

conv_ascii([],[]).
conv_ascii([XA|RA],[X|R]) :-
                   name(X,[XA]),
                   conv_ascii(RA,R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ?- tabela(L,[1,1,1]).
%    L = [3,1]
%
% ?- tabela(L,[3,3,3,3,3,3,3,3,3,3,3,3,3]).
%    L = [1,2,3]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tabela(LF,[X|R]) :- tamanho([X|R],N),
                    desmembra(N,L),
                    junta(L,[X],LF).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ?- agrupa([1,1,8],L).
%    L = [[1,1],[8]]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

agrupa([],[]).
agrupa(L,[L1|R]) :- separa(L,L1,L2),
                    agrupa(L2,R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ?- separa([1,1,8,2,2],L1,L2).
%    L1 = [1,1]
%    L2 = [8,2,2]
%
% ?- separa([8,2,2,1],L1,L2).
%    L1 = [8]
%    L2 = [2,2,1]
%
% L1 possui os primeiros elementos todos iguais
% L2 possui os restantes elementos
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

separa([],[],[]).
separa([X],[X],[]).
separa([X,X|R],[X|C],L2) :- separa([X|R],C,L2).
separa([X,Y|R],[X],[Y|R]).


%%%%%   PREDICADO PRINCIPAL     %%%%%
```

```
eleiturade(O,I) :- agrupa(I,LL),
                   usa_tabela(LL,O), !.

usa_tabela([],[]).
usa_tabela([X|R],L) :- tabela(Y,X),
                       usa_tabela(R,T),
                       junta(Y,T,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%   PREDICADOS AUXILIARES   %%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

junta([],L,L).
junta([X|R],L,[X|RR]) :-
  junta(R,L,RR).

tamanho([],0).
tamanho([_|R],N) :-
  tamanho(R,N1),
  N is N1 + 1.
```

3. Os dez degraus do Miguel

"À entrada da casa do Miguel há uma escada com 10 degraus. Cada vez que entra em casa, o Miguel avança pelas escadas subindo um ou dois degraus em cada passada. De quantas maneiras diferentes pode o Miguel subir as escadas?"[1]

3.1. Tarefa. A sua tarefa consiste em desenvolver um programa em Prolog que resolva este problema, para um número qualquer de degraus.

3.2. Os Dados. O único dado necessário para a resolução deste problema é o número de degraus. Esse valor será um parâmetro do predicado *jogo_degraus* que tem de desenvolver, como se explica a seguir.

3.3. Os Resultados. O seu programa deve ser activado através do predicado *jogo_degraus/3* que recebe o número de degraus e devolve o número de possibilidades diferentes de subir as escadas, assim como a respectiva lista de possibilidades (cada possibilidade é, por sua vez, também representada por uma lista).

Exemplo:

```
?-jogo_degraus(3,N,L).
  N = 3
  L = [[1,1,1],[1,2],[2,1]]
```

3.4. Uma solução.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Realizado por Delfim F. Marado Torres
%%  -- 07/Abril/1998 --
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% jogo_degraus(NumeroDegraus,
%%              NumPossibilidades,
%%              ListaPossibilidades)
%%
%% Exemplo:
%%  ?- jogo_degraus(5,N,L).
```

[1] José Paulo Viana e Cristina Sampaio, *desafios*, Pública (revista integrante do Jornal Público), Domingo, 29 Março (também 5 Abril), 1998.

```
%%  N = 8
%%  L = [[1,1,1,1,1],[1,1,1,2],[1,1,2,1],
%%       [1,2,1,1],[1,2,2],[2,1,1,1],
%%       [2,1,2],[2,2,1]]
%%  yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

jogo_degraus(ND,NP,LP) :-
  degraus(ND,LP),
  comprimento(LP,NP),!.

comprimento([],0).
comprimento([_|R],N) :-
  comprimento(R,N1),
  N is N1 + 1.


degraus(1,[[1]]).
degraus(2,[[1,1],[2]]).
degraus(N,L) :-
  N1 is  N-1,
  N2 is N-2,
  degraus(N1,L1),
  degraus(N2,L2),
  adiciona(1,L1,NL1),
  adiciona(2,L2,NL2),
  concatena(NL1,NL2,L).

adiciona(_,[],[]).
adiciona(N,[X|R],[NX|NR]) :-
  concatena([N],X,NX),
  adiciona(N,R,NR).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados auxiliares normalmente disponibilizados
%% nos interpretadores de Prolog:
%%       concatena/3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

concatena([],L,L).
concatena([X|R],L,[X|C]) :-
  concatena(R,L,C).
```

15
Guião para a Sexta Aula

Diofanto

1. Cálculo do Máximo Divisor Comum de dois números

É bem sabido que quaisquer dois inteiros não nulos a e b admitem um *m.d.c.* Vamos examinar um dos métodos de cálculo do *m.d.c.* chamado *algoritmo de Euclides*.[1] Admitiremos, para fixar as ideias, que $|a| \geq |b|$. Procedemos por etapas.

Primeira Etapa.: Divide-se a por b:

$$a = q_1\, b + r_1, \qquad 0 \leq r_1 < |b|\,.$$

Se $r_1 = 0$, então $m.d.c.\,(a,\, b) = b$. Se $r_1 \neq 0$, passamos à
Segunda Etapa.: Divide-se b por r_1:

$$b = q_2\, r_1 + r_2, \qquad 0 \leq r_2 < r_1.$$

Se $r_2 \neq 0$, passamos à etapa seguinte.
Terceira Etapa.:

$$r_1 = q_3\, r_2 + r_3, \qquad 0 \leq r_3 < r_2.$$

Iteramos este processo, repetindo sucessivamente as respectivas operações. Em cada estágio o resto obtido será estritamente inferior ao da etapa precedente, isto é,

$$|b| > r_1 > r_2 > \ldots,$$

logo, num certo estágio, digamos na $k-$ésima etapa, onde necessariamente $k < |b|$, o resto será nulo, quer dizer:

$k-$ésima Etapa.:

$$r_{k-2} = q_k\, r_{k-1}.$$

É fácil de ver que o último resto distinto do zero, a saber r_{k-1}, será o *m.d.c.* de a e b que procuramos. De facto, têm lugar as seguintes igualdades:

$$\begin{array}{rrcl}
(1) & a & = & q_1\, b + r_1; \\
(2) & b & = & q_2\, r_1 + r_2; \\
(3) & r_1 & = & q_3\, r_2 + r_3; \\
 & & & \vdots \\
(k-1) & r_{k-3} & = & q_{k-1}\, r_{k-2} + r_{k-1}; \\
(k) & r_{k-2} & = & q_k\, r_{k-1}.
\end{array}$$

[1] Este método aparece descrito nos *"Elementos"* de Euclides.

Da última destas igualdades decorre que r_{k-1} divide r_{k-2}; da penúltima, dado que r_{k-1} divide r_{k-1} e r_{k-1} divide r_{k-2}, resulta que r_{k-1} divide r_{k-3}. Assim, passando de cada uma das igualdades à precedente, constatamos finalmente que r_{k-1} divide r_2, r_{k-1} divide r_1, r_{k-1} divide b e r_{k-1} divide a. Logo r_{k-1} é um divisor comum de a e b. Para mostrarmos que r_{k-1} é o *m.d.c.* de a e b falta-nos mostrar que qualquer divisor comum de a e b divide r_{k-1}. Suponhamos então que c divide a e c divide b. Então, das igualdades (1), (2), ..., $(k-1)$, obteremos sucessivamente que c divide r_1, c divide r_2, ..., c divide r_{k-1}, provando-se o pretendido.

Examinemos o exemplo com $a = 858$ e $b = 253$. Encontremos o *m.d.c.* destes números, sem decompô-los em factores primos, aplicando o algoritmo de Euclides:

$$\begin{array}{rrcl} (a) & 858 & = & 3 \cdot 253 + 99; \\ (b) & 253 & = & 2 \cdot 99 + 55; \\ (c) & 99 & = & 1 \cdot 55 + 44; \\ (d) & 55 & = & 1 \cdot 44 + 11; \\ (e) & 44 & = & 4 \cdot 11, \end{array}$$

donde o *m.d.c.* $(858, 253) = 11$.

2. Algumas equações em números inteiros

Um resultado também bem conhecido[2] é o de que o número $d = m.d.c.\,(a, b)$ pode ser escrito como uma combinação linear de a e b:

$$d = s\,a + t\,b,$$

onde s e t são inteiros. O algoritmo de Euclides permite determinar o valor de s e t. Não vamos dar aqui uma demonstração geral deste facto, ilustrando-o simplesmente para o exemplo que acabámos de examinar. Assim, pretendemos encontrar os inteiros s e t que satisfazem a relação

$$11 = s \cdot 858 + t \cdot 253.$$

Das igualdades (d), (c), (b) e (a) obtemos, respectivamente:

$$\begin{array}{rcl} 11 & = & 55 + (-1) \cdot 44, \\ 44 & = & 99 + (-1) \cdot 55, \\ 55 & = & 253 + (-2) \cdot 99, \\ 99 & = & 858 + (-3) \cdot 253. \end{array}$$

Substituindo na primeira destas igualdades a expressão para 44 da segunda, em seguida, substituindo a expressão para 55 da terceira, etc., obteremos

$$\begin{array}{rcl} 11 & = & 55 + (-1) \cdot (99 + (-1) \cdot 55) = 2 \cdot 55 + (-1) \cdot 99 = \\ & = & 2 \cdot (253 + (-2) \cdot 99) + (-1) \cdot 99 = 2 \cdot 253 + (-5) \cdot 99 = \\ & = & 2 \cdot 253 + (-5) \cdot (858 + (-3) \cdot 253) = (-5) \cdot 858 + 17 \cdot 253. \end{array}$$

Logo, $s = -5$, $t = 17$.

As igualdades que ocorrem ao se encontrar o *m.d.c.* de dois números a e b mediante o algoritmo de Euclides permitem, pois, resolver em números inteiros qualquer equação do tipo

$$d = x\,a + y\,b$$

com $d = m.d.c.\,(a, b)$.

Em geral, qualquer equação da forma

$$(ED) \qquad x\,a + y\,b = c,$$

onde a, b e c são números inteiros conhecidos e da qual se procuram as soluções inteiras, denomina-se *equação diofantina*[3] *linear* com duas incógnitas. A equação é linear, as incógnitas

[2] Ver, por exemplo, o Cap. I, Teorema 3, do livro *Divisão Inexata* de A. A. Belsky e L. A. Kalujnin, publicado na colecção *Iniciação na Matemática* da editora MIR.

[3] O termo *"diofantina"* vem de Diofanto, matemático grego da antiguidade (III século a. C.) que na sua *"Aritmética"* estudou equações em números inteiros.

x e y estando na primeira potência. O termo *"diofantina"* provém do facto de serem inteiros os coeficientes e de se procurarem soluções inteiras.

Passemos à análise da existência de soluções da equação (ED) no caso geral. Observemos, antes de tudo, que nem toda a equação deste tipo admitirá uma solução. De facto, suponhamos que os inteiros $x = x_0$ e $y = y_0$ satisfazem a equação (ED), quer dizer, $c = x_0\,a + y_0\,b$. Neste caso, se $d = m.d.c.\,(a,\,b)$, então o facto de d ser divisor comum de a e b resulta que d dividirá ambos os termos à direita nesta igualdade e que, por conseguinte, dividirá c. Daqui decorre a seguinte afirmação.

> *Para que uma equação do tipo (ED) admita uma solução em números inteiros, é necessário que o número c seja divisível pelo m.d.c. dos coeficientes a e b.*

A equação

$$9\,x + 15\,y = 7,$$

por exemplo, não terá solução em números inteiros, 7 não sendo divisível por $3 = m.d.c.\,(9,\,15)$. Por outro lado, uma equação (ED), na qual c é divisível por d, admitirá uma solução em números inteiros. Somos mesmo capazes de encontrá-la. De facto, seja $c = c'\,d$ e sejam s e t dois números inteiros tais que

$$d = a\,s + b\,t.$$

Então

$$c = c'\,d = a\,\left(s\,c'\right) + b\,\left(t\,c'\right),$$

isto é, $x_0 = s\,c'$ e $y_0 = t\,c'$ constituem uma solução da equação (ED).

Resolvamos, a título de exemplo, a equação diofantina

$$33 = 858\,x + 253\,y.$$

Mais acima, mostrámos que

$$11 = 858 \cdot (-5) + 253 \cdot 17,$$

11 sendo o $m.d.c.$ de 858 e 253. Multiplicando a última igualdade por 3, obteremos

$$33 = 858 \cdot (-15) + 253 \cdot 51.$$

Logo, $x = -15$ e $y = 51$ é uma solução da equação $33 = 858\,x + 253\,y$. Seria, porém, erróneo pensar que esta solução é única. Mesmo mais. Constata-se que qualquer equação diofantina da forma (ED), que admitir uma solução, admitirá uma infinidade delas. A demonstração desta última afirmação sai no entanto fora do âmbito deste trabalho e deixamos a sua análise à consideração do leitor curioso.

3. Tarefa

- Implemente, usando o algoritmo de Euclides acima descrito, o predicado

  `mdc(N1,N2,M,L)`

  onde `M` é o máximo divisor comum de `N1` e `N2` e `L` é uma lista de listas com a informação de todas as etapas do algoritmo. Cada etapa, que consiste numa divisão `D1 = Q * D2 + R <=> R = D1 - Q * D2`, será representada pela lista `[R,[-Q,D2],[1,D1]]`.
- Implemente o predicado

  `diofantina(A,B,C,X,Y),`

  que recebe três inteiros `A`, `B` e `C` e retorna uma solução `X,Y` (caso exista) da equação diofantina `A*X + B*Y = C`, pelo método acima descrito.

4. Os Dados

Neste problema não existem outros dados para além do conhecimento acima descrito.

5. Os Resultados

Os predicados `mdc(N1,N2,M,L)` e `diofantina(A,B,C,X,Y)` devem produzir resultados consonantes com os exemplos que se seguem.

Exemplo "mdc"

```
?-mdc(858,253,M,L).
M =11
L = [ [ 11,[-1,44],[1,55] ],
      [ 44,[-1,55],[1,99] ],
      [ 55,[-2,99],[1,253] ],
      [ 99,[-3,253],[1,858] ]
    ]
```

Exemplos "diofantina"

```
?-diofantina(434,233,3,X,Y).
X =-153
Y = 285

?-diofantina(858,253,33,X,Y).
X = -15
Y =  51

?-diofantina(15,9,7,X,Y).
No
```

6. Uma solução

```
% Solução para o problema DIOFANTO
% por Delfim F. Marado Torres, Fev. 1999


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                mdc(N1,N2,M,L)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% M é o máximo divisor comum
% de N1 e N2 (supomos ambos estritamente positivos com N1 > N2)
%
% O método usado para obter M
% é o algoritmo de Euclides
%
% L é uma lista de listas com a informação
% de todas as etapas do algoritmo. Cada
% etapa consiste numa divisão
%        D1 = Q * D2 + R <=> R = D1 - Q * D2
% que será representada pela lista
%        [R,[-Q,D2],[1,D1]]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

mdc(N1,N2,M,L) :-
  integer(N1),
  integer(N2),
  N1 > N2,
  N2 > 0,
  euclides(N1,N2,M,L).


euclides(N1,N2,N2,[]) :-
    etapa(N1,N2,_,0).
```

```
euclides(N1,N2,M,L) :-
    etapa(N1,N2,Q,R),
    euclides(N2,R,M,L1),
    concatena(L1,[[R,[-Q,N2],[1,N1]]],L).

etapa(D1,D2,Q,R) :-
  Q is D1 // D2,
  R is D1 mod D2.

concatena([],L,L).
concatena([X|R],L,[X|C]) :-
  concatena(R,L,C).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                diofantina(A,B,C,X,Y)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Dado A, B e C, pretendemos resolver uma equação diofantina da
% forma
%                A*X + B*Y = C              (supomos A > B)
%
% Se B = mdc(A,B) Entao se B divide C temos solução trivial:
%                X = 0, Y = C/B
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                 EXEMPLOS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ?- diofantina(434,233,3,X,Y).
%%    X = -153
%%    Y = 285
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ?- diofantina(858,253,33,X,Y).
%%    X = -15
%%    Y = 51
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ?- diofantina(15,9,7,X,Y).
%%    No
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


diofantina(A,B,C,X,Y) :-
  A > B,
  mdc(A,B,M,L), !,
  0 is C mod M,
  (
   M = B, L = [], X = 0, Y is C // B ;
   expande(L,[M,[Y1,B],[X1,A]]), Mlt is C // M, X is X1*Mlt, Y is Y1*Mlt
  ).


expande([L],L).
expande([[A1,[B1,_],[D1,E1]],[_,[B2,E1],[D2,E2]]|R],L) :-
        B is B1 * B2 + D1,
        D is B1 * D2,
        expande([[A1,[B,E1],[D,E2]]|R],L).
```

16
Guião para a Sétima Aula

1. Objectivos da aula

São dois os objectivos desta aula:

- aprender a aceder a ficheiros em Prolog;
- resolver (com um grafo) o problema PARENTES proposto no CeNPL'99.

2. Acesso a ficheiros

Introdução.

Até agora, o meio de comunicação entre o utilizador e os nossos programas Prolog têm sido as questões que colocamos ao interpretador e as respostas que ele nos dá por instanciação de variáveis ou por mensagens escritas no ecrã (pelo `write` ou `put`). Vamos agora ver um outro método de comunicação: uso de ficheiros para entrada e saída de informação e não apenas o teclado e o ecrã.

Muitos dos predicados de acesso a ficheiros dependem do interpretador Prolog usado. No seguimento da filosofia do nosso curso, limitamo-nos aqui aos predicados "standard".

Filosofia.

No Prolog, tal como no C, os dados introduzidos pelo utilizador via teclado e os dados de saída que aparecem no ecrã, são tratados como casos particulares, respectivamente, de dados provenientes de ficheiros de entrada e de dados a escrever em ficheiros de saída. Estes pseudo ficheiros (teclado e ecrã) são referidos (ambos) por `user`. O nome dos outros ficheiros podem ser escolhidos pelo programador.

Em cada instante, o Prolog tem (apenas) dois ficheiros activos: um para entrada (input) e outro para saída (output). Estes dois ficheiros são designados por *ficheiro de entrada corrente* e *ficheiro de saída corrente*. Por defeito, o ficheiro de entrada corrente é o teclado e o ficheiro de saída corrente é o ecrã.

Manipulação de ficheiros.

O ficheiro de entrada corrente pode ser mudado para um ficheiro de nome `FichL` com o predicado `see/1`:

```
see(FichL)
```

A menos que ocorra algo de errado com o sistema operativo, a chamada ao átomo `see(FichL)` tem sucesso e todo o `read` ou `get` posterior irá buscar a informação ao ficheiro de nome `FichL` em vez do habitual teclado.

De um modo geral, queremos ler algo de um ficheiro e no fim tornar a colocar o teclado como o ficheiro de entrada corrente. Assim sendo, um exemplo típico é:

```
...
see(fich1),
le_do_ficheiro(Informacao), % predicado vosso
```

```
seen,
see(user),
...
```

O comando `seen` serve para fechar o ficheiro de leitura corrente.

O ficheiro de saída corrente pode ser mudado para um ficheiro de nome `FichE` com o predicado `tell/1`:

```
tell(FichE)
```

Uma sequência típica é:

```
...
tell(fich2),
escreve_no_ficheiro(Informacao), % predicado vosso
told,
tell(user),
...
```

O comando `told` fecha o ficheiro de escrita corrente.

Todos os ficheiros são ficheiros de texto. Estes ficheiros são processados sequencialmente. Significa isto que depois de lermos uma informação, não podemos voltar atrás e tornar a lê-la; ou escrever algo e voltar atrás (o comportamento é idêntico ao do ecrã).

Após o comando `see` a *posição corrente* coincide com o início do ficheiro. Depois de uma leitura, a posição corrente é movida para o próximo item não lido, de maneira que a próxima leitura começará nesta nova posição. Se fizermos um pedido de leitura e a posição corrente coincidir com o fim do ficheiro, então a informação retornada por tal pedido será a constante `end_of_file`.

Com a escrita passa-se o equivalente. Logo após o comando `tell`, a posição corrente coincide com o início do ficheiro. A cada pedido de escrita corresponde o acrescento dessa informação a seguir à posição corrente e a mudança da posição para o final da informação no ficheiro. Não é possível mexer a posição corrente para trás e sobrepor parte do ficheiro.

Dependendo da forma da informação, podemos olhar para os ficheiros de duas maneiras. Uma consiste em olhar para o caracter como o elemento básico do ficheiro. Nesse caso usamos os predicados `get` (ou `get0`) e `put` para, respectivamente, ler e escrever um (único) caracter. Estes predicados lidam com códigos ASCII. Um exemplo:

```
?- put(65), put(66), put(67).
   ABC
   yes
```

A diferença entre o `get` e `get0` é que o primeiro despreza os espaços brancos (na verdade todos os caracteres não visíveis) enquanto o segundo não.

A segunda maneira consiste em considerar unidades de informação maiores (por exemplo cláusulas) como as unidades de informação básicas do ficheiro. Neste caso, um pedido de leitura ou escrita corresponderá à transferência da cláusula completa do ficheiro de entrada corrente ou para o ficheiro de saída corrente. Os predicados de leitura e escrita são neste caso o `read` e o `write`.

Exercício.

Suponhamos que temos um ficheiro constituído por factos da forma

```
item(NumeroDoItem, Descricao, Preco, Fornecedor).
```

Pretendemos produzir um outro ficheiro que contém apenas os itens de um dado fornecedor. Como o fornecedor, neste novo ficheiro, será sempre o mesmo, basta que o seu nome seja escrito apenas uma vez, no início do ficheiro, e omitido nos factos seguintes.

Se `camaras.dat` contiver a seguinte informação

```
item(3122800,'QV-7000SX',161,casio).
item(9991700,'PhotoPC',130,epson).
item(3421122,'MX-500',100,fuji).
item(3421132,'MX-700',145,fuji).
```

```
item(1112233,'Q-M100V',130,konica).
```

e introduzirmos no interpretador

```
?- see('camaras.dat'), tell('fuji.dat'),
   selecciona(fuji), seen, told,
   see(user), tell(user).
yes
```

queremos obter o ficheiro `fuji.dat` com a seguinte informação:

```
fornecedor(fuji).
item(3421122, MX-500, 100).
item(3421132, MX-700, 145).
```

Uma resolução

```
% Delfim F. Marado Torres
% Testado no SWI-Prolog em 4/Maio/1999

selecciona(Fornecedor) :-
  write(fornecedor(Fornecedor)), write('.'), nl,
  sel(Fornecedor).

sel(F) :-
  read(Item),
  processa(Item,F).

processa(end_of_file,_).
processa(item(N,D,P,F),F):- % o Item lido e' do fornecedor desejado
  write(item(N,D,P)),
  write('.'), nl,
  sel(F).
processa(_,F) :-       % o Item lido n~ao e' do fornecedor desejado
  sel(F).
```

3. Parentes

Árvore Genealógica

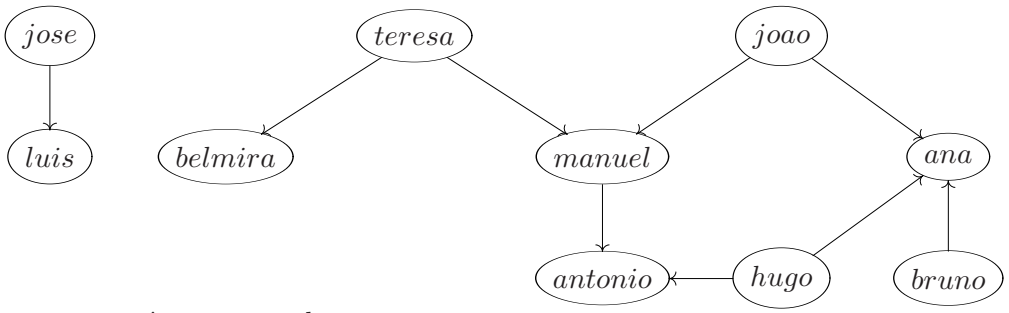
`Familias` é um grafo orientado não-pesado em que os vértices (ou nodos) representam indivíduos (identificados por um nome único) e a relação binária que define os ramos (ou arestas) do grafo é `eFilhoDe`. Assim, se os indivíduos `I1` e `I2` formam um ramo (o par `(I1,I2)` pertence à relação), quer dizer que `I1` é filho de `I2`.

Tarefa

A sua tarefa é escrever um programa Prolog que, usando `Familias`, seja capaz de verificar se há laços de sangue entre dois indivíduos dados, isto é se eles são parentes.
Além de verificar a *consanguinidade* entre os indivíduos, interessa também detectar a proximidade ou afastamento dessa ligação, determinando o grau de parentesco entre ambos. Para esse efeito, considera-se o *grau de parentesco* entre dois indivíduos como sendo a *soma das distâncias de cada um ao parente comum mais próximo*; note, para isso, que a distância de um filho ao seu progenitor é igual a 1.

Os Dados

Uma instância do grafo `Familias`, pode ser, por exemplo,



que será representado como:

```
eFilhoDe(teresa,belmira).
eFilhoDe(teresa,manuel).
eFilhoDe(joao,manuel).
eFilhoDe(joao,ana).
eFilhoDe(bruno,ana).
eFilhoDe(manuel,antonio).
eFilhoDe(hugo,antonio).
eFilhoDe(hugo,ana).
eFilhoDe(jose,luis).
```

Os Resultados

O programa deve ser chamado através do seguinte predicado:

```
consanguineos( Ind1,Ind2,Grau ).
```

em que se supõe instanciados `Ind1` e `Ind2` com nomes de indivíduos que figuram em `Familias` como vértices; o resultado virá em `Grau`, que será igual a 0 (zero) se não houver qualquer laço de sangue entre os dois, ou então indicará o grau de parentesco entre eles.

Exemplos

Para a base de conhecimento acima, temos

```
?- consanguineos(joao,hugo,G).
   G = 2
?- consanguineos(bruno,teresa,G).
   G = 0
```

Uma resolução

```
% Delfim F. Marado Torres; 5/Maio/1999

consanguineos(I1,I2,G) :-
  vertice(I1),
  vertice(I2),
  I1 \= I2,
  dist_todos_caminhos(I1,I2,Lista),
  menor(Lista,G).
consanguineos(_,_,0).

vertice(X) :- eFilhoDe(X,_).
vertice(X) :- eFilhoDe(_,X).

dist_todos_caminhos(I1,I2,L) :-
  findall(
    G,
    ( vertice(X), X \= I1, X \= I2,
      caminho(I1,X,D1), caminho(I2,X,D2), G is D1+D2
    ),
```

```
    L
  ).

menor([X],X).
menor([X|R],Z) :-
  menor(R,Y),
  (X =< Y, !, Z = X ; Z = Y).

caminho(X,Y,D) :- cam(X,Y,D,[]).

cam(X,Y,1,_) :- eFilhoDe(X,Y).
cam(X,Y,D,L) :-
  eFilhoDe(X,Z),
  not member(Z,L),
  cam(Z,Y,D1,[Z|L]),
  D is D1 + 1.
```

17
Guião para a Oitava Aula

Codifica

1. Objectivos da aula

Resolução do problema Codifica proposto no CNPL'98.

2. Codificador Primo

Consideremos os números naturais e a ideia muito simples de número primo: um número que não se pode obter pela multiplicação de dois números mais pequenos onde não se inclui a unidade. Sabia-se desde tempos remotos, como um facto empírico, que todos os números se exprimem, de forma única, como um produto de primos; e os Gregos conseguiram prová-lo. Algumas perguntas sobre primos surgem naturalmente. Por exemplo: Quantos primos existem e como estão os primos distribuídos entre os outros naturais? Como se determinam primos? Como é que se mostra que um dado natural é, ou não, um primo? Como se factoriza, num produto de primos, um natural arbitrário? Mesmo com tão simples matéria-prima, a matemática fez uma obra espantosa e, se questões como as acima colocadas, podem ser olhadas numa perspectiva abstracta e como um ramo da matemática inaplicável, a verdade é que estas questões têm um significado crucial para a criptografia; tanto mais que têm havido sérias tentativas de classificar alguns dos resultados desta área como segredos militares. Tudo indica que são os métodos de codificação baseados em números primos que constituirão a parte central do sistema de segurança da futura auto-estrada da informação.

3. Tarefa

Implemente em Prolog um codificador de mensagens e o respectivo descodificador. O codificador deve substituir o código ASCII de cada caracter da mensagem por um outro de acordo com o algoritmo de encriptação descrito pelos seguintes passos:

- Converter a mensagem dada numa lista L com os códigos ASCII de cada caracter.[1]
- Gerar a lista P de todos os primos até ao código ASCII do primeiro caracter da frase a codificar (1^o elemento de L), usando um predicado `primos/2` que terá de desenvolver segundo o método a indicar mais adiante.
- O primeiro caracter da frase não é alterado.
- Ao segundo caracter somamos o primeiro número primo em P; ao terceiro caracter somamos o segundo elemento de P, etc, considerando a lista P de um modo circular —depois de somar o último primo em P, voltar ao início da lista.

[1] Sugere-se o uso do predicado pré-definido `name/2`.

Lembramos que o código ASCII é um número entre 0 e 255 e que é preciso ter o cuidado de garantir que os códigos ASCII codificados também estão nesta gama! Mais uma vez, deve usar a noção de circularidade: por exemplo, se estamos a somar o primo 3, o código ASCII 33 será transformado no 36 enquanto o 254 será transformado no 1.

O algoritmo de descodificação deverá proceder à operação inversa do de codificação.

3.1. Subtarefa. Implemente um predicado em Prolog para gerar números primos segundo o algoritmo designado por *Crivo de Eratóstenes*. Esse método permite-nos construir uma lista de todos os primos até um limite dado, de acordo com a seguinte descrição[2]:

> *"Escrevemos os naturais desde* 2 *até ao limite desejado; por exemplo* 200*:*
>
> 2, 3, 4, 5, 6, 7, . . . , 198, 199, 200.
>
> *Começando no princípio da lista, o primeiro número que encontramos é* 2*, um primo. Deixamos* 2 *de lado, e passamos à frente marcando os números de* 2 *em* 2*, isto é, marcamos* 4, 6, 8, 10, ···*. Depois de ter marcado todos os números pares até* 200*, voltamos ao princípio da lista para encontrar o primeiro número depois de* 2 *que não foi marcado: é* 3*. Deixamos* 3 *de lado (é primo), e passamos à frente marcando os números de* 3 *em* 3*: marcamos* 6, 9, 12, 15, ···*. Prosseguimos assim. Depois de fazermos este jogo repetidamente para* 2, 3, . . . , N*, onde* N *é o maior natural não marcado tal que* $N \leq \sqrt{200}$*, então os números da lista que ainda não foram marcados são os primos até* 200[3]*."*

4. Os Dados

O único dado necessário para a resolução da tarefa principal deste problema é a mensagem a codificar. Essa mensagem será um parâmetro do predicado `encode/2` que tem de desenvolver, como se explica na secção seguinte.

Para a subtarefa de geração de números primos também precisa de conhecer à partida apenas um dado: o valor do limite superior da lista a gerar.

5. Os Resultados

O seu programa deve ser invocado através dos predicados:

a): `encode/2`
que recebe a mensagem e retorna no segundo argumento a lista dos códigos ASCII da mensagem codificada

b): `decode/2`
que recebe a lista dos códigos ASCII da mensagem codificada e devolve no segundo argumento a mensagem original.

c): `primos/2`
que recebe como primeiro argumento um número natural N e devolve no segundo argumento a lista de todos os primos até N.

Exemplos:

```
?- primos(100,L).
  L = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,
       67,71,73,79,83,89,97]

?- encode('Ola malta',C), decode(C,D).

  C = [79,110,100,37,116,108,121,133,116]
  D = 'Ola malta'
```

[2] B. Owen, *Teoria Elementar dos Números I*, Boletim da SPM, 33, (1995), p. 17-36.

[3] Sugerimos que evitem o cálculo da raiz quadrada. Na situação apresentada, o teste $N \leq \sqrt{200}$ pode ser substituído por $N^2 \leq 200$.

6. Uma solução

```
%% Realizado por Delfim F. Marado Torres -- 30/Mar,co/1998
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% primos/2
%% primos(N,L)  - L 'e a lista de todos os primos =< N
%%
%% Exemplo:
%%    ?- primos(200,L).
%%    L = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,
%%         89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,
%%         179,181,191,193,197,199]
%%    yes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

primos(N,P) :-
  cria_lista2N(N,L),
  eratostenes(N,L,[],[],PI), % No in'icio a lista de marcados 'e vazia
  inverte(PI,P),!.           % assim como a de primos

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados auxiliares normalmente disponibilizados
%% nos interpretadores de Prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

membro(X,[X|_]).
membro(X,[_|R]) :-
  membro(X,R).

concatena([],L,L).
concatena([X|R],L,[X|C]) :-
  concatena(R,L,C).

nao_e_verdade(P) :- call(P), !, fail ; true.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% cria_lista2N/2
%% cria_lista2N(N,L)  -  L 'e a lista de naturais de 2 at'e N
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

c_lista(2,[2]).
c_lista(N,[N|R]) :-
  N1 is N-1,
  c_lista(N1,R).

inverte([],[]).
inverte([X|R],I) :-
  inverte(R,RI),
  concatena(RI,[X],I).

cria_lista2N(N,[]) :-
  N < 2.
cria_lista2N(N,L) :-
  c_lista(N,I),
  inverte(I,L).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% eratostenes/5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

eratostenes(_,[],_,P,P).
eratostenes(N,[X|L],M,P,R) :-
  membro(X,M),
  eratostenes(N,L,M,P,R).
eratostenes(N,[X|L],M,P,R) :-
  X2 is X * X,
  X2 =< N,
  marca(X,X,L,M1),
  concatena(M,M1,NM),
  eratostenes(N,L,NM,[X|P],R).
eratostenes(N,[X|L],M,P,R) :-
  eratostenes(N,L,M,[X|P],R).

marca(_,I,L,[]) :-
  nao_e_verdade(esimo_elemento(I,L,_)).
marca(N,I,L,[X|M]) :-
  esimo_elemento(I,L,X),
  NI is I + N,
  marca(N,NI,L,M).

esimo_elemento(1,[X|_],X).
esimo_elemento(N,[_|R],X) :-
  N1 is N-1,
  esimo_elemento(N1,R,X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% encode/2  - codifica strings
%% decode/2  - descodifica as strings codificadas com encode/2
%%
%% Exemplo:
%%          ?- encode('Encontra-me ao anoitecer',C), decode(C,D).
%%
%%          D = 'Encontra-me ao anoitecer'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

encode(M,C) :-
  name(M,[X|L]),
  primos(X,P),
  codifica(L,P,C).


decode([X|L],M) :-
  primos(X,P),
  descodifica(L,P,LD),
  name(M,[X|LD]).


cod(N,X,Y) :-   Y is (X + N) mod 256.
dec(N,Y,X) :-   X is (256 + Y - N) mod 256.

codifica(L,P,LC) :- transforma(cod,L,P,LC).
```

```
descodifica(L,P,LD) :-
  transforma(dec,L,P,LD).

transforma(_,[],_,[]).
transforma(T,[X|R],[P1|P],[XT|RT]) :-
  C =.. [T,P1,X,XT],
  call(C),
  concatena(P,[P1],NP),
  transforma(T,R,NP,RT).
```

7. Outra solução

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Realizado por Delfim F. Marado Torres
% 13/Maio/1999
% Testado no SWI-Prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

primos(N,P) :-           % N inteiro >= 2
  listaAte(L,N),
  eratostenes(L,[],P).   % ListaMarcados = []

listaAte([2],2).
listaAte(L,N) :-
  N1 is N-1,
  listaAte(R,N1),
  append(R,[N],L).

eratostenes([],_,[]).    % Aplicamos o processo ate' 'a lista vazia
                         % Nao aplicamos o criterio =< sqrt(N)
eratostenes([X|R],M,P) :-
  member(X,M),
  eratostenes(R,M,P).
eratostenes([X|R],M,[X|P]) :-
  marca(X,X,[X|R],MX),
  append(MX,M,NM),       % nao faz mal haver repetidos
  eratostenes(R,NM,P).

marca(_,_,[],[]).
marca(X,0,[Y|R],[Y|M]) :-
  marca(X,X,[Y|R],M).
marca(X,P,[_|R],M) :-
  P1 is P-1,
  marca(X,P1,R,M).

%%%%%%%%%%%%%%%%%%%%%%%%%%

encode(M,L) :-
  name(M,L1),
  transforma(soma,L1,L).

decode(L,M) :-
  transforma(subtrai,L,L1),
  name(M,L1).
```

```
transforma(T,[X|R],[X|L]) :-
  primos(X,P),
  codifica(T,R,P,L).

codifica(_,[],_,[]).
codifica(T,[X|R],[A|RA],[C|RC]) :-
  G =.. [T,X,A,C],
  call(G),
  append(RA,[A],P),
  codifica(T,R,P,RC).

soma(X,Y,R) :-
  R is (X + Y) mod 256.

subtrai(X,Y,R) :-
  R is (X - Y + 256) mod 256.
```

18
Guião para a Nona Aula

Comandos

1. Objectivos da aula

Resolução do problema Comandos proposto no CeNPL'99.

2. Controlo à Distância

Um aparelho usado para controlar à distância o funcionamento de uma aparelhagem electrónica de som, com sintonizador AM/FM, leitor de CDs e leitor de cassetes, aceita comandos para: ligar e desligar (ON/OFF) a aparelhagem; escolher a modalidade que se pretende ouvir (entre os 3 dispositivos existentes). Após aceitar um comando, o controlador à distância envia-o para o aparelho (processo esse que é irrelevante neste contexto) e continua a aceitar novos comandos.
O comando para desligar a aparelhagem pode ser emitido em qualquer momento, após se ter ligado o equipamento.
Só faz sentido seleccionar um modo de funcionamento depois de se ter ligado a aparelhagem. Além disso, só são *úteis* as *sequências de comandos* que verifiquem as seguintes indicações: no caso de se escolher o sintonizador, deve indicar-se a seguir a banda (AM ou FM) e a frequência do canal a ouvir; se a escolha for o leitor de CDs, indicar-se-á apenas o número da pista; finalmente, optando-se pelo leitor de cassetes, indicar-se-á a operação desejada (FF, RW, Play), ou Stop depois de uma das outras.
Note que a introdução de uma *sequência inválida de comandos* faz com que a aparelhagem seja imediatamente (logo que se detecte o erro) desligada.

3. Tarefa

A sua tarefa é escrever um programa Prolog que dada uma sequência de comandos indique, caso a sequência seja válida, o estado em que o aparelho fica a funcionar.
Para resolver o problema, considere que o controlador à distância pode ser encarado como uma Máquina de Transição de Estados, e então implemente o *autómato determinista* que modela o sistema.

4. Os Dados

Para representar cada comando é usado o seguinte alfabeto (conjunto de símbolos terminais):

```
{ on, off,
  sint, lcd, lcasset,
  am, fm, freq, pista, ff, rw, play, stop }
```

5. Os Resultados

O programa deve ser chamado através do seguinte predicado:

```
simulaControlador( Cmds,Estado ).
```

em que se supõe instanciado `Cmds`, com uma das sequências definidas em `comandos/1`; os resultados virão em `Estado`.

Exemplos:

```
?- simulaControlador([on],E).

E = 'ligado, espera comandos' ;
no

?- simulaControlador([on,freq,off],E).

E = 'desligado apos erro, sequencia cmds inutil' ;
no

?- simulaControlador([on,lcasset,rw],E).

E = 'leitor de cassetes a puxar a fita para tras' ;
no

?- simulaControlador([on,lcasset,stop],E).

E = 'desligado apos erro, sequencia cmds inutil' ;
no
```

6. Uma solução

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Solucao para o problema COMANDOS proposto no CeNPL'99
% Realizado em 19/Maio/1999 por Delfim F. Marado Torres
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Descricao da 'aparelhagem'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

simbolosEntrada([on,off,sint,lcd,lcasset,am,fm,freq,pista,ff,rw,play,stop]).

estadoInicial(0).

delta(0,on,1).
delta(1,sint,2).
delta(2,am,3).
delta(2,fm,4).
delta(3,freq,'5a').
delta(4,freq,'5b').
delta(1,lcd,6).
delta(6,pista,7).
delta(1,lcasset,8).
delta(8,ff,9).
delta(8,rw,10).
delta(8,play,11).
delta(E,stop,8) :-
  member(E,[9,10,11]).
```

```
delta('0e',off,'0e').
delta(_,off,0).
delta('0e',on,1).
delta(_,C,'0e') :-
  simbolosEntrada(L),
  member(C,L).

descreveEstado(0,'aparelho desligado').
descreveEstado(1,'ligado, espera comandos').
descreveEstado(2,'sintonizador escolhido, espera seleccao da banda').
descreveEstado(3,'banda AM escolhida, espera seleccao da frequencia').
descreveEstado(4,'banda FM escolhida, espera seleccao da frequencia').
descreveEstado('5a','a ouvir radio, banda AM').
descreveEstado('5b','a ouvir radio, banda FM').
descreveEstado(6,'leitor de CDs escolhido, espera seleccao da pista').
descreveEstado(7,'a ouvir CD').
descreveEstado(8,'leitor de cassetes escolhido,
                  espera seleccao da operacao desejada').
descreveEstado(9,'leitor de cassetes a puxar a fita para a frente').
descreveEstado(10,'leitor de cassetes a puxar a fita para tras').
descreveEstado(11,'a ouvir cassete').
descreveEstado('0e','desligado apos erro, sequencia cmds inutil').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% simulaControlador(Cmds,Estado)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

simulaControlador(Cmds,DescricaoEstado) :-
  estadoInicial(EI),
  simula(Cmds,EI,DescricaoEstado).

simula([],EstCor,Descricao) :-
  descreveEstado(EstCor,Descricao).
simula([C|R],EstCor,Descricao) :-
  delta(EstCor,C,ProxEst),!,        % usamos o cut, pois trata-se
  simula(R,ProxEst,Descricao).      % de um automato determinista
```

19
Guião para a Décima Aula

1. Objectivos da aula

São objectivos da aula:

- Resolver o problema das oito rainhas.
- Implementar, usando uma gramática (DCG), um interface em língua natural que permita interagir (acrescentar e consultar informação) com uma base de conhecimento (BC) com predicados `profissao/2` e `curso/2`.

2. O problema das 8 rainhas

2.1. Enunciado e estratégia de resolução. O problema das 8 rainhas é um problema bem conhecido, muitas vezes usado para introduzir a ideia de retrocesso (backtracking). Consiste em colocar 8 rainhas num tabuleiro de xadrez (8×8) de tal modo que nenhuma rainha ataque outra. A abordagem habitual, para resolver este problema, consiste numa pesquisa exaustiva de todas as possibilidades de colocação das rainhas. Numerando as linhas e as colunas de 1 a 8, então cada uma dessas possibilidades pode ser representada por um óctuplo

$$[[1, c_1], \ldots, [8, c_8]]\,,$$

onde $[i, c_i]$ denota a posição de uma rainha no quadrado da linha i e coluna c_i. Podemos representar esta configuração apenas por

$$[c_1, \ldots, c_8]\,,$$

convencionando que a i-esima posição denota, implicitamente, a linha i. Além disso, como duas rainhas na mesma coluna atacam-se mutuamente, apenas precisamos de considerar os óctuplos $[c_1, \ldots, c_8]$ que sejam uma permutação de [1, 2, 3, 4, 5, 6, 7, 8]. Para resolver este problema temos então, essencialmente, de:

- achar uma permutação de [1, 2, 3, 4, 5, 6, 7, 8];
- verificar se essa permutação é uma solução.

2.2. Uma solução.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Problema das 8 rainhas
% Delfim F. Marado Torres, 24/Maio/1999
%
% No livro do Bratko, pp. 108--118,
% pode encontrar outras resolucoes
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicado principal: rainhas(Solucao)
%
% Solucao sera' instanciada com uma
% lista da forma [ [1,Y1], ..., [8,Y8] ]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rainhas(S) :-
  permutacao([1,2,3,4,5,6,7,8],P),
  formaPares(P,S),
  naoAtaca(S).

formaPares(L,LP) :-
  fp(1,L,LP).

fp(_,[],[]).
fp(N,[X|R],[[N,X]|RP]) :-
  N1 is N + 1,
  fp(N1,R,RP).

naoAtaca([]).
naoAtaca([P|RP]) :-
  naoAtacaNenhuma(P,RP),
  naoAtaca(RP).

naoAtacaNenhuma(_,[]).
naoAtacaNenhuma([X,Y],[[X1,Y1]|RP]) :-
  not emLinha([X,Y],[X1,Y1]),
  naoAtacaNenhuma([X,Y],RP).

emLinha([X,_],[X,_]).
emLinha([_,Y],[_,Y]).
emLinha([X1,Y1],[X2,Y2]) :-
  D is abs(Y2 - Y1),
  D is abs(X2 - X1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicados ja realizados anteriormente
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

permutacao([],[]).
permutacao([X|R],P) :-
  permutacao(R,RP),
  adiciona(X,RP,P).

adiciona(X,R,[X|R]).
adiciona(X,[Y|R],[Y|A]) :-
  adiciona(X,R,A).
```

2.3. Uma curiosidade. Sabe quantas soluções diferentes existem para o problema das 8 rainhas? Nada mais nada menos que 92! Tal como eu, pode descobrir isso com o comando

```
?- tell('rainhas.txt'), rainhas(S), write(S), nl, fail.

No
told.   % escrevi eu

Yes
```

```
?-
```

dando depois uma vista de olhos ao ficheiro `rainhas.txt`.

3. Gramática 'BC'

3.1. O pretendido. Pretende-se construir, e posteriormente consultar, uma Base de Conhecimento em Prolog com predicados `profissao/2` e `curso/2`

```
profissao(Pessoa,Actividade).
curso(Pessoa,Curso).
```

através de um interface que use a língua portuguesa. Eis alguns exemplos do funcionamento pretendido:

- Se dissermos "O xico é fotógrafo.", então será acrescentado (automaticamente, por meio de um `assertz`) à BC o facto

  ```
  profissao(xico,fotografo).
  ```

- Se afirmarmos "A Zé estuda matemática.", então será acrescentado à BC o facto

  ```
  curso(ze,matematica).
  ```

- Se, depois das afirmações anteriores, perguntarmos "O que faz o xico?", obteremos como resposta "xico é fotógrafo.".
- Se perguntarmos "O que faz a Zé?", obteremos como resposta "Zé estuda matemática.".

Para o sistema Prolog reconhecer as frases acima e fazer, sozinho, as respectivas acções (acrescentar ou consultar conhecimento) vamos recorrer a uma DCG.

3.2. Uma solução.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Gramatica 'BC'
%%
%% Delfim F. Marado Torres
%% 25/Maio/1999
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gramBC -->
  afirmacao(Tipo,Pess,Act),
  { acrescentaBC(Tipo,Pess,Act) }.
gramBC -->
  questao(Pess),
  { daActividade(Pess) }.

afirmacao(profissao,P,A) -->
  det(_),
  [P, e, A, '.'].
afirmacao(curso,P,A) -->
  det(_),
  [P, estuda, A, '.'].

questao(Pess) -->
  det(masculino),
  [que, faz],
  det(_),
  [Pess, '?'].

det(masculino) --> [o] | ['O'].
det(feminino) -->  [a] | ['A'].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Accoes Semanticas da Gramatica
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
acrescentaBC(Tipo,Pess,Act) :-
  C =..[Tipo,Pess,Act],
  assertz(C),
  write('A clausula '),
  write(C),
  write(' foi adicionada com sucesso.'),
  nl.

daActividade(P) :-
  profissao(P,A),
  informa(' e ',P,A).
daActividade(P) :-
  curso(P,A),
  informa(' estuda ',P,A).
daActividade(P) :-
  write(P),
  write(' nao consta da BC!'),
  nl.

informa(T,P,A) :-
  write(P),
  write(T),
  write(A),
  write('.'),
  nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicado Principal: bc/1
%%
%% Recebe uma string em vez de uma lista
%%
%% EXEMPLOS:
%%
%% ?- bc('O xico e fotografo.').
%% A clausula profissao(xico,fotografo) foi adicionada com sucesso.
%%
%% ?- bc('A ze estuda matematica.').
%% A clausula curso(ze,matematica) foi adicionada com sucesso.
%%
%% ?- bc('O que faz o xico?').
%% xico e fotografo.
%% ?- bc('O que faz a ze?').
%% ze estuda matematica.
%%
%% ?- bc('O que faz o delfim?').
%% delfim  nao consta da BC!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bc(String) :-
  converte(String,Lista),!,
  gramBC(Lista,[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predicado ja' usado na teorica.
%% Transforma uma string numa lista de palavras.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
converte(S,L) :-
  name(S,LA1),
  adicionaEspacos(LA1,LA2),
  conv(LA2,L).

adicionaEspacos([],[]).
adicionaEspacos([X|R1],[32,X,32|R2]) :-
  member(X,[58,59,44,46,33,63]),
  adicionaEspacos(R1,R2).
adicionaEspacos([X|R1],[X|R2]) :-
  adicionaEspacos(R1,R2).

conv([],[]).
conv(LA,[P|R]) :-
  palavra(LA,PA,RA),
  name(P,PA),
  conv(RA,R).

palavra([],[],[]).
palavra([32|R],[],L) :-
  desprezaEspacos(R,L).
palavra([X|R1],[X|R2],L) :-
  palavra(R1,R2,L).

desprezaEspacos([32|R1],R) :-
  desprezaEspacos(R1,R).
desprezaEspacos(L,L).
```

20

Guião para a Décima Primeira Aula

Torres de Hanoi

1. Objectivos da aula

Resolução do problema Torres de Hanoi.

2. Enunciado do Problema

Algures perto de Hanoi, existe um mosteiro cujos monges devotam as suas vidas a uma tarefa muito importante. No pátio desse mosteiro existem três postes enormes. Nestes postes existe um conjunto de sessenta e quatro discos, cada um deles com um orifício no centro e com um raio diferente. Quando o mosteiro foi construído, todos os discos estavam num dos postes: cada disco descansando sobre o disco de raio imediatamente a seguir. A tarefa dos monges consiste em mover todos os discos para um dos outros postes. Apenas um disco pode ser movido de cada vez e todos os restantes discos devem estar num dos postes. Para além disso, em nenhum instante durante o processo, pode um disco estar colocado em cima de um disco menor. O terceiro poste pode, claro, ser usado como local temporário de descanso para os discos. Qual é a maneira mais rápida de os monges cumprirem a sua missão?

Mesmo a melhor solução para este problema, precisará de muito tempo para que os monges a consigam terminar. Ainda bem que assim é, pois a lenda conta que o mundo acabará quando os monges tiverem terminado a sua tarefa ...

3. Abordagem Recursiva

O problema é muito fácil de resolver se pensarmos de modo generalista e recursivo. Começamos por generalizar o problema da seguinte maneira.

Em vez de considerarmos um número fixo de discos, 64, consideramos o problema mais geral em que o número de discos é arbitrário: digamos N. A generalização de problemas é algo de muito importante e constitui um dos paradoxos da programação: muitas vezes a maneira mais fácil de resolver um problema consiste em resolver uma sua generalização!

Vamos agora resolver o problema generalizado de modo recursivo. O caso trivial acontece para $N = 0$: não há nenhum disco, não é preciso fazer nada. Para $N > 0$, temos no início os discos num dos postes, digamos no `posteA`, e no final pretendemos que eles estejam todos em outro, digamos, no `posteB`. A utilidade da recursividade está em percebermos que se soubermos resolver o problema para $N - 1$ discos, facilmente o resolvemos também para N discos:

- Movemos os $N - 1$ discos do `posteA` para o `posteC` (usando o `posteB` como auxiliar).

- Movemos o único disco que permanece no `posteA` (o de maior raio) para o `posteB`. Este disco fica na posição desejada. Nunca mais mexemos nele.
- Movemos os $N-1$ discos do `posteC` para o `posteB` (usando o `posteA` como auxiliar).

E pronto. Já temos todos os ingredientes para implementar uma solução em Prolog.

4. Uma solução

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Torres de Hanoi
% Delfim F. Marado Torres, 24/Maio/1999
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicado Principal: torresHanoi(N)
%
% N representa o numero de discos
% Os tres postes sao denotados por posteA, posteB e posteC
% No inicio todos os discos estao no posteA
% No fim pretendemos ter todos os discos no posteB
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

torresHanoi(N) :-
  move(N,posteA,posteB,posteC).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% move(NumDiscosAMover,Origem,Destino,Auxiliar)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

move(0,_,_,_).
move(N,A,B,C) :-
  N1 is N-1,
  move(N1,A,C,B),
  informa(A,B),
  move(N1,C,B,A).

informa(A,B) :-
  write('Move disco do '),
  write(A),
  write(' para o '),
  write(B),
  nl.
```

5. Comentário Final

Resolver o problema das Torres de Hanoi com um número de discos muito pequeno, por exemplo 3, pode ser facilmente feito "à mão" (com papel e lápis e esboçando a configuração dos postes no pátio dos monges). A solução é:

```
?- torresHanoi(3).
   Move disco do posteA para o posteB
   Move disco do posteA para o posteC
   Move disco do posteB para o posteC
   Move disco do posteA para o posteB
   Move disco do posteC para o posteA
   Move disco do posteC para o posteB
   Move disco do posteA para o posteB
```

Tentar fazer, no entanto, o mesmo para 10 discos ... A folha termina num emaranhado de riscos ... com grande probabilidade de nos 'perdermos' ... Se executarmos o comando

```
?- tell('hanoi10.txt'), torresHanoi(10), told, tell(user).
   Yes
```

e dermos uma vista de olhos ao ficheiro `hanoi10.txt`, concluímos que precisamos de 1023 operações `Move` para alcançarmos o desejado! O programa, no meu computador, apenas é capaz de ir até $N = 13$: para 13 discos a solução envolve 8191 operações `Move`!

Quantas operações serão necessárias antes 'do final dos tempos'? De um modo mais geral: é capaz de encontrar uma fórmula, em função do número `N` de discos, que nos dê o número de operações necessárias para o fim do jogo?

Enunciados de Trabalhos e Exames

21
Primeiros Trabalhos Práticos

1. Ano lectivo de 1998/1999 (10/Março/1999)

O trabalho deve ser executado em grupos de 2 alunos, e deve ser entregue a funcionar e acompanhado dum relatório dentro de **2 semanas**, dia 24 de Março.
Lembre-se que a nota prática global entrará na nota final com um peso de 50%.

2. Objectivos e Organização

Este trabalho prático tem como principais **objectivos**:

- iniciação à programação em Prolog, para criação de Bases de Conhecimento (BC) que permitam calcular a resposta a questões diversas que se possam inferir directamente das regras e factos contidos na BC, sem recursos a listas, nem outros termos estruturados;
- familiarização com o interpretador e ambiente de programação SWI-Prolog.

Para o efeito, esta folha contém quatro enunciados, dos quais deverá resolver **pelo menos um**. Esses enunciados são, propositadamente, deixados relativamente vagos por forma a respeitar a capacidade imaginativa e de organização de cada grupo e, até, para nos permitir avaliar soluções diferentes de grupo para grupo.

Quanto ao **relatório** a elaborar, que deverá ser entregar conjuntamente com a disquete do seu programa Prolog no dia dia 24 de Março, deve ser claro e sucinto e, além do respectivo enunciado e listagem do programa (apresentados em apêndice), deverá conter uma explicação dos predicados incluídos na BC, bem como exemplos de execução em que se mostrem os resultados produzidos para várias questões.

3. Enunciados

Questão 1 (docentes). Construa uma BC em que se registem diversos dados sobre os docentes do Departamento de Matemática —sua identificação (nome, telefone, extensão, email, http, ...), posição actual na Universidade (assistente estagiário, assistente, professor auxiliar, ...), o gabinete, as disciplinas que lecciona ou já leccionou (nome da disciplina, semestre, cursos a que é leccionada, ...), a área de investigação em que trabalha (grupo da UI&D a que pertence) e a indicação de outras responsabilidades que possui (membro da assembleia de representantes, comissões pedagógicas, ...).
O seu programa lógico deverá ser capaz de encontrar:

- os docentes pertencentes a uma determinada área de investigação;

- os *docentes "adequados"* para uma dada situação —p. ex., leccionar uma determinada cadeira, integrar um determinado projecto de investigação, fazer parte de um determinado pelouro ... ;
- responder a outras questões previstas pelo seu grupo!

Os critérios de *adequação* dos docentes às situações serão também definidos por cada grupo. Para a construção da BC, poderá encontrar toda a informação que necessita no servidor www do nosso departamento: http://www.mat.ua.pt

QUESTÃO 2 (universidade). A Universidade de Aveiro fez este ano lectivo 25 anos e, no âmbito das comemorações, resolveu promover o lançamento de um conjunto de estudos estatísticos sobre a universidade. Quando se começaram a realizar tais estudos, reparou-se que a informação disponível não era por vezes coerente ou induzia a erros que levavam, não muito raramente, a situações paradoxais. Estas situações decorriam do facto de que os sistemas de bases de dados disponíveis são pobres em termos de tratamento da integridade da informação e os humanos (as pessoas que introduzem os dados em computador) são dados a erros ... A identificação de todas as incongruências revelou-se difícil e, de modo garantir alguma validade científica aos trabalhos, a Universidade resolveu contrata-lo a si para desenvolver um programa em Prolog que permitisse ajudar a identificar essas incongruências. Caso aceite a missão ;-)
deve construir uma BC em que se registem os dados disponíveis referentes aos alunos (número de estudante, nome, curso, ano que frequenta ou a data em que terminou o curso, média, ...) e aos funcionários (código, nome, data de nascimento, morada, categoria profissional, departamento ou secção onde trabalha, ...). O seu programa lógico deverá ser capaz de ajudar a encontrar falhas nas restrições de integridade. Exemplos de restrições de integridade são:

- Um funcionário não pode pertencer a mais do que um departamento.
- Um estudante só pode inscrever-se em disciplinas do seu curso.
- Um aluno apenas pode estar inscrito num dado curso no momento t, se esse curso, e o próprio aluno, existirem!

Nos exemplos que considerar tente incluir casos "problemáticos" do tipo: um ex-aluno de licenciatura (já licenciado) inscreve-se num outro curso (de mestrado, por exemplo); um funcionário que trabalhou alguns anos num departamento trabalha agora em outro; ...

QUESTÃO 3 (cães). Construa uma BC em que se registem diversos dados sobre *cães* — sua identificação (raça, ano de nascença, sexo, ...), alguns atributos físicos (cor, tamanho do pêlo, tipo de pelo, peso médio, tamanho, quantidade de alimentação diária, ...), atitudes comportamentais (agressividade, nervosismo, comportamento com crianças, ...). O seu programa lógico deverá ser capaz de encontrar os cães (raças) de acordo com os gostos pessoais do utilizador (pode querer um cão de guarda, de companhia, guia de cegos, para criação, ...) e compatíveis com a sua habitação (quinta, apartamento, ...).

QUESTÃO 4 (empresa). Construa uma BC em que se registe a actividade de uma determinada *empresa*. A dinâmica da empresa será caracterizada pelos seus *clientes* (número de contribuinte, nome, morada, telefone, ...), pelas *encomendas* (número de encomenda, data, estado, quantidade, ...), e pelos produtos (ou serviços) que fornece (código do produto, designação, preço, ...). Pretende-se que o seu programa em lógica seja capaz de responder a questões do tipo:

- A que cliente pertence a encomenda `X`?
- Quais as encomendas do cliente `Y`?
- Quais os produtos referenciados na encomenda `Z`?
- Quais as encomendas satisfeitas no mês `A`?
- Que encomendas ainda estão por satisfazer?
- ...

4. Enunciados de Anos anteriores

Questão 1 (casamentos). Construa uma BC em que se registem diversos dados sobre *rapazes* e *raparigas* —sua identificação (nome, idade, BI, contacto, ...), alguns atributos físicos (cor dos olhos, cor da pele/cabelos, altura, peso, ...) e alguns gostos relativos a actividades culturais e desportivas.
O seu programa lógico deverá ser capaz de encontrar os *indivíduos compatíveis* com um novo elemento. Por *compatíveis* entende-se, no mínimo, de sexos opostos e idades aproximadas que partilhem determinados gostos em comum, deixando-se ao critério de cada grupo a definição precisa desse conceito de concordância.

Questão 2 (vinhos). Construa uma BC em que se registem diversos dados sobre *vinhos* —sua identificação (nome, tipo, ano de colheita, zona, ...), alguns atributos físicos (cor, cheiro, acidez, uva principal, grau, ...), temperatura a ser servido e alguns pratos com os quais o vinho se adequa.
O seu programa lógico deverá ser capaz de encontrar os vinhos de acordo com os gostos pessoais do utilizador e compatíveis com uma determinada ocasião.

Questão 3 (mediateca). Construa uma BC em que se registem diversos dados sobre os *vídeos, CD's, discos, cassetes, livros* que alguém possui na sua *mediateca*. Além do tipo de obra, título e autores, a BC deve conter algumas características que permitam qualificar a obra —p. ex., estilo musical (folclore, clássico, ligeiro, orquestra, ...), tipo de filme ou de livro (policial, judicial, romance, descrição, ...), apreciação (calmo, agitado, suspense, cómico, irritante, ...).
O seu programa lógico deverá ser capaz de encontrar as *obras do tipo pedido adequadas* para uma dada situação —p. ex., *jantar a dois*, *noite de festa*, *viagem*, ... O critério de *adequação* e as situações previstas serão definidos por cada grupo.

Questão 4 (turismo). Construa uma BC em que se registem diversos dados sobre recursos portugueses para *turismo de habitação* e *turismo rural* —tipo do alojamento, nome, contacto, preço, localização, classificação (número de estrelas), etc. Além disso, a BC deve dispor também de algum informação geográfica de modo a poder saber a província em que se situa determinado recurso, a distância aproximada a Aveiro e o tipo de clima em cada estação do ano.
O seu programa lógico deverá ser capaz de encontrar os *alojamentos do tipo pedido adequadas* para uma dada situação —p. ex., *fazer praia*, *praticar montanhismo*, *repousar*, ... O critério de *adequação* e as situações previstas serão definidos por cada grupo.

Questão 5 (ensino de geometria). Construa uma BC que contenha informações sobre *figuras planas e sólidos geométricos*. Para além do nome, deverá saber-se quantos lados tem, o que o caracteriza, como se calcula a área, o perímetro, ou o volume, etc.
O seu programa lógico deverá servir para apoiar alunos do ensino secundário que estão a fazer a sua iniciação à geometria, prestando-lhes diversos esclarecimentos quanto às várias formas geométricas, incluindo a capacidade de classificar uma forma dados alguns elementos descritivos. O tipo de apoio que o programa poderá prestar é deixado ao critério de cada grupo.

Questão 6 (associação académica). Construa uma BC em que se registem diversos dados sobre os *alunos da UA* e sócios da respectiva associação académica. Além do número de aluno, número de sócio, nome, elementos de contacto, curso ou actividade profissional, deverá conter elementos que permitam saber os seus passatempos preferidos e a sua envolvência na vida académica (núcleos recreativos a que pertence, ...) e as quotas (valor e última paga).

O seu programa lógico deverá ser capaz de encontrar: as pessoas pertencentes a um determinado núcleo; os candidatos interessados em determinada actividade; os sócios com quotas em atraso; alunos de determinado curso que pratiquem certa actividade; os cinco cursos com mais sócios num determinado núcleo.

22 Segundos Trabalhos Práticos

1. Ano lectivo de 1998/1999 (24/Março/1999)

O trabalho deve ser realizado em grupo, e deve ser entregue a funcionar e acompanhado de um relatório, **impreterivelmente**, até ao dia **26 de Maio**.
*** Lembre-se que a nota prática global entrará na nota final com um peso de 50%. ***

2. Objectivos e Organização

Este trabalho prático tem como principais **objectivos**:

- Sedimentar os conhecimentos envolvidos no paradigma de programação lógico, expandindo-os à análise/especificação de problemas mais complexos.
- Desenvolver as capacidades de programação em Prolog.

Para atingir esses objectivos, esta folha contém vinte enunciados de problemas. Deverá resolver **pelo menos 5 problemas**.
Quanto ao **relatório** a elaborar, para entregar junto com os programas, deve ser claro e sucinto e, além dos respectivos enunciados e listagem dos programas (apresentados em apêndice), deverá conter uma explicação dos predicados incluídos na BC, bem como exemplos de execução em que se mostrem os resultados produzidos para várias questões.
Uma síntese do trabalho (caracterização do problema, estratégia de resolução adoptada e resultados atingidos) deverá ser apresentada **oralmente a toda a turma, nas aulas teóricas dos dias 2 de Junho e 4 de Junho.**

3. Enunciados

3.1. Triângulo. Sendo dada uma lista `L` de caracteres (constantes de comprimento 1), pretende-se desenhar no ecrã um conjunto de triângulos que se situam uns dentro dos outros e que são desenhados com os caracteres em `L` (a ordem interessa ...). Para isso deve implementar um predicado `triangulo/1` que recebe como input a lista `L`. Por exemplo,

```
?- triangulo([a,b,c,d]).
```

terá como resultado:

```
          a
         aba
        abcba
       abcdcba
      abcccccba
     abbbbbbbbba
    aaaaaaaaaaaaa
```

3.2. Tabuleiro. Considere um tabuleiro `4x4` com um operador aritmético binário e um operando inteiro em cada casa (quadrado). Começando com o valor 0 como valor corrente, escolha um caminho que visite cada casa do tabuleiro exactamente uma vez. Cada uma das vezes que visita uma casa, efectua a operação nela indicada considerando o valor corrente como o primeiro operando e o número no quadrado como o segundo operando. O resultado da operação passa então a constituir o valor corrente. No final do caminho, o valor corrente (valor final do caminho) irá depender do caminho escolhido. Pretende-se que implemente um predicado `tabuleiro/3` que encontra o valor final máximo entre todos os caminhos e o número de caminhos que possuem este valor final máximo. Segue-se um exemplo de tabuleiro e da estrutura de dados que o representa:

```
[
   f(*,3),  f(-,1),  f(*,2),  f(+,4),
   f(*,3),  f(-,0),  f(*,2),  f(+,4),
   f(*,3),  f(-,9),  f(*,2),  f(+,4),
   f(*,3),  f(-,8),  f(*,2),  f(+,4)
]
```

* 3	- 1	* 2	+ 4
* 3	- 0	* 2	+ 4
* 3	- 9	* 2	+ 4
* 3	- 8	* 2	+ 4

O predicado `tabuleiro/3` que irá desenvolver, recebe no primeiro argumento o tabuleiro, produzindo no segundo argumento o valor final máximo e no terceiro o número de caminhos que têm como valor o valor final máximo.

3.3. Serpente. A serpente belga é um animal cujo corpo tem um padrão repetido. No entanto esse padrão não é necessariamente repetido um número inteiro de vezes. O padrão consiste numa sequência de anéis. Cada anel tem um identificador que é uma constante de comprimento 1.

A serpente belga é um animal que gosta de repousar enrolada de uma maneira muito particular: repousa sempre na forma de um rectângulo, a cabeça no canto superior esquerdo e preenchendo o rectângulo linha por linha —ver exemplo abaixo.

Pretende-se que implemente um predicado `serpente/3`, que mostre no ecrã uma serpente belga em repouso. Esse predicado será chamado com os seguintes três argumentos:

1. uma lista de constantes representando o padrão;
2. uma lista cujo comprimento é igual ao número de anéis numa linha (diferente de `[]`);
3. uma lista cujo comprimento é igual ao número de anéis numa coluna (diferente de `[]`).

O seu predicado `serpente/3` deverá produzir o resultado no ecrã. Segue-se um exemplo em que a serpente esticada tem o aspecto `a b c d a b c d a b c d a b c`:

```
?- serpente([a,b,c,d],[_,_,_,_,_],[_,_,_]).

a b c d a
b a d c b
c d a b c
```

Antes de começar a implementar o seu programa Prolog, deverá saber que as serpentes belgas têm uma grande aversão por computações aritméticas. Assim sendo, o seu programa não deverá usar nenhum dos operadores aritméticos. Em particular, o seu programa será rejeitado se contiver qualquer um dos símbolos

```
is  <  >  >=  =<  +  -  *  name  arg  functor  =..
```

3.4. Tardes Vazias. Na pág. 13 do livro de Y.I. Perelman, *Matemática Recreativa*, da colecção *a Ciência ao alcance de todos*, Editora MIR, pode ser encontrado o seguinte problema:

> Na nossa escola —começou um estudante —funcionam cinco grupos: de desporto, de literatura, de fotografia, de xadrez e de canto. O de desporto funciona dia sim dia não; o de literatura, uma vez em cada três dias; o de fotografia, uma vez em cada quatro; o de xadrez, uma vez em cada cinco; o de canto, uma vez em cada seis. No primeiro dia de Janeiro, reuniram-se na escola todos os grupos e continuaram depois a reunir-se nos dias designados, sem falhar um

só. Trata-se de saber quantas tardes mais, no primeiro trimestre (90 dias), se reuniram na escola os cinco grupos simultaneamente.

Não vos vamos pedir que resolvam este problema, pois ele tem solução trivial! Basta saber determinar o mínimo múltiplo comum (operação esta já disponibilizada em muitos interpretadores de Prolog) entre 2, 3, 4, 5 e 6. É fácil verificar que este número é 60 e por isso no 61^o dia todos os grupos se reúnem novamente: o de desporto depois de 30 intervalos de dois dias; o de literatura depois de 20 intervalos de 3 dias; o de fotografia depois de 15 intervalos de 4 dias; o de xadrez depois de 12 intervalos de 5 dias; e o de canto depois de 10 intervalos de 6 dias. Passados outros 60 dias chegará uma nova tarde semelhante, durante o segundo trimestre.

Aquilo que vos vamos pedir é algo talvez mais trabalhoso :-) Pretendemos encontrar resposta à seguinte pergunta: *Em que tardes não se reunirá nenhum dos cinco grupos?* Já que vamos implementar um programa em Prolog, vamos fazer as coisas "como devem ser feitas": considerem uma situação mais geral, em que possa existir um número arbitrário de grupos com períodos de funcionamento especificados. Para isso o vosso programa deve pressupor a existência de um predicado `grupos/1` que contém como argumento uma lista de pares `actividade/periodo`. Para a situação acima descrita, a vossa Base de Conhecimento conteria o seguinte facto:

```
grupos([desporto/2,literatura/3,fotografia/4,xadrez/5,canto/6]).
```

O vosso programa deverá ser invocado através do predicado

```
tardesVazias(NumDias,Lista).
```

Para a situação descrita:

```
?- tardesVazias(30,Dias).
   Dias = [2,8,12,14,18,20,24,30]
```

3.5. Formas. Considere uma matriz `NxM` (`N` e `M` diferentes de 0) preenchida com as cores preto e branco. A cor preta representa a cor de fundo. Um conjunto de entradas brancas, ligadas, constitui uma *forma*. O objectivo consiste em determinar o número de formas na matriz. Duas entradas brancas na matriz pertencem a uma mesma forma, se existir um caminho de uma à outra através de entradas brancas. Um caminho consiste em entradas sucessivas com coordenadas `(i,j)` e `(k,l)`, tais que `|i-k| =< 1` e `|j-l| =< 1`.

O seu programa não poderá usar nenhum dos seguintes símbolos:

```
arg  =..  functor  name
```

A matriz será especificada como uma lista de listas contendo `preto` ou `branco` —isto deverá ser óbvio a partir dos exemplos— que constituirá o primeiro argumento do predicado `formas/2` que terá de escrever. O segundo argumento deverá ser unificado com o número de formas no primeiro argumento. Eis alguns exemplos:

```
?- formas([ [preto, branco,preto],
            [branco,preto, preto],
            [branco,preto,branco] ], N).
   N = 2

?- formas([ [branco,branco,branco],
            [branco,preto, preto],
            [branco,preto, preto],
            [branco,preto, preto] ], N).
   N = 1
```

3.6. Gera Código. Considere um computador com n registos, $r_1, \ldots, r_n$, organizados em anel, de modo a que apenas seja possível:

- mover o valor do registo r_i para r_{i+1}, $1 \leq i < n$, com a instrução `move(i)`;
- mover o valor do registo r_n para r_1, com a instrução `move(n)`;
- trocar o conteúdo de quaisquer dois registos, com a instrução `troca(i,j)`, $i < j$.

Dados os conteúdos iniciais de todos os registos (alguns registos podem conter "wild cards") e os conteúdos finais desejados de todos os registos (mais uma vez são admitidos "wild cards"),

pretende-se que implemente o predicado `geraCodigo/3`, que gera uma sequência de instruções mínima, de `move/1` e `troca/2`, capaz de transformar o estado inicial da máquina no estado final, ou então que falhe se isso for impossível.

O predicado `geraCodigo/3` recebe no primeiro argumento uma lista com o conteúdo inicial dos registos; no segundo argumento o conteúdo final dos registos; e retorna, no terceiro argumento e se possível, a sequência de instruções que efectuam a transição. Exemplo:

```
?- geraCodigo([a,b,c,d],[a,d,a,b],L).
   L = [move(2), move(1), troca(2,3), troca(2,4)]
```

Os "wild cards" são representados com um $*$. No estado inicial, um $*$ tem o significado de "não se sabe o conteúdo"; no estado final, significa "não interessa o conteúdo". Eis alguns exemplos:

```
?- geraCodigo([a,*,c],[c,a,*],L).
   L = [troca(1,2), troca(1,3)]
   ou
   L = [move(1), move(3)]
   ou muitas outras solucoes correctas

?- geraCodigo([a,b,c],[a,a,*],L).
   L = [move(1)]
```

3.7. Fecha Válvulas. O meu canalizador deixou a canalização da minha casa num estado verdadeiramente deplorável. A rede de canos comporta-se como uma colecção de canos aleatoriamente ligados (ainda não consegui perceber a lógica, se é que existe). Felizmente, ele ligou —de uma ou mais maneiras— cada torneira ao depósito central de abastecimento de água e instalou válvulas em alguns canos, de maneira que eu posso evitar o fluxo de água (em qualquer das direcções) através desse cano. Esta manhã, uma torneira estava a pingar e antes de sair de casa, quis fechar um número suficiente de válvulas de modo a prevenir aquela torneira de pingar, mas não muitas, de modo a ter um número máximo de outras torneiras ainda capazes de dar água. Isto pareceu-me elementar: conhecendo a (planta da) minha casa como conheço, pensei numa solução, e fechei as válvulas que calculei a partir do meu plano. A torneira, essa continuou a pingar ... e mais uma vez cheguei atrasado às minhas aulas! Resulta que algumas válvulas não fecham correctamente :-(significando que não posso confiar no meu plano. Ajudem-me, por favor. Para isso escrevam um programa em Prolog que dada uma "planta da casa", como um conjunto de factos descrevendo a rede de canos:

- `entrada/1`: existe exactamente uma;
- `torneira/1`: existe pelo menos uma;
- `aPingar/1`: existe exactamente uma;
- `cano/2`: significa que 2 nós na rede estão ligados por um cano;
- `valvula/2`: um subconjunto de `cano/2`, significando que uma válvula foi instalada nesse cano.

Os argumentos de todos estes factos são constantes, representando um nó da rede, a entrada, ou uma torneira. Também pode usar um predicado `daAgua/2`, que é chamado dando no primeiro argumento uma torneira e no segundo argumento uma lista de válvulas que quer fechadas. A chamada ao predicado `daAgua/2` tem sucesso se, e somente se, o fechar das válvulas não evita o fluxo de água da entrada para a torneira. Digamos que é o análogo de eu andar a correr, escadas acima, escadas abaixo, fechando válvulas (andar de baixo) e verificando se a torneira (andar de cima) ainda está activa. A diferença é que eu fico mais cansado ...

Escreva um predicado `fechaValvulas/1` que é chamado com uma variável livre (por instanciar) e que a unifica com uma lista de válvulas que precisam de ser fechadas de modo a que a torneira que pinga deixe de pingar e de modo que o número de outras torneiras ainda capazes de deitar água seja máximo. Oh, o meu canalizador talvez tenha sido tão desajeitado, que seja impossível parar a torneira de pingar! Nesse caso o predicado `fechaValvulas/1` pode falhar —em tempo finito, claro! Mais uma coisa: eu não tenho a certeza de que todas as torneiras estão ligadas à entrada ... o meu canalizador foi mesmo muito mau!

3.8. Diamante. Desenhe no ecrã um "diamante". Generalize a partir dos exemplos. Deve implementar um predicado `diamante/1`, a ser chamado com um inteiro positivo, que especifica o tamanho do diamante. Eis alguns exemplos:

```
?- diamante(2).

   1
3     2
   4

?- diamante(3).

      1
   4     2
7     5     3
   8     6
      9

?- diamante(8).

                              1
                           9     2
                        17     10     3
                     25     18     11     4
                  33     26     19     12     5
               41     34     27     20     13     6
            49     42     35     28     21     14     7
         57     50     43     36     29     22     15     8
            58     51     44     37     30     23     16
               59     52     45     38     31     24
                  60     53     46     39     32
                     61     54     47     40
                        62     55     48
                           63     56
                              64
```

3.9. Compacta. Considerem-se sequências de letras do alfabeto (i.e., de a até z) como entrada. Um exemplo poderá ser

```
xcaabaabaabccadadcaabaabaabccadady
```

Uma sequência pode ser *compactada* por um algoritmo muito simples, que substitui n (digamos 7) ocorrências consecutivas de um símbolo (digamos p) pelo símbolo seguido de n —no nosso exemplo, p7, e p7 é claramente mais curto que ppppppp. O mesmo algoritmo de compactação pode ser também aplicado a subsequências. Por exemplo, ababab pode ser compactado por (ab)3. Note que (ab)3 tem 5 caracteres.

Deste modo, a sequência dada acima pode ser comprimida, por exemplo, por

```
x(c(a2b)3c2(ad)2)2y
```

No entanto esta não é a compactação mais curta, por causa da ocorrência `(ad)2`: `(ad)2` usa mais um caracter do que `adad`. Resulta claro quando é que os parêntesis são necessários e quando não. Uma vez que nem os dígitos nem os parêntesis ocorrem na string de entrada, não existe qualquer tipo de ambiguidade.

Deverá desenvolver um predicado `compacta/2`, que recebe como entrada uma sequência —representada como uma lista de letras— e que retorna a lista compactada mais curta, que lhe é equivalente. Para o exemplo apresentado, teremos:

```
?- compacta([x,c,a,a,b,a,a,b,a,a,b,c,c,a,d,a,d,c,a,a,b,a,a,b,a,a,b,c,c,a,d,a,d,y],L).
```

```
L = [x,(,c,(,a,2,b,),3,c,2,a,d,a,d,),2,y]
```

embora também seja correcto

```
?- compacta([x,c,a,a,b,a,a,b,a,a,b,c,c,a,d,a,d,c,a,a,b,a,a,b,a,a,b,c,c,a,d,a,d,y],L).
L = [x,(,c,(,a,a,b,),3,c,c,a,d,a,d,),2,y]
```

3.10. Troca. Pretende-se implementar o predicado `troca/2` que é chamado com o primeiro argumento instanciado com uma lista de inteiros positivos e com o segundo argumento livre (por instanciar). O predicado deve unificar o segundo argumento com o valor máximo do primeiro argumento. O valor básico de uma lista de inteiros, é obtido aplicando, alternadamente, as operações de diferença e soma aos seus elementos. Por exemplo

```
?- valor([9,8,2],V). % V = 9-8+2 = 3
   V = 3

?- valor([1,4,89,12],V). % V = 1-4+89-12 = 74
   V = 74
```

Qualquer outro valor da lista é o valor básico de uma lista que se obtém trocando a posição dos números na lista. Esta operação de troca de números é um bocado peculiar. Primeiro de tudo, ela funciona na representação decimal dos inteiros na lista. Em segundo lugar, ela só é possível em números que estejam em posições vizinhas na lista. Eis alguns exemplos de mudanças entre dois elementos vizinhos numa lista:

```
17,34    transforma-se em    43,71

123,45   transforma-se em    54,321
```

Percebido? Mais uma restrição: os números não poderão mover-se mais longe do que as posições vizinhas na lista. Assim,

```
[12,34,56]  em  [43,65,12]
```

não é possível, uma vez que o número 12 foi movido duas posições na lista.

Seguem-se dois exemplos mostrando o comportamento desejado do predicado `troca/2`:

```
?- troca([1,2],V).
   V = 1

?- troca([12,56,34],V).
   V = 78
```

3.11. Espiral. Dados dois inteiros (estritamente) positivos `a` e `b`, mostre no ecrã um rectângulo de largura `b` e altura `a`, contendo os números de 1 a $a * b$ na forma espiral. Para `a=4` e `b=3`, teremos:

```
?- espiral(4,3).

 1    2    3
10   11    4
 9   12    5
 8    7    6
```

As colunas devem estar convenientemente alinhadas à direita.

3.12. Dominó. Considere um conjunto de pedras de dominó. Estas pedras têm 2 números no mesmo lado, um na parte esquerda, outro na parte direita, do género do seguinte exemplo:

2	6		3	3

No jogo usual, cada participante recebe um certo número de pedras e uma pedra é colocada na mesa. Cada jogador, à vez, deve adicionar uma pedra às pedras já colocadas na mesa, ou passar. Ganha quem despachar todas as suas pedras em primeiro lugar. Para adicionar, de acordo com as regras, uma pedra na mesa, devemos encontrar uma pedra na mesa com uma parte livre contendo o número `x` e colocar uma pedra com lado `x` em contacto com o lado `x` livre na mesa. As pedras com dois números diferentes têm, inicialmente, dois lados livres — um para cada número. As pedras com dois números iguais têm 3 lados livres. Quando uma

pedra é colocada com um lado livre em contacto com o lado livre de outra pedra, ambos os lados livres ficam ocupados.

Aqui consideramos uma variante do jogo, chamada *dominó solitário*: é-lhe facultado um conjunto de pedras e o seu objectivo é terminar o jogo (todo) sozinho. Assim, deve começar por colocar uma das suas pedras na mesa (escolhe qual), e continuar a adicionar pedras, até não ter mais. Este processo pode falhar: pode não ser possível livrarmo-nos de todas as pedras.

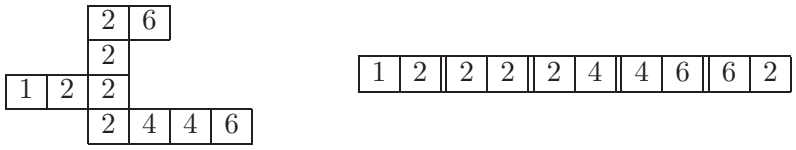
Segue-se um exemplo. Para os factos

```
pedra(2,2).
pedra(4,6).
pedra(1,2).
pedra(2,4).
pedra(6,2).
```

o predicado `domino/1` que irá escrever, retorna, por exemplo, a lista:

```
?- domino(L).
   L = [pedra(1,2),pedra(2,2),pedra(2,4),pedra(4,6),pedra(6,2)]
```

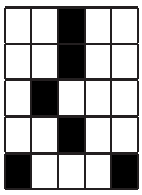
Esta lista pode representar mais do que uma configuração final na mesa (veja-se a figura), mas a ordem na lista dá-nos uma ordem correcta pela qual as pedras podem ser incrementalmente adicionadas a uma configuração que é sempre correcta de acordo com as regras do jogo.



3.13. Palavras Cruzadas. Considere um puzzle de palavras cruzadas vazio. Um exemplo pode ser o especificado pelos factos

```
dimensao(5).  % trata-se sempre de um puzzle quadrado
preto(1,3).   % significa que o quadrado 1,3 e' preto
preto(2,3).
preto(3,2).
preto(4,3).
preto(5,1).
preto(5,5).
```

que representam o puzzle vazio:



Considere depois uma lista de palavras. Por exemplo

```
palavras([do,ore,ma,lis,ur,as,po,so,pirus,oker,al,adam,ik]).
```

Todas as palavras têm pelo menos dois caracteres. O objectivo é preencher o puzzle usando todas as palavras exactamente uma vez. Se isso for impossível, o predicado `palavrasCruzadas/1` que irá desenvolver deve falhar. O seu programa deve dar a solução no argumento de saída do predicado `palavrasCruzadas/1` e na forma de uma lista de palavras na seguinte ordem: primeiro todas as palavras na horizontal, linha por linha, conforme elas ocorrem da esquerda para a direita; depois todas as palavras na vertical, coluna por coluna.

Para o exemplo dado acima, teremos

```
?- palavrasCruzadas(L).
   L = [as,po,do,ik,ore,ma,ur,lis,adam,so,al,pirus,oker]
```

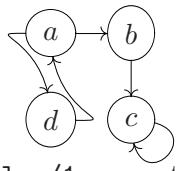
significando que o puzzle é preenchido da seguinte maneira:

A	S	■	P	O
D	O	■	I	K
A	■	O	R	E
M	A	■	U	R
■	L	I	S	■

Se existir mais do que uma solução, o seu programa deve ser capaz de as produzir por backtracking. Cada quadrado não preto pertence a uma palavra de 2 ou mais caracteres.

3.14. Ciclos. Considere um grafo dirigido representado por factos `seta/2`. Por exemplo:

```
seta(a,b).
seta(b,c).
seta(c,c).
seta(a,d).
seta(d,a).
```



Pretende-se que implemente um predicado `ciclos/1`, que retorne uma lista (possivelmente vazia) com todos os ciclos (mínimos) no grafo. Um ciclo é representado por uma lista que contém todos os nós no ciclo na ordem em que as setas os ligam e começa e acaba com o mesmo nó. Um ciclo mínimo não contém qualquer outro ciclo nele mesmo. Cada ciclo mínimo do grafo deve ser dado exactamente uma vez. Para o exemplo acima, temos

```
?- ciclos(C).
   C = [[c,c],[a,d,a]]
```

ou

```
?- ciclos(C).
   C = [[c,c],[d,a,d]]
```

ou

```
?- ciclos(C).
   C = [[a,d,a],[c,c]]
```

ou

```
?- ciclos(C).
   C = [[d,a,d],[c,c]]
```

O seu programa deve dar uma das listas `C` acima: a ordem dos ciclos na lista é irrelevante assim como o nó de começo de um ciclo.

3.15. Caminho. Num tabuleiro quadrado de tamanho `N`, por exemplo 4, dado pelo facto

```
tamanho(4).
```

existe um ponto de partida (que é sempre o quadrado 1, 1) e um ponto de chegada, dado, por exemplo, pelo facto

```
goto(1,4,f). % f = fim (do caminho)
```

Existe um caminho correcto entre o ponto de partida e o ponto de chegada e ele está marcado: cada quadrado do caminho contém a informação necessária para determinar o próximo quadrado do caminho, na forma de um facto:

```
goto(1,1,c). % significa que ao quadrado 1,1 sucede o quadrado 1,2 (c = cima)
goto(1,2,c).
goto(1,3,d). % significa que ao quadrado 1,3 sucede o quadrado 2,3 (d = direita)
goto(2,3,d).
goto(3,3,b). % significa que ao quadrado 3,3 sucede o quadrado 3,2 (b = baixo)
goto(3,2,b).
goto(3,1,d).
```

Um `e` teria significado, obviamente, 'esquerda'. A informação acima representa o tabuleiro:

d	d	b	
c		b	
c		d	f

Claro que não existe qualquer dificuldade em encontrar o caminho correcto de 1, 1 para 1, 4 em tal tabuleiro. Contudo, um vírus maligno no meu computador (para o qual, infelizmente, ainda não existe antídoto) remove informação do tabuleiro sempre que eu crio um. É por isso que vos estou a pedir ajuda. O vírus apaga alguns factos `goto/3`, de maneira que o caminho não fica completamente representado. Depois de algumas tentativas, cheguei à conclusão que, ainda assim, o vírus não é exageradamente maligno. Com efeito ele nunca remove a informação de dois quadrados vizinhos no caminho; nunca remove o ponto de chegada; e, no máximo, dois vizinhos de um quadrado removido pertencem ao caminho correcto (quadrados que se tocam na diagonal não são considerados vizinhos). Desde modo, ainda é fácil encontrar o caminho. Todavia, hoje fui infectado por um segundo vírus, mais maligno, que introduz informação para cada quadrado *não* no caminho e de tal modo que se seguirmos essa informação, das duas uma: ou andamos às voltas em círculos; ou saímos para fora do tabuleiro (em particular, não encontramos quadrados 'vazios'). De seguida mostro o tabuleiro acima depois de infectado pelos dois vírus:

e	e	c	d
d		b	b
	b		c
c	b	d	f

O que vos peço é que escrevam um predicado `caminho/1` que nos dê o caminho correcto, como uma lista começando por `(1,1)` e acabando no ponto de chegada. A lista deve estar ordenada ao longo do caminho. Assim, para o exemplo acima, devemos obter:

```
?- caminho(C).
   C = [(1,1),(1,2),(1,3),(2,3),(3,3),(3,2),(3,1),(4,1)]
```

3.16. Árvore de Natal. Uma árvore de Natal de tamanho 5, aparecerá no ecrã como:

```
?- arvoreNatal(5).

      *
     * *
    * * *
   * * * *
  * * * * *
```

Note que existe um espaço entre cada duas estrelas numa linha horizontal. Entre o topo de uma árvore de Natal de tamanho `N` e o lado esquerdo do ecrã, não deverá existir mais do que `N+2` espaços. Escreva o predicado `arvoreNatal/1` que é chamado com o seu argumento instanciado com um inteiro não negativo e que desenha uma árvore de Natal desse tamanho no ecrã.

3.17. Repete. Escreva um predicado `repete/3` que será chamado com inteiros positivos (não nulos) nos primeiros dois argumentos e com o terceiro argumento livre (por instanciar). Tal chamada ao predicado deve ter sucesso exactamente uma vez, retornando no terceiro argumento uma lista de inteiros (de 0 a 9) que representa a parte decimal repetida do número que se obtém dividindo o primeiro argumento pelo segundo. Eis alguns exemplos:

```
?- repete(3,4,R).  % 3/4 = 0.250000...
   R = [0]

?- repete(4,3,R).  % 4/3 = 1.333...
   R = [3]

?- repete(1,7,R).  % 1/7 = 0.142857142857...
   R = [1,4,2,8,5,7]
```

No último exemplo,

```
R = [2,8,5,7,1,4]
```

também é uma resposta correcta —assim como qualquer sua rotação.

3.18. Potências. Escreva um predicado `potencias/3` que deve ser chamado fornecendo no primeiro argumento uma lista de inteiros positivos maiores que 1, no segundo argumento um inteiro positivo `N` e o terceiro argumento livre (por instanciar). O predicado deve terminar com sucesso e unificar o terceiro argumento com a lista que contém (por ordem crescente) os `N` inteiros mais pequenos que são uma potência positiva (não nula) de um dos elementos do primeiro argumento. Alguns exemplos:

```
?- potencias([3,5,4],17,L).
   L = [3,4,5,9,16,25,27,64,81,125,243,256,625,729,1024,2187,3125]

?- potencias([2,3,4,5,6,7,8,9,10],50,L).
   L = [2,3,4,5,6,7,8,9,10,16,25,27,32,36,49,64,81,100,125,128,216,
        243,256,343,512,625,729,1000,1024,1296,2048,2187,2401,3125,
        4096,6561,7776,8192,10000,15625,16384,16807,19683,32768,
        46656,59049,65536,78125,100000,117649]

?- potencias([2,9999999,9999999],20,L).
   L = [2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
        65536,131072,262144,524288,1048576]

?- potencias([4,2],6,L).
   L = [2,4,8,16,32,64]
```

3.19. Saída. Considere um labirinto com as seguintes propriedades: cada posição no labirinto é caracterizada por duas coordenadas `(X,Y)`, ambas inteiras. O labirinto tem uma entrada, por convenção na posição `(1,1)`, e uma saída — já adivinhou: terá de encontrar as coordenadas da saída. O labirinto tem um estado interno chamado `posicao_corrente`, que é colocado, no início, na posição de entrada e que representa a posição onde estamos no labirinto. No entanto, não temos acesso directo a este estado interno! Podemos interrogar o labirinto por meio de dois predicados: `move/2` e `fim/0`. O predicado `fim/0` retorna "sim" se, e somente se, `posicao_corrente` coincide com a saída. O predicado `move/2` é um pouco mais complicado. Chamamo-lo com os argumentos instanciados, representando uma posição. Por exemplo

```
?- move(3,4).
```

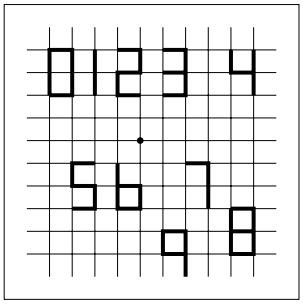
Se o labirinto for tal, que possamos ir, num único passo, da `posicao_corrente` para a posição `(3,4)`, o predicado retorna "sim" e actualiza o estado interno `posicao_corrente` para `(3,4)`. Se não for possível ir num passo da `posicao_corrente` para `(3,4)`, o predicado falha (retorna "não") e o estado interno `posicao_corrente` é reinicializado para a posição de entrada, isto é, somos colocados, de novo, na posição inicial, quando tentamos fazer um passo não admissível. O mesmo acontece se chamarmos `move/2` com argumentos por instanciar ou com argumento(s) "incorrecto(s)". Para limitar as possibilidades, um passo directo é apenas possível para uma posição adjacente: as coordenadas `X` ou `Y` diferem de uma unidade.

Deve escrever um predicado `saida/2` que será chamado com os argumentos livres (por instanciar) e cujos argumentos serão, em caso de sucesso, instanciados com as coordenadas da saída do labirinto. O tamanho do labirinto não é dado (o predicado `move/2` falha quando dada uma posição não existente).

Existem duas subtilezas neste problema:

1. Não temos ideia quão grande o labirinto é, nem a sua forma global.
2. Se um movimento falha, não podemos simplesmente retroceder essa tentativa, porque somos atirados para a entrada do labirinto!

3.20. Números. Considere uma grelha infinita que possui um número finito de elementos iluminados. Por exemplo



Estes elementos iluminados são representados por factos `luz/4`, que têm como argumentos as coordenadas do início e do fim. Tal como um mostrador digital, estas partes iluminadas pode ser vistas como formando números de 0 a 9.

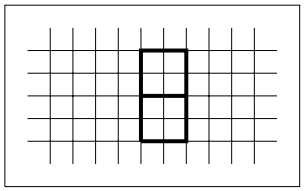
Pretende-se que implemente um predicado `numeros/1`, que retorna uma lista (sem elementos duplicados, mas onde a ordem não é importante) com todos os números que são reconhecidos a partir do conjunto de factos dado. No caso da figura acima, a grelha contém todos os números de 0 a 9, pelo que devemos obter (a menos da ordem dos elementos na lista):

```
?- numeros(N).
   N = [0,1,2,3,4,5,6,7,8,9]
```

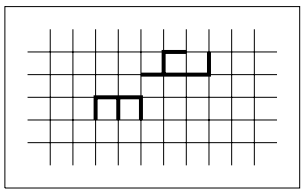
O ponto na grelha indica o ponto de coordenadas (0, 0). Desde modo, o número 3 pode ser representado pelos factos:

```
luz(1,2,2,2).
luz(2,2,2,3).
luz(2,3,2,4).
luz(2,3,1,3).
luz(2,4,1,4).
```

Admitimos, por simplicidade, que todos os números estão representados na escala indicada na figura acima. Por conseguinte, o predicado `numeros/1` não deverá reconhecer o número 8 na grelha



Por outro lado, os números podem ter sofrido rotações e devem ser igualmente reconhecidos como sub parte de uma configuração de elementos iluminados. Por exemplo para a figura



deveremos ter

```
?- numeros(L).
   L = [1,3,4,7]
```

A orientação dos números não precisa ser a mesma para todos eles e uma parte acesa pode ser usada como parte de mais do que um número. Por exemplo, um 4 e um 1 estão escondidos num 9 —que também poderá ser um 6 rodado!

4. Enunciados de Anos anteriores

4.1. Projecto Casino. O Casino Dakalecas, ponto de referência no desenvolvimento turístico da região de Aveiro, está presentemente a diversificar o seu leque de oferta em jogos. Nesse sentido, contratou-os para desenvolverem dois programas: um para jogar o Jogo das Cores e outro que implemente o Jogo da Vida.

4.1.1. *Jogo das Cores.* Este jogo desenvolve-se num tabuleiro formado por buracos organizados na forma de um quadrado ou de um rectângulo e utiliza um conjunto de peças de diversas cores. O objectivo é colocar as peças nos buracos de forma a que nenhum par de peças da mesma cor fique colocado de forma contígua, tanto na horizontal, como na vertical, como na diagonal.
Pretende a gerência do casino que o programa deixe o cliente definir a dimensão do tabuleiro, o número de cores e o número de peças de cada cor com que se jogará. O programa deve permitir dois modos de utilização:

- **a):** um modo automático, em que competirá ao programa preencher o tabuleiro de acordo com as regras (ou então indicar que tal não é possível);
- **b):** um modo interactivo em que o cliente preenche o tabuleiro ficando a cargo do programa a sua validação.

4.1.2. *Autómatos celulares e vida artificial.* A teoria matemática dos autómatos celulares foi criada por *John von Neumann*, nos anos cinquenta. Um autómato celular é um conjunto de células —quadrados dispostos numa grelha bidimensional, semelhante a um tabuleiro de xadrez, onde cada casa é uma célula.
Cada célula pode estar num certo número de estados (tal como cada casa do tabuleiro pode estar num estado branco ou num estado preto). O autómato inteiro pode evoluir no tempo de acordo com um conjunto de regras predefinidas; essas regras de evolução são baseadas no estado em que se encontram as células vizinhas da célula em causa (recorde-se que em volta de cada célula existem oito outras células). Por exemplo, uma regra podia então dizer que sempre que haja quatro ou mais células brancas em torno de uma outra, esta célula muda de cor no passo temporal seguinte.
Há inúmeras possibilidades de regras diferentes e cada conjunto de regras conduz a uma evolução diferente do autómato. A ideia de um autómato celular é simples ... decepcionalmente simples. Há, no entanto, quem veja os autómatos celulares para além de meras curiosidades matemáticas e chegue mesmo a afirmar que o Universo é um computador e em particular um autómato celular. Quem já brincou com autómatos celulares em ecrãs de computadores, ficou concerteza impressionado com o facto de, com certos tipos de regras, os autómatos poderem gerar ondas e outros tipos de movimentos observados na natureza.
Uma boa maneira de visualizar um autómato celular consiste em imaginar o ecrã de um monitor como uma matriz —a cada elemento da matriz, chamamos pixel. Neste sistema simples um pixel pode estar aceso, "1", ou apagado, "0". Estes pixels serão as células componentes do autómato a representar.
Para desenvolver o jogo pretendido, os requisitos da gerência do casino são:

- a implementação do Jogo da Vida a duas dimensões;
- o tamanho da "colónia de células" possa ser definido pelo utilizador;
- o cliente deverá poder definir a configuração inicial.

A definição das regras de evolução concretas a implementar e do número de estados possível para cada célula é deixada ao critério do grupo de trabalho.
Observação: O Casino Dakalecas considera que estes dois jogam partilham algo em comum! De facto ambos são baseados num tabuleiro rectangular, configurável. No jogo da vida, o facto de uma célula estar viva ou morta pode ser representado por uma cor. Se pensarmos que uma célula pode conter vários estados de energia (que vai da energia zero, que equivale a estar morta, a um nível de energia máximo) e representarmos esse nível de energia por uma cor, podemos olhar para o Jogo da Vida como um Jogo das Cores com regras específicas. Por este motivo a gerência do casino pretende ter **dois grupos de trabalho em simultâneo** que possam aproveitar um interface comum, discutir várias soluções e integrar os dois programas num só.

4.2. Projecto de Apoio à Feitura de Horários. No nosso departamento, os horários são feitos "à mão", havendo por vezes problemas com a gestão dos vários recursos: disciplinas, salas e docentes. Pretende-se a implementação de dois programas: um de gestão de salas e disciplinas; e um outro de gestão de docentes e disciplinas.
Tal como no projecto anterior, pretende-se que haja **dois grupos de trabalho em simultâneo** que possam aproveitar uma especificação comum da base de conhecimento —*as disciplinas sob a responsabilidade do nosso departamento*— e que no final sejam capazes de

discutir várias soluções e integrar os dois programas num só.
Este programa integrado de apoio à feitura de horários, deve possuir um menu de questões pertinentes, bem como as operações de manutenção necessárias.
Descrevem-se agora detalhadamente cada um desses problemas.

4.2.1. *Gestão de Salas e Disciplinas.* Pretende-se a construção de um programa que permita manter uma **base de conhecimento com as salas** existentes e as suas características (número de alunos que comporta, se está destinada exclusivamente a aulas com determinadas características, etc.), bem como o horário de ocupação de cada sala, em cada tempo lectivo (8h00 as 19h00) ao longo da semana, permitindo a realização de consultas pertinentes, como sejam:

- A sala X está livre no dia Y, das Z às T horas?
- Em que salas pode decorrer uma aula teórica da cadeira X no dia Y, das Z às T horas?
- Em que dias pode decorrer uma aula prática na sala X, das Z às T horas?

Para responder a questões deste tipo, será necessário manter em simultâneo uma **base de conhecimento das disciplinas**, que contenha as características próprias de cada cadeira (curso a que se destina, ano de licenciatura, escolaridade da disciplina, etc) assim como o número total de alunos inscritos. Para o caso de existirem vários turnos da mesma disciplina, será também necessário indicar a distribuição dos alunos por esses turnos. Sugere-se a manutenção de três triplos para cada disciplina:

- escolaridade: (T, TP, P) - $(3, 2, 0)$ para o caso da cadeira de IPL.
- número de turnos: (T, TP, P) - $(1, 2, 0)$ para o caso da cadeira de IPL.
- distribuição dos turnos: (T, TP, P) - $(1, 1, 0)$ para o caso da cadeira de IPL (se as aulas teóricas fossem dadas em altura diferente seria $(2, 1, 0)$).

4.2.2. *Gestão de Docentes e Disciplinas.* Este programa deve permitir manter uma **base de conhecimento com as disciplinas**, tal como foi sugerido na questão anterior, e uma **base de conhecimento de docentes**. Esta última deve conter atributos tais como: estatuto do docente (assistente, doutorado ou agregado), a área científica em que se integra (Análise, Álgebra, Computação, Investigação Operacional, etc), uma lista com as cadeiras que tem dado em anos anteriores, uma lista com as cadeiras que prefere dar no próximo ano, horário do semestre corrente, horário de atendimento, etc. Estes dados devem permitir responder a questões do género:

- Que cadeiras são adequadas ao perfil do docente X?
- Que docentes poderão dar as aulas práticas da cadeira X? (Os assistentes apenas podem dar aulas práticas, os doutorados podem dar aulas práticas e teóricas e os agregados apenas teóricas).

4.3. Espectáculo de Folclore. Em trabalho desenvolvido na UM (no contexto dum projecto de 5 º ano), foi desenvolvido um **compilador** que permite, ao Responsável por um Grupo de Danças Folclóricas, **descrever os recursos** de que dispõe para efectivar um determinado espectáculo, bem como o **programa** (*sequência de danças*) que pretende apresentar. Esse compilador tem acesso a uma base de dados onde estão definidos todos os recursos habituais do grupo: danças; dançadores; cantadores; tocadores.
Ao descrever a informação para planear um espectáculo, basta pois declarar: novos recursos que se queiram acrescentar à base de dados; recursos que se queira ver definitivamente removidos; e os membros habituais do grupo que não irão estar presentes no espectáculo a programar.
O compilador, após validar a informação descrita gera um programa em Prolog, que é constituído por um conjunto de factos que descrevem os 4 recursos básicos do Grupo Folclórico (acima enumerados) —correspondentes ao predicados danca, dancador, cantador, tocador— e um outro conjunto de factos que descrevem as danças a alocar, e os recursos humanos que não pode ser incluídos. Além disso, o programa inclui a invocação a um predicado, aloca/0, que implementa um algoritmo standard de alocação dos cantadores, tocadores e pares de dançadores a cada dança prevista para o programa do espectáculo.

Este trabalho consiste, precisamente, na análise da base de conhecimento gerada pelo compilador e no desenvolvimento do dito algoritmo de formação dos pares que irão dançar cada

dança (bem como dos tocadores e cantadores que serão usados).
A questão principal em causa, é a identificação de todas as restrições que devem ser respeitadas na formação dos ditos pares de dançadores, de modo a satisfazer as vontades de cada pessoa, os ideias estéticos do responsável pelo grupo e a proporcionar a todos os elementos uma intervenção semelhante no espectáculo (não se pretende que um dançador entre em 5 danças e um outro só dance 1).

4.4. Interfaces em língua natural. O intuito desta proposta de trabalho é melhorar os programas desenvolvidos pelos grupos na 1ª série de problemas (proposta nesta disciplina), dotando-os com um processador de linguagem que permita aos Utilizadores desses programas dialogarem numa linguagem mais livre, próxima da nossa língua natural (português).
Na verdade, na versão que foi construída na fase anterior, para que alguém possa usar os programas e coloque questões à base de conhecimento, é necessário que conheça a sintaxe exacta de cada predicado (nome preciso do predicado, número de argumentos e ordem pela qual eles devem aparecer). Para optimizar a interface desses trabalhos será desejável que o Utilizador possa construir questões numa sintaxe mais natural, com maior grau de liberdade.
Os grupos que escolherem este trabalho, terão de definir a linguagem que querem fornecer aos Utilizadores do seu programa, especificá-la usando uma gramática (segundo o formalismo DCG) e construir um tradutor que reconheça as novas frases e gere os predicados convenientes para questionar a base de conhecimento.

4.5. Diagramas de Pert. Desenvolva um programa em Prolog para ajudar a fazer o planeamento e controlo de actividades/projectos usando uma Rede de Actividades (Diagrama de Pert).
O seu programa deverá aceitar o definição do grafo orientado que modela a dita rede e deverá calcular as actividades críticas do projecto, bem como responder a outras questões relacionadas com as actividades que se podem executar em paralelo e as que têm de ser executadas em sequência.

4.6. Árvore de Decisão. O objectivo geral deste trabalho é desenvolver um programa em Prolog capaz de simular em computador um *processo de decisão com capacidade de aprendizagem*. Esse tipo de programa pode ser útil para *classificar* objectos (físicos, ou não) de acordo com as suas características observáveis, ou para decidir da adequabilidade de uma coisa ou pessoa para determinado fim, ou para nos ajudar a tomar uma opção. Em qualquer um dos casos, o programa deve ter a capacidade de incorporar mais conhecimento, *tornando-se mais esperto*, se cada vez que for usado não apresentar a solução que o utilizador acha correcta.
Assim e para cumprir tal objectivo, o grupo de trabalho deverá escolher uma situação concreta —por exemplo, *classificação de animais*, *classificação de figuras geométricas*, *classificação de doenças*, *escolha da toilete a usar em determinadas situações*, etc.— à qual quer aplicar esta técnica de programação baseada numa árvore binária de decisão.
Numa **1ª fase**, o grupo deve implementar a sua árvore de decisão e a aprendizagem, sem qualquer preocupação com a interface, ou seja, o programa será usado invocando directamente os predicados desenvolvidos, sem facilidades especiais para formatação / apresentação dos resultados calculados.
Numa **2ª fase**, pretende-se melhorar o programa desenvolvido dotando-o com um processador de linguagem que permita aos Utilizadores dialogarem numa linguagem mais livre, próxima da nossa língua natural (português). Na verdade, na versão que foi construída na fase anterior, para que alguém possa usar os programas e coloque questões à base de conhecimento, é necessário que conheça a sintaxe exacta de cada predicado (nome preciso do predicado, número de argumentos e ordem pela qual eles devem aparecer). Para optimizar a interface desses trabalhos será desejável que o Utilizador possa construir questões numa sintaxe mais natural, com maior grau de liberdade.

Os grupos que escolherem este trabalho, terão de definir a linguagem que querem fornecer aos Utilizadores do seu programa, especificá-la usando uma gramática (segundo o formalismo DCG) e construir um tradutor que reconheça as novas frases e gere os predicados convenientes para questionar a base de conhecimento.

4.7. O dicionário marciano.

> *"It is later than you think" could not be expressed in Martian - nor could "Haste makes waste", though for a different reason: the first notion was inconceivable while the latter was an unexpressed Martian basic, as unnecessary as telling a fish to bathe. But "As it was in the Beginning, is now and ever shall be" was so Martian in mood that it could be translated more easily than "two plus two makes four" - which was not a truism on Mars.*
>
> Robert A. Heinlein, Stranger in a Strange Land

[1]Ninguém sabe como ou por que desapareceu a civilização marciana, mas uma caverna no sopé do monte Olimpo guarda uma triste recordação desta cultura antiga e fabulosa. No interior da caverna existe uma placa, reminiscente da famosa pedra de Roseta, coberta por um tecido empoeirado de composição desconhecida. Afastando este tecido, surge o que parece ser uma espécie de dicionário: uma lista de palavras na metade esquerda da placa está associada a uma lista à direita. Duas frases aparecem gravadas no fundo da placa. Essas frases estão relacionadas de uma forma peculiar. A primeira pode ser transformada na segunda por substituição repetida de palavras contidas no dicionário: cada palavra da frase no lado esquerdo do dicionário pode ser substituída pela correspondente palavra no lado direito. Os sábios do Marte antigo defendiam que qualquer informação que valesse a pena aprender, podia ser obtida começando com uma frase básica e substituindo palavras de acordo com o dicionário da sabedoria, cujo único fragmento se encontra na caverna. Em geral, um computador não poderá verificar se uma dada frase pode ser derivada da frase básica. Por outras palavras, não existe qualquer possibilidade de escrever um programa de computador (não importa quão extenso ou rápido) que seja capaz de decidir correctamente, para cada dicionário e duas palavras (ou frases) de entrada, se é possível a tradução da primeira para a segunda palavra. A demonstração decorre da insolubilidade do chamado problema de paragem e do facto de que o problema da paragem pode ser convertido no problema da tradução marciana.

O problema da tradução marciana é um exemplo de uma família de puzzles que requerem a transformação de palavras, frases e até parágrafos inteiros em outras palavras, frases e parágrafos. Entre muitos passatempos matemáticos e simbólicos destaca-se uma transformação chamada "escada de palavras". Numa escada de palavras começamos com duas palavras específicas. A primeira palavra chama-se origem e a segunda destino. Poderemos transformar a palavra origem na palavra destino alterando uma letra de cada vez? Se as duas palavras têm o mesmo número de letras, a tarefa é trivial. Mas poderemos garantir que todas as cadeias de caracteres intermédias são também palavras? Podemos, de facto, partir de "gato" para ilustrar o processo. Num passo, a palavra pode ser alterada para "pato"; noutro passo, para "rato". Com esta alquimia até parece possível que "gato" pode ser facilmente modificado para qualquer nome com quatro letras. Será isto possível?

Algumas ideias para o trabalho - "achas para a fogueira"

- Arranje-se um dicionário e construa-se em Prolog um programa que pesquise o dicionário para encontrar palavras relacionadas. O utilizador especifica uma dada palavra e um predicado será o responsável pela formação da tal "teia de palavras", satisfazendo os requerimentos da escada de palavras.
- Para sabermos se a palavra *destino* pode ser encontrada a partir da palavra *origem*, poderíamos percorrer apenas a "teia de palavras" a partir de *origem* e ver se nesta viagem encontramos *destino*.
- Tentem transformar *bom* em *mau*. Melhor ainda, podem tentar converter *amor* em *ódio*, ganhando uma visão moral e lógica do processo...
- Considerem uma variação interessante das escadas de palavras: Para além da substituição de letras individuais, deixar as letras rastejarem, como lagartas, para fora da palavra, aumentando ou diminuindo as palavras de uma letra no início ou no fim.

[1]O que se segue pode ser encontrado (com mais detalhe...) em "A. K. Dewdney, *A Máquina Mágica —Um manual de Magia Computacional*, Ciência Aberta, 68, Gradiva, 1994, Cap. 21."

- Que tal complementar o dicionário de palavras com regras simples que permitam "baralhar" os caracteres? Por exemplo, uma regra da forma *aba baa* significa que o símbolo *b* pode ser deslocado uma posição à esquerda quando rodeado de *a*'s.

4.8. "Package de matemática". No âmbito das aulas práticas, foi implementado um "package de matemática" —ficheiro *math.pl* — que permite o cálculo do valor numérico de expressões matemáticas na notação usual. Com este trabalho pretende-se que aumentem a funcionalidade deste "package", principalmente no que diz respeito aos seguintes pontos:

- Tratamento de erros de *sintaxe* (dá jeito uma DCG)).
- Manejo de expressões que envolvam *variáveis* (com ou sem atribuição de valores).
- Permitir *derivação simbólica*[2] de funções reais em relação a uma variável. Salientamos alguns aspectos a ter em consideração:
  - *Simplificação* das expressões: não permitir respostas em que apareçam termos como $x + x$ ou $x + 1 + x$ quando pode escrever $2 * x$ e $2 * x + 1$.
  - *Naturalidade* na forma como as expressões podem ser escritas:
    - $2\,a\,x$ em vez de $2 * a * x$
    - $d\,(x\hat{}2)\,/dx$ em vez de algo como $deriva\,(x\hat{}2, x)$

Exemplo (esta ou outra sintaxe amigável):

```
?-calcula('{X=2+3!; y=2}, d(X^3+y)/dX * (3^2+1)+6-Z^y', R).
  R = 1926 - Z^2
```

[2] As regras de derivação básicas, implementadas em Prolog, podem ser encontradas em "Domingos Moreira Cardoso, *Programação em Lógica e Demonstração Automática de Teoremas*, Cadernos de Matemática $CM/D - 03$, pág. 54-56" ([4]).

23
Exames

1. Exame Final (14/Junho/1999)

Introdução à Programação em Lógica
Matemática Aplicada e Computação (3^o ano)
Departamento de Matemática
Universidade de Aveiro
Exame Final: 14/Junho/1999, 10:00

Dispõe de 3:00 horas para realizar este exame. O exame consta de duas partes (uma Teórica e outra Teórico-Prática) que deverá responder em folhas separadas. Leia as questões com toda a atenção e responda com calma e clareza.

Parte Teórica

1. Considere as seguintes afirmações:
   - Se um objecto A está no topo de outro objecto B então B suporta A.
   - Se A está acima de B e toca em B então A está no topo de B.
   - Uma taça está acima de um livro.
   - A taça toca no livro.

   a): Construa uma Base de Conhecimento (BC) em Prolog que traduza as afirmações acima enunciadas.

   b): Diga como colocaria a seguinte pergunta ao interpretador de Prolog:
   "O livro suporta a taça?"

   c): Justifique, por meio de uma *Árvore de Prova*, qual a resposta dada pelo interpretador à pergunta da alínea anterior.
2. Na sétima aula teórico-prática, implementámos um predicado para calcular o máximo divisor comum de dois números. O algoritmo usado foi o chamado *algoritmo de Euclides*. Vamos aqui propor um outro algoritmo, algoritmo esse recursivo, que resulta de um teorema matemático famoso, que afirma que o $m.d.c.(a,\, b) = m.d.c.(b,\, r)$, onde `r is a mod b`. O que é interessante acerca deste teorema, do ponto de vista recursivo, é que ele proporciona uma maneira de "simplificar" os dois parâmetros, de modo a que a função *m.d.c.* possa ser chamada outra vez para um caso mais simples. Com efeito, repare que `a mod b` retorna o resto da divisão, um inteiro entre 0 e o segundo argumento `b`. Se supusermos que `b` é menor que `a`, então o próximo par `(b,r)` é um par de inteiros ambos menores ou iguais a `b`, de tal modo que o máximo do par diminuiu de `a` para `b`. Por outro lado, se supusermos que é `a` menor que o `b`, então o novo par `(b,r)` é igual a `(b,a)`, uma vez que `a is a mod b` se `b` for maior que `a`: caímos então no primeiro caso em que o primeiro elemento é maior que o segundo e o próximo par terá um máximo de valor menor. Finalmente, se `a` for igual ao `b`, então o próximo par

é `(b,0)` e sabemos que o máximo divisor comum de qualquer inteiro `b` e 0 é o próprio número `b`. Uma vez que os dois parâmetros `a` e `b` são sempre inteiros não negativos, alcançamos sempre um ponto em que um dos parâmetros é zero. Por exemplo:

$$m.d.c.(15,\, 6) = m.d.c.(6,\, 3) = m.d.c.(3,\, 0) = 3.$$

a): Usando o método recursivo acima descrito, implemente o predicado `mdc/3`. Exemplo:

```
?- mdc(15,6,X).
   X = 3
```

b): Para a definição do predicado `mdc/3` que fez em **a)**, construa a *Árvore de Procura* standard e diga qual (ou quais) a(s) resposta(s) que um interpretador de Prolog daria à questão

```
?- mdc(72,30,X).
```

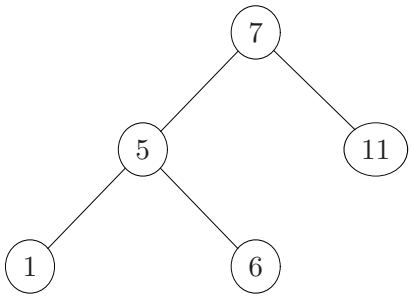
3. Considere a representação de árvores binárias em Prolog usada nas aulas Teóricas:

```
arvBin(Vertice,ArvBinEsq,ArvBinDir).
```

Por exemplo

```
arvBin( 7,
        arvBin( 5,
                arvBin(1,vazia,vazia),
                arvBin(6,vazia,vazia)
              ),
        arvBin(11,vazia,vazia)
      )
```

corresponde à árvore binária



Defina o predicado `vertices/2` que junta todos os vértices de uma árvore binária numa lista. Por exemplo:

```
?- vertices(arvBin( 7,
                    arvBin( 5,
                            arvBin(1,vazia,vazia),
                            arvBin(6,vazia,vazia)
                          ),
                    arvBin(11,vazia,vazia)
                  ),
            L).
   L = [7,5,1,6,11]
```

4. A Liga Contra os Discursos Sexistas (LCDS) pretende identificar escritores com preconceitos sexistas. Para isso pretende analisar textos na procura da palavra `Homem`. Para evitar formas subtis de machismo, também pretende identificar ocorrências disfarçadas como nas palavras `Lobishomem` (além do mais mal escrita, claro!), `Homemetro` (não existe pois não? ;-)).

   Apresente duas soluções para o problema:
   a): uma baseada num autómato finito não determinista;
   b): outra num autómato finito determinista.

5. O gerente de uma loja tradicional de tintas resolveu modernizar-se e aderir às novas tecnologias da informação. Mas como quer ele quer os seus empregados não estão

habituados a usar ferramentas informáticas, deseja que lhes seja facultado um interface com o computador em língua natural. A Joana, que é licenciada em MAC e teve a cadeira de IPL, resolveu facilmente o assunto recorrendo ao Prolog e às DCGs. Segue-se uma parte da BC desenvolvida pela Joana.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% TINTAS
%%
%% tinta(X,L,P)
%%   X - designacao, L - lata de L litros, P - preco da lata
%%
%% rentabil(X,R,L)
%%   X - designacao,
%%   R - rendimento (m2/1 passagem) em paredes rugosas,
%%   L - rendimento (m2/1 passagem) em paredes lisas.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tinta(stucomat,5,6000).
tinta(stucomat,10,11000).
tinta(stucomat,20,21000).
tinta(rep,10,15000).
tinta(rep,15,21000).
tinta(rep,20,27500).
tinta(charme,5,8000).
tinta(charme,20,37000).

rentabil(stucomat,7,10).
rentabil(rep,8,12).
rentabil(charme,10,16).

pintor(R) --> quais(R), ['?'].
pintor(R) --> qual(R), ['?'].
quais(R) --> [quais],[as], q1(R).
qual(R) --> [qual],[o],q2(R).
q1(R) --> [tintas],{rentabil(R,_,_)}.
q1(R) --> [embalagens], [de], [X], {tinta(X,R,_)}.
q2([L,P]) --> [preco],[da],[X],{tinta(X,L,P)}.
q2(R) --> [rendimento], q3(R).
q3([R,L]) --> [da],[X],{rentabil(X,R,L)}.
q3(R) --> [em],[paredes],paredes(R).
paredes([X,V]) --> [lisas],{rentabil(X,_,V)}.
paredes([X,V]) --> [rugosas],{rentabil(X,V,_)}.
```

a): Indique, justificando, qual a resposta do interpretador de Prolog a cada uma das seguintes questões (no caso de haver "várias hipóteses" force sempre o retrocesso com o `;`).

```
?- pintor(R,[quais,as,tintas,'?'],[]).
?- pintor(R,[quais,as,embalagens,de,stucomat,'?'],[]).
?- pintor(R,[qual,o,preco,da,charme,'?'],[]).
?- pintor(R,[qual,o,rendimento,da,rep,'?'],[]).
?- pintor(R,[qual,o,rendimento,em,paredes,lisas,'?'],[]).
```

b): Altere a DCG supra, de tal modo que quando haja vários resultados (acima obtidos forçando o retrocesso com o `;`) `R` seja uma lista com todos esses resultados (e deste modo já não seja preciso forçar o retrocesso).

6. Considere as seguintes duas implementações do predicado `triappend/4` que serve para concatenar três listas:

```
% append usual

append([],L,L).
append([X|R1],L,[X|R2]) :-
  append(R1,L,R2).

%%%%%%%%%%%%%
% 1a versao
%%%%%%%%%%%%%

triappend(A,B,C,L) :-
  append(A,B,L1),
  append(L1,C,L).

%%%%%%%%%%%%%
% 2a versao
%%%%%%%%%%%%%

triappend(A,B,C,L) :-
  append(B,C,L1),
  append(A,L1,L).
```

Diga, justificando, qual das duas implementações é mais eficiente. Repare que o esforço computacional do `append` depende única e exclusivamente do tamanho da primeira lista.

7. Diga, por palavras suas, o que é um Sistema Pericial.

Parte Teórico-Prática

1. Relativamente à questão **"cães"** do primeiro trabalho prático, a Joana resolveu considerar os predicados `cao/5` e `comportamento/4`:

```
% cao(Raca,Cor,Tamanho,Pelo,AlimentacaoDiariaKg)
% comportamento(Raca,Inteligencia,Obediencia,Agressividade)
```

   Dada uma BC com vários factos sobre cães e respectivos comportamentos, pretende-se obter uma "listagem" de todos os cães (raças) que verifiquem certos requisitos. Para cada uma das alíneas seguintes, explique como formularia a pergunta ao interpretador de Prolog. Se achar necessário, desenvolva predicados auxiliares.

   a): "Quais os cães que exigem uma alimentação diária inferior a 3 Kg?"

   b): "Quais os cães de tamanho médio, muito obedientes e de agressividade baixa?"

2. Relativamente aos segundos trabalhos práticos, diga, para cada uma das questões que se seguem, qual o resultado esperado.

   a): `?- arvoreNatal(5).`

   b): `?- potencias([4,2,3],3,X).`

3. Defina em Prolog o predicado `seleccionaPN/3` que separa todos números estritamente positivos e negativos de uma lista arbitrária de inteiros. Por exemplo:

```
?- seleccionaPN([0,-1,-5,5],X,Y).
   X = [5]
   Y = [-1,-5]
?- seleccionaPN([1,-2,4,2,3],X,Y).
   X = [1,4,2,3]
   Y = [-2]
```

4. A função de Ackermman define-se de uma das seguintes maneiras equivalentes:

$$\begin{array}{rcl} F_0(x) & = & x+1 \\ F_{n+1}(x) & = & \underbrace{F_n F_n \ldots F_n}_{x+1 \text{vezes}}(1) \end{array} \quad \left| \quad \begin{array}{l} F_0(x) = x+1 \\ F_{n+1}(0) = F_n(1) \\ \\ F_{n+1}(x+1) = F_n\left(F_{n+1}(x)\right) \end{array} \right.$$

Escolhendo a definição que mais lhe agradar, implemente, em Prolog, a função de Ackermman. Não se esqueça de exemplificar, com um ou dois exemplos, o funcionamento do predicado.

2. Exame de Recorrência (12/Julho/1999)

Introdução à Programação em Lógica
Matemática Aplicada e Computação (3^o ano)
Departamento de Matemática
Universidade de Aveiro
Exame de Recorrência: 12/Julho/1999, 10:00

Dispõe de 3:00 horas para realizar este exame. O exame consta de duas partes (uma Teórica e outra Teórico-Prática) que deverá responder em folhas separadas. Leia as questões com toda a atenção e responda com calma e clareza.

Parte Teórica

1. Os dois primeiros números de Fibonacci são o 1, 1 e depois continuam como a soma dos dois números precedentes: 2, 3, 5, 8, 13, 21, 34, ... Segue-se uma implementação recursiva simples do predicado `fib(N,F)`, onde `F` é o `N`-ésimo número de Fibonacci.

```
fib(1,1).
fib(2,1).
fib(N,F) :- N1 is N - 1, N2 is N - 2,
            fib(N1,F1), fib(N2,F2),
            F is F1 + F2.
```

   a): Recorrendo ao percurso-construção standard da *Árvore de Procura*, indique a resposta do interpretador à questão

```
?- fib(5,X).
```

   b): Relativamente à eficiência computacional, existem algumas questões importantes a serem consideradas com programas recursivos deste tipo. Para começar a perceber isto, tenha em conta a *Árvore de Procura* que fez na alínea anterior e responda à seguinte questão. Quantas "chamadas" a `fib(3,Var)` são feitas para responder à pergunta `?- fib(5,X).`

   c): Para a pergunta

```
?- fib(7,X).
```

   diga, mostrando a *Árvore de Prova* (não se pretende que mostre as *Árvores de Prova Parciais* das etapas intermédias, apenas a *Árvore de Prova*), quantas "chamadas" a `fib(5,Var)`, `fib(4,Var)` e `fib(3,Var)` são feitas.

   d): Apresente, por palavras, um método para melhorar a eficiência computacional do predicado `fib`. (Não é necessário implementar a ideia em Prolog.)

2. Dizemos que uma árvore binária `arvBin(V,ABesq,ABdir)`, com números nos `V`értices, é *ordenada* se: todos os vértices na subárvore `ABesq` são menores que `V`; todos os vértices na subárvore `ABdir` são maiores do que `V`; ambas as subárvores são igualmente ordenadas. A vantagem das árvores binárias ordenadas é que a procura de um número na árvore é muito mais eficiente. Defina o predicado `pertence(V,ABO)` que é verdadeiro se e somente se `V` pertencer à árvore binária ordenada `ABO`.
3. Modele, por meio de um autómato finito, uma máquina de pagamento dos parques de estacionamento. Para simplificar, considere que, independentemente do tempo, o pagamento a fazer é de 50$00. Aceitam-se moedas de 10, 20 e 50$00. Não se esqueça de indicar no seu esquema as acções semânticas que achar pertinentes.
   Usando o simulador de Prolog desenvolvido nas aulas teóricas, crie uma BC com a descrição do autómato que considerou e dê três exemplos de simulação: um de aceitação sem troco; um outro de aceitação com troco; e por fim um exemplo de rejeição.
4. A Joana, aluna de IPL muito interessada nestas "coisas" do Prolog, decidiu dar uso à sua hábil capacidade de resolver problemas e implementar o tradicional **Jogo do Galo**. Para começar, implementou o predicado `galo` de aridade 2, que identifica as posições

(linhas, colunas e/ou diagonais) que levaram certo jogador à vitória. Por exemplo para a situação

o	o	x
o	x	o
x	x	x

temos

```
?- galo(x,[[o,o,x],[o,x,o],[x,x,x]]).
  Linha 3
  Diagonal 2
  Yes
?- galo(o,[[o,o,x],[o,x,o],[x,x,x]]).
  Yes
```

Eis o programa da Joana:

```
galo(X,L):- linhas(X,L), colunas(X,L), diagonais(X,L).

linhas(X,L):- analisa('Linha ',1,X,L).

colunas(X,L) :- forma_colunas(L,LC), analisa('Coluna ',1,X,LC).

diagonais(X,L) :- forma_diagonais(L,LD), analisa('Diagonal ',1,X,LD).

analisa(_,_,_,[]).
analisa(S,P,X,[[X,X,X]|R]) :-
  write(S), write(P), nl,
  P1 is P + 1,
  analisa(S,P1,X,R).
analisa(S,P,X,[_|R]) :-
  P1 is P+1,
  analisa(S,P1,X,R).

forma_colunas([],[]).
forma_colunas([X|R],L) :-
  forma_colunas(R,L1),
  adiciona(X,L1,L).

adiciona([],L,L).
adiciona([X|R1],[],[[X]|R2]) :- adiciona(R1,[],R2).
adiciona([X|R],[L1|R1],[L2|R2]) :-
  append([X],L1,L2),
  adiciona(R,R1,R2).

forma_diagonais(L,[D1,D2]) :- fd(1,L,D1),
                              inverte(L,LI),
                              fd(1,LI,D2).

fd(_,[],[]).
fd(N,[X|R],L) :-
  N1 is N+1,
  fd(N1,R,L1),
  nth1(N,X,E),       % predicado pre' definido: retorna o N-e'simo
  append([E],L1,L).  % elemento E da lista X (1o elemento na posicao 1)

inverte([],[]).
inverte([X|R],[XI|RI]) :-
  reverse(X,XI),     % predicado pre' definido: inverte uma lista
  inverte(R,RI).
```

a): Indique a resposta a cada uma das seguintes questões:

```
?- analisa('ola ',2,2,[[2,2],[2,2,2],[1,1,1]]).

?- adiciona([ola,sol,mar],[[1,2],[3,4],[5,6]],L).

?- forma_colunas([[qual,o],[resultado,disto]],L).

?- inverte([[1,2,ola],[vez,3]],L).

?- fd(2,[[1,2,3],[4,5,ola]],L).

?- galo(x,[[x,x,o],[o,x,o],[x,x,x]]).
```

b): O programa da Joana é bastante elegante: com uma pequeníssima alteração, o programa é capaz de funcionar para jogos do galo em tabuleiros `NxN`. Proceda a essa alteração. (Basta alterar uma única cláusula ...)

c): Visto que nem todas as possibilidades de listas da forma `[ [x,o,x,...], ...]` são válidas para um Jogo do Galo, pretende-se que implemente o predicado `verifica(S,LL)` que recebe o símbolo `S` do jogador que iniciou o jogo e a lista `LL` de configuração e indica se a lista é ou não válida. Não se esqueça que `LL` é válida quando:

- `LL` for uma lista de `N` listas, cada uma delas de comprimento `N`.
- Para um tabuleiro `NxN`, se `N` é par devem existir tantos "`x`" quantos "`o`", se `N` é ímpar deverá existir mais um símbolo do jogador que iniciou o jogo.
- As listas apenas contêm os símbolos `x, o`.

Pode pressupor (apenas) a existência dos seguintes predicados: `length(L,C)`, que retorna o comprimento `C` da lista `L`; `todasComp(LL,C)`, que resulta verdadeiro sse todas as listas de `LL` (`LL` é uma lista de listas) tiverem comprimento `C`; `par(N)`, que resulta verdadeiro quando `N` é um número par; e `conta(S,LL,N)`, que devolve o número `N` de ocorrências do símbolo `S` na lista de listas `LL`.

5. **a):** Construa uma DCG de tal modo que seja possível a seguinte interacção com o interpretador de Prolog:

```
?- interface([inicia,o,jogo],[]).
   Yes

?- interface(
     [qual,o,simbolo,na,posicao,'(',1,',',1,')','?'],
     []
   ).
   Yes

?- interface([qual,o,numero,de,pecas,de,'o','?'],[]).
   Yes

?- interface([qual,o,numero,de,casas,que,faltam,'?'],[]).
   Yes

?- interface([acaba,o,jogo],[]).
   Yes
```

b): Acrescente à DCG que definiu na alínea anterior uma acção semântica, e os argumentos que achar necessários, de tal modo que quando fizermos uma questão do tipo *"Qual o símbolo na posição X,Y?"* seja escrito no ecrã o respectivo símbolo ou então uma mensagem de erro. Por exemplo:

```
?- interface([[o,o,x],[o,x,o],[x,x,x]],
             [qual,o,simbolo,na,posicao,'(',1,',',1,')','?'],
             []
           ).
o
Yes
```

```
?- interface([[o,o,x],[o,x,o],[x,x,x]],
              [qual,o,simbolo,na,posicao,'(',3,',',1,')','?'],
              []
             ).
   x
   Yes
?- interface([[o,o,x],[o,x,o],[x,x,x]],
              [qual,o,simbolo,na,posicao,'(',4,',',1,')','?'],
              []
             ).
   Posicao invalida!
   Yes
```

6. Diga quais as principais diferenças entre um Sistema Pericial e um programa informático convencional.

Parte Teórico-Prática

1. Relativamente aos segundos trabalhos práticos, diga, para cada uma das questões que se seguem, qual o resultado esperado.
   a): `?- diamante(3).`
   b): `?- triangulo([a,b,c,d]).`
   c): `?- serpente([a,b,c,d],[*,*,*,*],[*,*,*]).`
2. Já que parece estar em moda a (perversa) *Tolerância Zero*, defina em Prolog os seguintes predicados:
   a): `zerosConta/2` que conta o número de zeros de uma dada lista. Por exemplo:
   ```
   ?- zerosConta([1,0,0,5],X).
      X = 2
   ?- zerosConta([],X).
      X = 0
   ```
   b): `zerosElimPre/2` que elimina todos os zeros no início de uma lista. Por exemplo:
   ```
   ?- zerosElimPre([0,0,0,4,5],X).
      X = [4,5]
   ?- zerosElimPre([4,0,5],X).
      X = [4,0,5]
   ```
   c): `zerosElimPos/2` que elimina todos os zeros no fim de uma lista. Por exemplo:
   ```
   ?- zerosElimPos([4,5,0,0,0],X).
      X = [4,5]
   ?- zerosElimPos([4,0,5],X).
      X = [4,0,5]
   ```
   d): `zerosElim/2` que elimina todos os zeros de uma lista. Por exemplo:
   ```
   ?- zerosElim([0,4,0,5,0,0],X).
      X = [4,5]
   ?- zerosElim([4,5],X).
      X = [4,5]
   ```
3. Defina em Prolog o predicado `incrementa(NumInic,Inc,N,L)`, que devolve em `L` a lista
   `[Num1, Num2, ..., NumN]`
   onde $Num_i = NumInic + (i-1) * Inc, \quad 1 \leq i \leq N$.

Bibliografia

[1] Arity Corporation, *The Arity/Prolog Compiler and Interpreter*, Version 6.1, 1992. Chapter 13: Definite Clause Grammar Notation, pp. 339–348.

[2] I. Bratko, *Prolog: programming for Artificial Intelligence*, Addison Wesley, 1990.

[3] W. D. Burnham and A. R. Hall, *Programação e Aplicações em Prolog*, Editorial Presença, 1987.

[4] D. M. Cardoso, *Programação em Lógica e Demonstração Automática de Teoremas*, Cadernos de Matemática, CM/D-03, Dep. de Matemática da U.A., 1995.

[5] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 2nd ed., 1984.

[6] H. Coelho and J. C. Cotta, *Prolog by Example, How to Learn, Teach and Use It*, Springer-Verlag, 1988.

[7] R. G. Crespo, *Processadores de linguagens, da concepção à implementação*, IST Press, 1998.

[8] P. Deransart, *Initiation à Prolog*, Univ. Orleans, 1986.

[9] P. Lucas and L. D. Gaag, *Principles of Expert Systems*, Addison Wesley, 1991.

[10] D. Merritt, *Building Expert Systems in Prolog*, Springer Verlag, 1989.

[11] C. Nikolopoulos, *Expert Systems, Introduction to First and Second Generation and Hybrid Knowledge Based Systems*, Marcel Dekker, 1997.

[12] D. W. Patterson, *Introduction to Artificial Intelligence & Expert Systems*, Prentice Hall, 1990.

[13] J. L. Pereira, *Tecnologia de Bases de Dados*, Tecnologias de Informação, FCA, 2^a Ed., 1998.

[14] N. C. Rowe, *Artificial Intelligence through Prolog*, Prentice-Hall, 1988.

[15] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT-Press, 1986.

[16] A. M. Tenenbaum and M. J. Augenstein, *Data Structures using Pascal*, Prentice-Hall, Cap. 2.

[17] D. A. Waterman, *Building expert systems*, Addison Wesley, 1983.

[18] D. A. Waterman, *A guide to expert systems*, Addison Wesley, 1986.

[19] D. Wood, *Theory of Computation*, Harper & Row.

Índice