Paradigmas da Programação III

Apontamentos Teórico-Práticos
(LESI/LMCC)

José Creissac Campos, António Nestor Ribeiro
{jose.campos, anr}@di.uminho.pt

DI/UM

2004/05

Conteúdo

Lista de Figuras

Lista de Tabelas

Ficha Prática 1

1.1 Objectivos

1. Aprender a trabalhar com o interpretador.
2. Fazer interrogações à informação existente.

1.2 Conceitos

1.2.1 Prolog

Prolog (*PROgramming in LOGic*) é uma linguagem declarativa para computação simbólica:

- Um programa em Prolog não descreve como *calcular* a solução de um dado problema.
- Um programa em Prolog consiste numa base de dados de factos e relações lógicas (regras) que descrevem o problema (também chamada de Base de Conhecimento).

Em vez de *correr* o programa para obter a solução, o utilizador faz uma pergunta. Quando uma pergunta é colocada, o sistema efectua uma procura na base de dados de factos e regras até determinar (por dedução lógica) a resposta.

1.2.2 Átomos, variáveis e números

Átomos podem ser construídos de três formas:

1. *strings* de letras, dígitos e *underscores* começando por uma letra minúscula:

```
rui
mario
tcp4
t_3
t___
parad_prog_3
```

2. *strings* de caracteres especiais:

```
<...>
===>
..:..
:::::
```

3. *strings* de carateres entre plicas:

```
'daytona'
'kafka'
```

Variáveis são sequências de letras começadas por letra maiúscula ou por um *underscore*.

```
X
Y
_X
NomeVar
```

Números em Prolog incluem os números inteiros e os reais. No entanto os números reais não costumam ser muito utilizados, uma vez que o Prolog é antes de mais uma linguagem adequada à computação simbólica.

1.2.3 Factos e regras

O facto de que o Mário é pai do Manuel pode ser escrito em Prolog da seguinte forma:

```
pai(mario, manuel).
```

Neste exemplo `pai` é o nome da relação, `mario` e `manuel` são os argumentos. Note que `mario` e `manuel` foram escritos com letra minúscula. Porquê? Consegue pensar numa alternativa?

Se o Mário é pai do Manuel, então o Manuel é filho do Mário. Podemos dizer que:

> Para todo o A e B, A é filho de B se B é pai de A.

Este conhecimento pode ser descrito pela expressão:

```
filho(A,B) :- pai(B,A).
```

a que se chama uma regra.

A principal diferença entre factos e regras é que os factos expressão relações que são sempre verdadeiras, enquanto as regras definem que uma relação é verdadeira em determinadas condições. Neste caso se a condição `pai(B,A)` (o corpo da regra) for verdade, então podemos concluir que filho(A,B) (a cabeça da regra) é verdade.

Um conjunto de factos e regras com o mesmo nome definem um predicado (ou procedimento). Neste caso definimos os predicados pai/2 e filho/2 (os números após a barra indicam o número de parâmetros do predicado).

1.3 Exemplo

Consideremos a árvore genealógica representada na figura 1.1. A árvore representa informação sobre o grau de parentesco entre diversos indivíduos. Várias conclusões se podem extrair da figura: o Mário é pai do Manuel, o Mário é pai da Teresa, o João é filho do Manuel, o Manuel é avô da Maria, etc. Se escolhermos a relação "*ser pai de*" como a relação relevante a representar, o conhecimento adquirido a partir da figura leva à existência dos seguintes ***factos***:

```
pai(mario, manuel).
pai(mario, teresa).
pai(manuel, joao).
pai(joao, maria).
pai(joao, rui).
```

Este conhecimento é aquele que é explícito a partir da figura. É agora possível fazer perguntas e obter mais conhecimento a partir da informação existente. Comece por escrever os factos no ficheiro `bd.swi`[1].

[1]Tradicionalmente a extensão de ficheiros Prolog é ".pl". No entanto, em alguns sistemas isso pode causar conflito com o Perl pelo que, durante a instalação, o SWI-Prolog permite configurar a extensão a utilizar. Nestes apontamentos iremos utilizar ".swi".

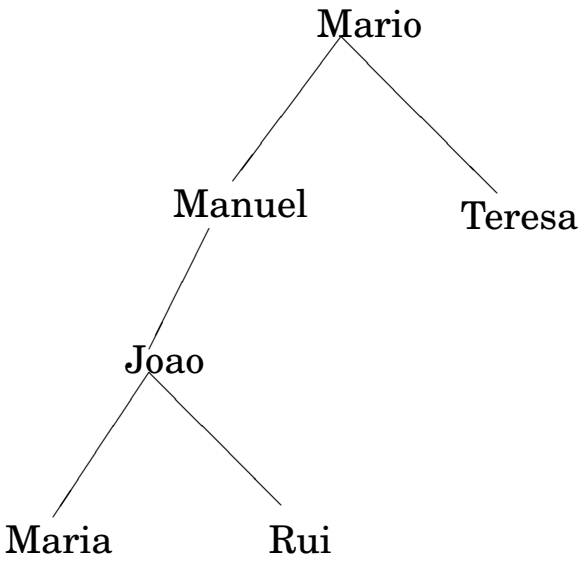

Figura 1.1: Árvore Genealógica

1.3.1 SWI-Prolog

O interpretador de Prolog que iremos utilizar é o SWI-Prolog (encontra-se disponível em `http://www.swi-prolog.org`). Ao instalar o SWI-Prolog em Windows preste atenção à directoria de trabalho que define, será essa a directoria em que os ficheiros serão procurados por omissão (é possível mudar de directoria de trabalho com o predicado `cd/1`).

- Arrancar com o interpretador:
  - Windows: procure a entrada apropriada no menu Start/Iniciar (nos laboratórios pode também procurar o ficheiro `pl.exe` em `t:\bin\pl\bin\plwin`).
  - Linux: execute o comando `pl`.

  O *prompt* "`| ?-`" significa que o interpretador está à espera de instruções.

- Carregar um ficheiro

  ```
  | ?- consult('bd.swi').
  ```

  ou,

  ```
  | ?- consult(bd). %% bd corresponde ao nome do ficheiro sem extensão
  ```

  ou,

  ```
  | ?- [bd].
  ```

- Visualizar o conhecimento existente na base de dados:

  ```
  | ?- listing.

  pai(mario, manuel).
  pai(mario,teresa).
  pai(manuel, joao).
  ```

```
pai(joao, maria).
pai(joao, rui).

yes
```

1.3.2 Interrogações à Base de Conhecimento

É então possível interrogar o sistema de maneira a extrair informação da Base de Conhecimento. Por exemplo:

- Verificar se o `mario` é pai do `manuel`.

```
| ?- pai(mario,manuel).

yes
```

- Verificar se o `rui` é pai do `joao`.
- Verificar se a `mario` é pai da `teresa`.

Ou ainda fazer perguntas mais complexas como:

- Determinar os filhos do `mario`.

```
| ?- pai(mario,X).

X = manuel ? ;    %% ";" dá a resposta seguinte no caso de existir

X = teresa ? ;

no
| ?-
```

- Quem é o pai da `teresa`?
- Os pares pai/filho existentes na Base de Conhecimento.
- Verificar se o pai do `rui` é o mesmo que o pai da `maria`.

```
| ?- pai(X,rui),pai(X,maria).
```

Repare que a ","significa a conjunção de termos. Tem a mesma semântica que o AND (ou $\wedge$).

- Quem é o avô do `joao`.

```
| ?- pai(X,joao),pai(Y,X).

X = manuel,
Y = mario ? ;   %% Y é o avô

no
```

1.4 Exercícios

1.4.1 Socorro!

Experimentar os predicados `help` e `apropos`:

1. Depois de iniciar o interpretador, escreva a *query* "`apropos(help).`" (não esqueça o ponto — '.')
   Note que o predicado `apropos` lhe fornece uma lista de predicados e secções do manual relacionados com o assunto indicado como parâmetro. (Se estiver a utilizar a versão 4.0 ou superior do SWI Prolog, essa lista aparecerá numa nova janela — ver figura 1.2.)

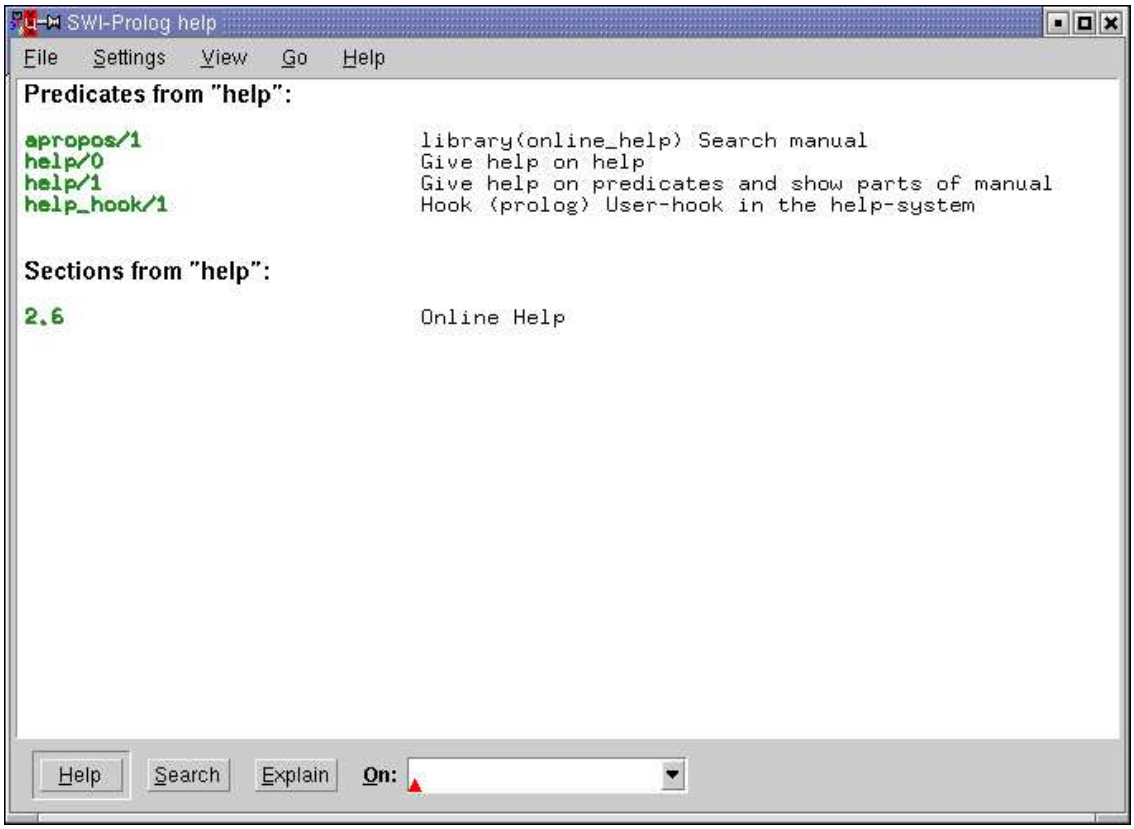

Figura 1.2: Janela de *Help* do SWI-Prolog

2. Experimente agora a *query* "`help(help).`". (Se estiver a utilizar a versão 4.0 ou superior, poderá utilizar a área de diálogo que aparece na janela de *help*.)
   Note que o predicado `help` lhe fornece ajuda sobre os predicados indicados como parâmetro.
3. Experimente as *queries* "`help(1).`" e "`help(2-1).`".
   Note que quando o parâmetro do predicado `help` é um número, é apresentada a secção do manual com esse número. Na verdade tem todo o manual disponível *online* para consulta!
4. Finalmente, experimente a *query* "`help(halt).`".

1.4.2 Factos, *queries* e regras

1. Escreva os seguintes factos (pode utilizar "`[user].`" ou então escrever os factos num ficheiro e utilizar `consult/1`):

```
aluno(joao,ppi).
aluno(pedro,ppi).
aluno(maria,ppiii).
aluno(rui,ppiii).
aluno(manuel,ppiii).
aluno(pedro,ppiii).
aluno(rui,ppiv).
```

(a) Verifique que os factos estão presentes na Base de Conhecimento (utilize o predicado `listing`).

(b) Escreva uma *query* que verifique se `joao` é aluno de `ppiii`.

(c) Escreva uma *query* que verifique se `rui` é aluno de `ppi` — note o efeito do Princípio do Mundo Fechado.

(d) Escreva uma *query* que verifique se `joao` e `maria` são ambos alunos de `ppiv` — `joao` e `maria` são ambos alunos de `ppiv` se `joao` for aluno de `ppiv` e `maria` for aluna de `ppiv`.

(e) Escreva uma *query* que permita saber quem é aluno de `ppiii`.

(f) Escreva uma *query* que permita saber de que disciplinas é `rui` aluno.

2. Adicione os seguintes factos à Base de Conhecimento (se anteriormente utilizou "`[user].`" é agora uma boa altura para começar a utilizar ficheiros):

```
estuda(joao).
estuda(maria).
estuda(manuel).
```

(a) Sabendo que a aluno A faz a disciplina P se A é aluno de P e A estuda, escreva uma *query* que lhe permita saber se `maria` vai fazer `ppiii`.

(b) Experimente agora a *query* "`aluno(X,ppiii),estuda(X).`". O que lhe permite esta *query* saber?

(c) Utilizando a *query* da alínea anterior, acrescente à Base de Conhecimento o predicado `fazppiii/1` e escreva uma *query* que lhe permita saber quem faz `ppiii` (não se esqueça de fazer `consult` do ficheiro).

Ficha Prática 2

2.1 Objectivos

1. Praticar a escrita de Bases de Conhecimento, *queries* e predicados.

2.2 Conceitos

Uma base de conhecimento representa a informação que se possui sobre um dado problema. Nos exercícios que se seguem, comece por analisar qual a informação que possui e de que forma a pode representar em Prolog. A este nível a informação será representada essencialmente através de factos. Mais tarde começaremos a utilizar regras de uma forma mais extensiva.

Relativamente a regras, relembre o último exercício da Ficha Prática 1. O predicado comAprovação/1, que sucede quando a disciplina passada como parâmetro tem alunos que passam, pode ser definido pela seguinte regra Prolog:

```
comAprovacao(Disc) :- aluno(Alguem, Disc), estuda(Alguem).
```

A regra é equivalente à seguinte expressão matemática:

$$\forall_{Disc} \cdot \text{comAprovacao}(Disc) \leftarrow \exists_{Alguem} \cdot \text{aluno}(Alguem, Disc) \wedge \text{estuda}(Alguem)$$

2.3 Exercícios

2.3.1 Coloração de mapas

A coloração de mapas é um problema famoso da matemática. O objectivo é garantir que não existam regiões adjacentes com a mesma cor num dado mapa.

Considere o mapa apresentado na figura 2.3.

1. Escreva uma base de conhecimento que represente a informação presente no mapa. (Como se pode deduzir pelas questões que se seguem, a informação relevante consiste em saber que regiões são adjacentes de que regiões.)
2. Escreva *queries* que lhe permitam saber:
   (a) se a região **a** é adjacente da região **d**;
   (b) se a região **a** é adjacente da região **e**;
   (c) quais as regiões adjacentes da região **c**.
3. Considere agora o mapa apresentado na figura 2.4.
   (a) Acrescente à base de conhecimento factos que indiquem a cor de cada região.
   (b) Escreva uma *query* que teste se as regiões **b** e **d** estão em conflito.

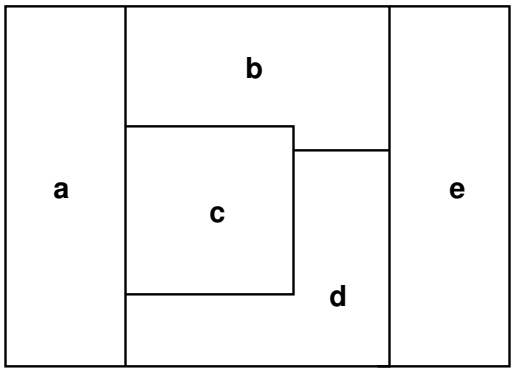

Figura 2.3: Mapa por colorir

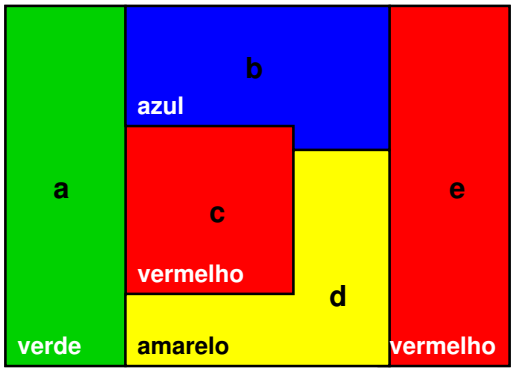

Figura 2.4: Mapa colorido

(c) Escreva o predicado conflito/0 que deverá suceder caso exista um conflito no mapa, teste-o.

(d) Altere a base de conhecimento de modo a que a região **b** seja amarela, teste novamente o predicado conflito/0.

(e) Escreva agora o predicado conflito/2 que tem como parâmetros os identificadores de duas regiões e sucede caso essas regiões estejam em conflito. Utilize-o em diferentes versões do mapa para saber:

i. se as regiões **b** e **d** estão em conflito;

ii. se alguma região está em conflito com a região **d**;

iii. se existem regiões em conflito.

2.3.2 Árvore Genealógica [2]

Considere a seguinte Base de Conhecimento:

```
% filho(A,B) :- A é filho de B.
filho(jose,francisco).
filho(ana,francisco).
filho(madalena,francisco).
filho(francisco,domingos).
filho(rui,abilio).
```

```
filho(antonio,abilio).
filho(abilio,domingos).
filho(augusto,manuel).
```

1. Escreva uma *query* que permita saber se `rui` é filho de `francisco`.
2. Escreva uma *query* que permita saber todos os filhos do `francisco`.
3. Adicione o predicado `pai/2` à Base de Conhecimento. Escreva uma *query* que permita saber quem é o pai de `rui`.
4. Adicione o predicado `tio/2` à Base de Conhecimento. Escreva uma *query* que permita saber de quem `francisco` é tio.
5. Adicione o predicado `primo/2` à Base de Conhecimento. Escreva uma *query* que permita saber quem é primo de quem.
6. Adicione o predicado `avo/2` à Base de Conhecimento. Escreva uma *query* que lhe permita saber de quem `domingos` é avô.
7. Adicione o predicado `descendente/2` à Base de Conhecimento. Escreva uma *query* que lhe permita saber quem é descendente de quem.

2.3.3 Controlo de Tráfego

Considere a seguinte informação relativa a partidas e chegadas no aeroporto internacional de Braga[2].

Partidas:

Voo	Destino	Hora Prev.	Hora Real
TP123	Lisboa	14h30	14h30
NI234	Manchester	15h25	16h00
TP876	Faro	14h18	14h30
NI498	Madrid	15h00	15h00

Chegadas:

Voo	Origem	Hora Prev.	Hora Real
TP123	Lisboa	14h00	14h35
NI533	Funchal	15h00	15h00
TP877	Santiago	14h30	15h00
NI498	Manchester	16h00	15h50

1. Modele a informação representada nas tabelas.
2. Escreva e teste os seguintes predicados:
   (a) parteAHoras/1 — sucede se a hora real de partida do voo indicado como parâmetro for a hora inicialmente prevista;
   (b) vaivem/1 — sucede se o voo indicado como parâmetro efectua viagens de, e para, uma mesma cidade;
   (c) ligacao/2 — sucede se existe um voo que chega da cidade indicada como primeiro parâmetro e parte para a cidade indicada como segundo parâmetro (a cidade destino da ligação não deverá ser a mesma que a cidade origem);
   (d) chegaAtrasado/1 — sucede se a hora real de chegada do voo indicado como parâmetro for posterior à hora inicialmente prevista;
   (e) emConflito/1 — este predicado testa, para os voos em que esteja prevista uma partida e uma chegada, se a hora real de partida é anterior à hora real de chegada;

[2]Em fase de planeamento!

2.3.4 Alice na Floresta do Esquecimento

Considere a seguinte história:

> A Alice tinha má memória. Um dia entrou na Floresta do Esquecimento e esqueceu-se do dia-da-semana. Os seus amigos Coelho e Cuco são visitantes frequentes da floresta. Estes dois são criaturas estranhas. O Coelho mente às Segundas, Terças e Quartas e diz a verdade no resto da semana. Por outro lado o Cuco mente às Quintas, Sextas e Sábados e diz a verdade nos outros dias. Um certo dia a Alice encontrou estes dois debaixo de uma árvore. Eles fizeram as seguintes declarações:
>
> > Coelho: ontem foi um dos dias em que eu menti.
> >
> > Cuco: ontem foi um dos dias em que eu menti.
>
> A Alice foi capaz, usando estas declarações, de deduzir o dia-da-semana em que se encontrava.

1. Escreva uma Base de Conhecimento que represente o conhecimento descrito nesta história.
2. Escreva um predicado `diadehoje/1` que lhe permita saber qual o dia-da-semana usando o conhecimento representado.

Ficha Prática 3

3.1 Objectivos

1. Praticar a escrita de predicados.

3.2 Conceitos

3.2.1 Definição de Operadores

A declaração:

```
:-op(700, xfy, impl)
```

no início de um ficheiro Prolog, define o operador `impl` (terceiro parâmetro de `op`).

O primeiro parâmetro define a precedência do operador. A precedência pode ser um número entre 1 e 1200. Uma precedência de 0 remove o operador. Quanto menor o número, maior a precedência do operador. Neste caso, o operador `impl` tem precedência 700.

O segundo parâmetro define o tipo do operador. O tipo pode ter um dos seguintes valores: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` ou `fx`. O `f` indica a posição do functor (infixa ou prefixa). O `x` e o `y` indicam a posição dos argumentos. O `x` lê-se: 'nesta posição deve ocorrer um termo com precedência estritamente inferior à do functor'. O `y` lê-se: 'nesta posição deve ocorrer um termo com precedência inferior ou igual à do functor'. A precedência de um termo é 0, excepto se o seu principal functor for um operador, caso em que a precedência do termo é a precedência do operador. Um termo entre parêntesis tem precedência 0. A utilização do x e do y permite controlar a forma como as precedências são aplicadas a uma expressão. Neste caso o operador `impl` é um operador infixo.

O terceiro parâmetro, tal como já foi mencionado, é o nome do operador. Este parâmetro pode também ser uma lista de nome[3]. Neste caso, todos os operadores da lista passam a ter a mesma precedência e tipo.

Com a declaração acima é agora possível escrever termos da forma:

termo1 `impl` *termo2*

Esta possibilidade ser-lhe-á útil na resolução do exercício 3.3.3.

Alguns operadores do SWI Prolog são apresentados na tabela 3.1.

3.3 Exercícios

3.3.1 Contactos

Considere agora uma nova Base de Conhecimento contendo os predicados `telefone/2`, `visita/2` e `emCasa/1`:

[3]Listas serão tema de uma Ficha Prática futura.

Precedência	Tipo	Nome do operador
1200	xfx	:-
1200	fx	:-, ?-
1100	xfy	;
1000	xfy	,
700	xfy	<, >, =<, >=
500	yfx	+, -
500	fx	+, -
400	yfx	*, /, //

Tabela 3.1: Alguns operadores SWI-Prolog

```
% telefone(P, T) :-
%   o n. de telefone da casa da pessoa P é T.
telefone(ana, 123).
telefone(ze, 234).
telefone(rui, 345).
telefone(pedro, 456).
telefone(marta, 567).
telefone(olga, 678).


% visita(X, Y) :-
%   a pessoa X está de visita à pessoa Y.
visita(olga, ana).
visita(marta, ze).
visita(rui, olga).
visita(pedro, olga).

% emCasa(X) :- X está em casa.
emCasa(ze).
emCasa(ana).
```

1. Escreva uma *query* que determine se `ana` está a visitar alguém.
2. Escreva uma *query* que determine se `ana` tem visitas.
3. Sabendo que uma pessoa P está acompanhada se tem visitas, acrescente à Base de Conhecimento o predicado `acompanhada/1`.
4. Acrescente à base de conhecimento o predicado `inconsistente/0` que determina se, na base de conhecimento, existe alguém que está simultaneamente em casa e a visitar alguém.
5. Supondo que quando alguém sai para fazer uma visita leva consigo todos os que o estão a visitar[4], acrescente à Base de Conhecimento o predicado `em_casa_de/2` que lhe permite determinar se uma pessoa está em casa de outra.
6. Acrescente à Base de Conhecimento o predicado `contacto/2` que lhe permite determinar qual o número de telefone em que cada pessoa está contactável[5].
7. Sabendo que três ou mais pessoas numa casa correspondem a uma festa, escreva um predicado `a_dar_festa/1` que determina se uma pessoa está a dar uma festa.

[4]Exemplos: O Rui em casa da Ana: o Rui está a visitar a Olga, como a Olga está a visitar a Ana, então o Rui foi com a Olga para a casa da Ana.

[5]Assuma um mundo em que ainda não existem telemóveis!

3.3.2 Mapa de Acessibilidades

1. Considere o mapa da figura 3.5, que indica os tipos de ligações possíveis entre diversas cidades.

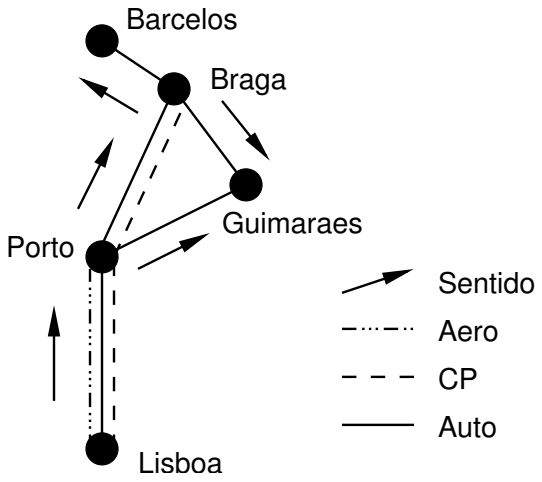

Figura 3.5: Mapa de acessibilidades

   (a) Escreva uma Base de Conhecimento que expresse a informação contida no mapa (utilize o predicado `ligacaodirecta/3` em que `ligacaodirecta(O,D,T)` se existe ligação pelo meio de transporte `T` entre `O` e `D`).

   (b) Escreva os seguintes predicados:

      i. `ha_ligacao/2` — `ha_ligacao(A,B)` sucede se existe ligação entre as cidades A e B.

      ii. `ha_ligacao/3` — `ha_ligacao(A,B,T)` sucede se é possível viajar entre as cidades A e B usando apenas o meio de transporte T.

2. Considere o predicado:

```
ha_ligacao_aux(A,B) :- ha_ligacao(A,B,_).
```

   Identifique e discuta as diferenças existentes entre os predicados `ha_ligacao/2` e `ha_ligacao_aux/2`

3. Cada vez mais os meios de transporte modernos fornecem formas de chegar cada vez mais depressa a zonas cada vez mais congestionadas. Considere que na Base de Conhecimento acima é acrescentado o predicado `nao_engarrafado/2`, indicando que numa dada cidade um dado tipo de meio de transporte não se encontra engarrafado. Redefina os predicados definidos anteriormente de modo a apenas considerar ligações que passem por cidades onde os meios de transporte a utilizar não estão engarrafados (considere que o engarrafamento só afecta quem quer entrar na cidade).

3.3.3 Demonstrador de Teoremas

Escreva um demonstrador de teoremas para o cálculo proposicional.

O demonstrador deverá ser capaz de lidar com equivalências ($\leftrightarrow$), implicações ($\rightarrow$), disjunções ($\vee$), conjunções ($\wedge$) e negação ($\neg$). Utilizando `op/3` defina os seguintes operadores (atribuindo-lhes tipos e precedências apropriados):

- `equiv` — para a equivalência;
- `impl` — para a implicação;
- `ou` — para a disjunção;
- `e` — para a conjunção;
- `nao` — para a negação.

Defina o demonstrador através do predicado `prova/1`. Teste o demonstrador com os seguintes exemplos:

```
?- prova(falso impl verdade).

Yes
?- prova(verdade impl falso).

No
?- prova((falso equiv verdade) equiv falso).

Yes
?- prova(verdade impl X).

X = verdade

Yes
?- prova(falso impl X).

X = _G155
```

Ficha Prática 4

4.1 Objectivos

1. Perceber o operador de unificação.
2. Praticar a escrita de predicados envolvendo expressões aritméticas.

4.2 Conceitos

4.2.1 Unificação

A unificação é a operação mais importante que o interpretador de Prolog realiza. A unificação representa-se pelo símbolo "=" (`A=B` lê-se: `A` unifica com `B`) e é essencialmente um processo de comparação. Este processo é imediato para atómos e números. Dois átomos unificam se e só se são o mesmo átomo. Do mesmo modo, dois números unificam se e só se são o mesmo número. Por exemplo, a unificação `ola=ola` sucede, a unificação `3=2` não sucede.

Para termos mais complexos a unificação é efectuada tentando unificar os sub-termos que os compõem. Se algum dos sub-termos for, por sua vez, um termo complexo, o algoritmo é aplicado recursivamente até o processo de unificação ficar reduzido a comparação de termos elementares. Por exemplo, a unificação `hora(12,45)=hora(12,45)` sucede pois as unificações `hora=hora`, `12=12` e `45=45` todas sucedem.

Se os termos a unificar possuirem variáveis não instanciadas, estas são instanciadas durante o processo de unificação. Na verdade, a unificação é o único modo pelo qual as variáveis são instanciadas. Se um dos termos a unificar é uma variável não instanciada e o outro um átomo, a unificação sucede e a variável é instanciada com o átomo. Se os dois termos a unificar forem variáveis não instanciadas, elas unificam e os nomes das duas variáveis tornam-se sinónimos para a mesma variável (não instanciada). Por exemplo, a unificação `hora(12,45)=hora(12,M)` sucede e a variável `M` fica instanciada com `45`. A unificação `hora(12,M)=hora(12,N)` sucede e as variáveis `M` e `N` passam a ser a mesma variável. Já a expressão `hora(12,M)=hora(12,N), M=12, N=5` não sucede. Porquê?

4.2.2 Aritmética em Prolog

Tal como descrito na secção anterior, o operador de unificação procede à comparação de termos e efectua as instanciações necessárias para que essa comparação suceda. O operador é simbólico pelo que não efectua qualquer interpretação dos termos em causa. Assim, a unificação `A=2+1` sucede com `A` a ficar instanciado com a expressão `2+1` (e não com o valor `3`) e a unificação `3=2+1` falha pois os dois termos (`3` e `2+1`) são diferentes.

O cálculo aritmético em Prolog é efectuado utilizando um conjunto pré-definido de predicados artiméticos que manipulam expressões aritméticas. A tabela 4.2 apresenta uma lista de predicados aritméticos do SWI-Prolog[6].

[6] A indicação "+", "−" e "?" nos argumentos tem o seguinte significado: "+" — o argumento deve estar instanciado; "−" — o argumento deve ser uma variável não instanciada; "?" — o argumento pode, ou não, estar instanciado.

Predicado	Significado
`between(+Min, +Max, ?Valor)`	sucede se Min<Valor<Max
`succ(?Int1, ?Int2)`	sucede se Int2 = Int1 + 1
`plus(?Int1, ?Int2, ?Int3)`	sucede se Int3 = Int1 + Int2 (pelo menos dois argumentos devem estar instanciados)
`+Expr1 > +Expr2`	sucede se o valor da expressão Expr1 for maior que o valor da expressão Expr2
`+Expr1 < +Expr2`	sucede se o valor da expressão Expr1 for menor que o valor da expressão Expr2
`+Expr1 >= +Expr2`	sucede se o valor da expressão Expr1 for maior ou igual ao o valor da expressão Expr2
`+Expr1 =< +Expr2`	sucede se o valor da expressão Expr1 for menor ou igual ao o valor da expressão Expr2
`+Expr1 =\= +Expr2`	sucede se o valor da expressão Expr1 for diferente do valor da expressão Expr2
`+Expr1 =:= +Expr2`	sucede se o valor da expressão Expr1 for igual ao valor da expressão Expr2
`−Número is +Expr`	sucede se Número unificar com o o valor da expressão Expr

Tabela 4.2: Predicados aritméticos do SWI-Prolog

As expressões aritméticas podem ser construidas a partir de funções aritméticas, números e variáveis. O SWI-Prolog disponibiliza as funções aritméticas usuais (soma, subtracção, etc.). Para uma lista completa das funções existentes, e respectiva notação, consultar a secção 4.27 do manual[7].

4.3 Exercícios

4.3.1 Unificação

Para cada uma das expressões seguintes diga qual é o resultado da unificação (verifique as suas resposta no Prolog):

1. `Y=5, X is 10+Y.`
2. `Y=X+5, Z is Y+1, X=5.`
3. `data(D,M,A)=data(5,D+1,M+1990).`
4. `data(2000,12,21,H)=data(A,M,D,hora(14+D,33,0)).`
5. `data(2000,12,21,hora(H,M,S))=data(A,M,D,hora(14,X,0)).`
6. `data(2000,12,21,hora(H,Min,S))=data(A,M,D,hora(14,X,0)).`

4.3.2 Expressões Aritméticas

Para cada uma das seguintes funções matemáticas, defina um predicado Prolog equivalente:

1. factorial:

$$fact(n) = \begin{cases} 1 & \Leftarrow n = 0 \\ fact(n-1) * n & \Leftarrow n > 0 \end{cases}$$

[7] Utilize a *querie*:
`?- help(4-27).`

2. multiplicação de números naturais por somas sucessivas:

$$mult(x,y) = \begin{cases} 0 & \Leftarrow y = 0 \\ x + mult(x, y-1) & \Leftarrow y > 0 \end{cases}$$

3. divisão inteira de números naturais por subtracções sucessivas:

$$div(x,y) = \begin{cases} 0 & \Leftarrow x < y \\ 1 + div(x-y, y) & \Leftarrow x \geq y \end{cases}$$

4. resto da divisão inteira de números naturais por subtracções sucessivas:

$$mod(x,y) = \begin{cases} x & \Leftarrow x < y \\ mod(x-y, y) & \Leftarrow x \geq y \end{cases}$$

5. função de Fibonacci:

$$F(N) = \begin{cases} F(N-1) + F(N-2) & \Leftarrow N > 2 \\ 1 & \Leftarrow N = 2 \\ 1 & \Leftarrow N = 1 \end{cases}$$

6. função de Ackerman:

$$A(M,N) = \begin{cases} N+1 & \Leftarrow M = 0 \\ A(M-1, 1) & \Leftarrow N = 0 \\ A(M-1, A(M, N-1)) & \text{noutro caso} \end{cases}$$

4.3.3 Planeamento arquitectónico

Escreva um programa Prolog que permita realizar o planeamento arquitectónico de um edifício com base nas seguintes restrições:

- o edifício deverá consistir em duas divisões rectangulares;
- as divisões deverão estar alinhadas com os pontos cardeais;
- cada divisão deverá possuir uma janela e uma porta interior;
- as divisões estão ligadas pela porta interior;
- uma das divisões também deverá possuir uma porta exterior;
- uma parede pode apenas conter uma janela ou uma porta;
- nenhuma janela deverá estar virada a norte;
- as janelas não podem estar situadas em faces opostas do edifício.

O programa deverá definir o predicado `plano/2` de tal forma que:

> `plano(sala(PFrente, PInt1, Jan1), sala(PInt2, Jan2)) :-` a primeira sala tem a porta da frente na parede PFrente[8], a porta interior na parede PInt1 e a janela na parede Jan1, e a segunda sala tem a porta interior na parede PInt2 e a janela na parede Jan2.

[8] Norte, Sul, Este ou Oeste.

Ficha Prática 5

5.1 Objectivos

1. Praticar a escrita de predicados sobre listas.

5.2 Conceitos

5.2.1 Listas

Notação

Em Prolog, as listas representam-se entre parênteses rectos ("`[`", "`]`"), com os elementos separados por vírgulas ("`,`"). Os elementos de uma lista podem ser qualquer tipo de termo.

Alguns exemplos de listas válidas:

```
[1, 2, 3]
[um, dois, tres]
[1, dois, 3]
[ana, [maria, madalena], X]
```

A **lista vazia** é representada por "`[]`". Uma **lista não vazia** pode ser representada utilizando a notação "`[H|T]`", em que "`H`" é a cabeça da lista e "`T`" é a cauda da lista. Assim, a lista "`[1, 2, 3]`" pode também ser representada das seguintes formas:

```
[1 | [2, 3]]
[1, 2 | [3]]
[1, 2, 3 | []]
```

Exemplos

Muitos dos predicados sobre listas que iremos definir serão **predicados recursivos**. Eles serão definidos para:

- o caso base (normalmente a lista vazia "`[]`");
- o caso recursivo (normalmente, para uma lista "`[H|T]`", definir alguma condição sobre a cabeça "`H`" e depois utilizar o predicado, recursivamente, na cauda "`T`").

Por exemplo, o predicado `tamanho/2`, que define o tamanho de uma lista, pode ser escrito da seguinte forma:

```
tamanho([], 0).
tamanho([_|T], N) :- tamanho(T, N1), N is N1+1.
```

Já o predicado `elemento/2` para testar a existência de um elemento numa lista pode ser definido da seguinte forma:

```
elemento(H, [H|_]).
elemento(E, [_|T]) :- elemento(E, T).
```

Note-se que aqui o caso base não está definido sobre a lista vazia: como a lista vazia não tem elementos, tal não faria sentido.

5.3 Exercícios

5.3.1 Percorrer listas

Escreva os seguintes predicados sobre listas (note que para alguns deles já existe equivalente em Prolog — no entanto, vale a pena o exercício):

1. `ultimo/2`:

   `ultimo(E,L) :-` `E` é o último elemento de `L`.

2. `somatorio/2`:

   `somatorio(L,N) :-` `N` é o somatório dos elementos presentes em `L`.

3. `adjacentes/3`:

   `adjacentes(E1,E2,L) :-` `E1` e `E2` são elementos adjacentes na lista `L`.

4. `selecciona/3`:

   `selecciona(E,L1,L2) :-` `E` é um elemento de `L1` e `L2` é o resto da lista.

5. `nesimo/3`:

   `nesimo(L, N, X) :-` `X` é o elemento que se encontra na posição `N` de `L`.

6. `tamanho/2`:

   o predicado tamanho definido na secção 5.2.1 está definido utilizando recursividade à esquerda, redefina-o por forma a utilizar recursividade à direita.

7. `media/2`:

   `media(L,N) :-` `N` é a média dos elementos da lista `L`.

8. `conta/3`:

   `conta(E,L,N) :-` `E` existe `N` vezes em `L`.

9. `maximo/2`:

   `maximo(L,N) :-` `N` é o maior elemento da lista `L`.

10. `posmax/2`:

    `posmax(L,N) :-` `N` é a posição do maior elemento da lista `L`.

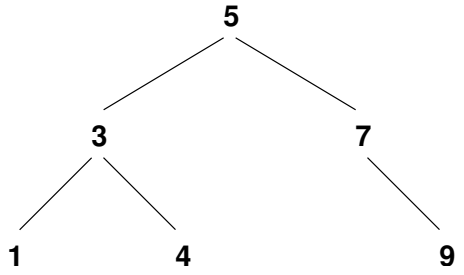

Figura 5.6: Árvore

5.3.2 Problemas com Listas

1. Considere a árvore binária apresentada na figura 5.6. Utilizando termos da forma "`arvore(sub-arv-esq, nodo, sub-arv-dir)`" para representar a árvore (utilize o átomo `nil` para indicar a inexistência de sub-árvores), escreva os seguintes predicados:

   (a) `procura/2`:

   `procura(Arv, Val) :-` `Val` existe na árvore `Arv`.

   (b) `to_list/2`:

   `to_list(Arv, L) :-` `L` é a representação sob forma de lista da árvore `Arv`.

2. Considere a figura 5.7. Escreva um predicado que lhe permita obter todos os possíveis

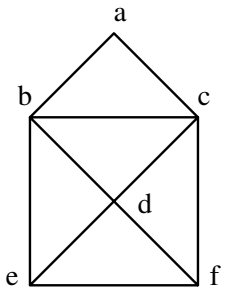

Figura 5.7: Casa

modos de desenhar a figura utilizando uma linha contínua.

Ficha Prática 6

1. Praticar a escrita de predicados para manipulação de listas.
2. Utilizar o `trace` para *debugging*.

6.1 Conceitos

6.1.1 Predicados sobre listas

A tabela 6.3 apresenta alguns dos predicados para manipulação de listas presentes no SWI-Prolog. Para mais informações, consultar as secções 4.29 a 4.31 do manual.

member(?Elem, ?List)	`Elem` existe em `List`
append(?List1, ?List2, ?List3)	`List3` é a concatnação de `List1` e `List2`
delete(+List1, ?Elem, ?List2)	`List2` é `List1` com todos os `Elem` removidos
select(?Elem, ?List, ?Rest)	`Elem` é um elemento de `List`, `Rest` são os restantes
nth0(?Index, ?List, ?Elem)	elemento na posição `Index` é `Elem` (contagem a partir de 0)
nth1(?Index, ?List, ?Elem)	elemento na posição `Index` é `Elem` (contagem a partir de 1)
last(?Elem, ?List)	`Elem` é o último elemento de `List`
reverse(+List1, -List2)	`List2` é a lista `List1` invertida
length(?List, ?Int)	`Int` é o número de elementos em `List`

Tabela 6.3: Alguns predicados sobre listas do SWI-Prolog

6.1.2 *Debug*

Numa visão procedimental da semântica de programas Prolog, cada predicado pode ser visto como uma caixa com quatro *portas* (ver figura 6.8):

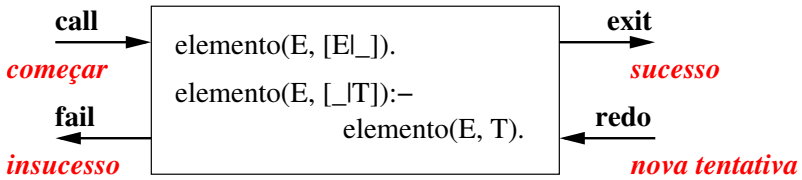

Figura 6.8: Predicados prolog como *caixas*

- **call** — corresponde a tentar aplicar o predicado;
- **exit** — corresponde a ter aplicado o predicado com sucesso;
- **redo** — corresponde a tentar encontrar uma nova solução;

- **fail** — corresponde à falha da última tentativa de aplicar o predicado.

Para fazer *debug* em Prolog utiliza-se o predicado `trace`. Quando o `trace` está activo, o interpretador pára sempre que uma das portas apresentadas acima é activada. De seguida apresenta-se um exemplo (texto em *itálico* corresponde a comentários):

```
?- trace.

Yes
% Foi activado o trace.
[trace]  ?- elemento(E,[a,b,c]).
% Call: vai ser utilizado o predicado elemento:
   Call: (6) elemento(_G286, [a, b, c]) ? creep
% creep corresponde a continuar (basta premir Enter).
   Exit: (6) elemento(a, [a, b, c]) ? creep
% Exit: sucesso! Utilizou a primeira regra (um facto) e a variável ficou unificada com o átomo a.

E = a ;
% Como respondemos com ";" (equivalente a forçar o Fail da solução apresentada) o Prolog vai procurar uma nova solução para o último Call terminado com sucesso. Para isso utiliza a porta Redo.
   Redo: (6) elemento(_G286, [a, b, c]) ? creep
% Utilizando a segunda regra, vai aplicar elemento à cauda da lista (o número entre parêntesis indica o aninhamento da invocação de predicados).
   Call: (7) elemento(_G286, [b, c]) ? creep
   Exit: (7) elemento(b, [b, c]) ? creep
% Sucesso!
   Exit: (6) elemento(b, [a, b, c]) ? creep
% Como conseguimos resolver o corpo da regra, resolvemos a cabeça.

E = b ;
   Redo: (7) elemento(_G286, [b, c]) ? creep
% Tenta-se sempre o Redo do último call. É por isso que aqui saltou para o nível sete...
   Call: (8) elemento(_G286, [c]) ? creep
   Exit: (8) elemento(c, [c]) ? creep
   Exit: (7) elemento(c, [b, c]) ? creep
   Exit: (6) elemento(c, [a, b, c]) ? creep

E = c ;
   Redo: (8) elemento(_G286, [c]) ? creep
% ... e aqui para o nível oito!
   Call: (9) elemento(_G286, []) ? creep
   Fail: (9) elemento(_G286, []) ? creep
% Para a lista vazia o predicado falha! (como não há mais regras que se possam aplicar, vai falhar a querie.)
   Fail: (8) elemento(_G286, [c]) ? creep
   Fail: (7) elemento(_G286, [b, c]) ? creep
   Fail: (6) elemento(_G286, [a, b, c]) ? creep

No
[debug]  ?-
```

6.2 Exercícios

6.2.1 Manipular listas

Escreva os seguintes predicados sobre listas (tal como na Ficha 5, para alguns deles já existe equivalente em Prolog — continua a valer a pena o exercício)::

1. `conc/3`:

   `conc(L1,L2,L)` :- `L` é a concatenação de `L1` e `L2`.

2. `inverte/2`:

   `inverte(L1,L2)` :- a lista `L2` é a lista `L1` invertida.

3. `interseccao/3`:

   `interseccao(L1,L2,L)` :- `L` é a intersecção de `L1` e `L2`.
   Considere que as listas `L1` e `L2` não possuem elementos repetidos e estão ordenadas por ordem crescente.

4. `filtrapar/2`:

   `filtrapar(L1, L2)` :- a lista `L2` tem os elementos em posição par na lista `L1`.

5. `apaga_primeiro/3`:

   `apaga_primeiro(L1,X,L2)` :- `L2` é o resultado de retirar a `L1` a primeira ocorrência de `X`.

6. `apaga/3`:

   `apaga(E,L1,L2)` :- `L2` é a lista `L1` com os elementos iguais a `E` removidos.

6.2.2 Problemas de *Debugging*

1. Considere a definição de elemento apresentada na figura 6.8. Com esta definição o predicado repete soluções. Faça um *dry run* da query

   ```
   ?- elemento(X, [a,b,a,c]).
   ```

   para descobrir a causa do problema. Confirme o seu *dry run* utilizando o `trace` no Prolog.

2. Considere a seguinte definição:

   ```
   conta(E, L, N) :- conta(E, L, 0, N).
   conta(_, [], Ac, Ac).
   conta(E, [E|T], Ac, N) :- Ac1 is Ac+1, conta(E, T, Ac1, N).
   conta(E, [_|T], Ac, N) :- conta(E, T, Ac, N).
   ```

   Este predicado não está correctamente definido:

   ```
   ?- conta(a, [a,b,a,c], X).
   X = 2;
   X = 1;
   X = 1;
   X = 0;
   No
   ```

   Faça um *dry run* da query para descobrir a causa do problema. Confirme o seu *dry run* utilizando o `trace` no Prolog.

6.2.3 Problemas com Listas

1. Relembre o problema 3.3.2 (página 13), escreva os seguintes predicados:

   (a) `ha_ligacao/3`:

   > `ha_ligacao(A,B,C) :-` existe ligação entre `A` e `B` percorrendo o caminho `C`.

   Considere que os caminhos deverão ser representados por listas de triplos `O-T-D`, em que cada triplo indica que se vai de `O` até `D` utilizando o transporte `T`.

   (b) `ha_ligacao/4`:

   > `ha_ligacao(A,B,C,T) :-` existe ligação entre `A` e `B` percorrendo o caminho `C` e utilizando apenas o meio de transporte `T`.

   Neste caso os caminhos deverão ser representados por listas contendo os nomes das cidades por onde se passa.

2. Relembre o exercício 5 da secção 4.3.2 (página 16).

   Escreva o predicado `serieFibonacci/2`:

   > `serieFibonacci(N, L) :-` `L` é uma lista contendo a série de Fibonacci até ao `N`-ésimo elemento.

Ficha Prática 7

7.1 Objectivos

1. Praticar a escrita de Árvores de Prova.

7.2 Conceitos

7.2.1 Árvores de Prova

Relembre a secção 6.1.2. Nela é apresentado o *trace* de uma *query*. Uma forma mais útil de representar a execução de um programa é utilizando Árvores de Prova. Na figure 7.9 apresenta-se a árvore de prova correspondente ao exemplo apresentado na secção 6.1.2.

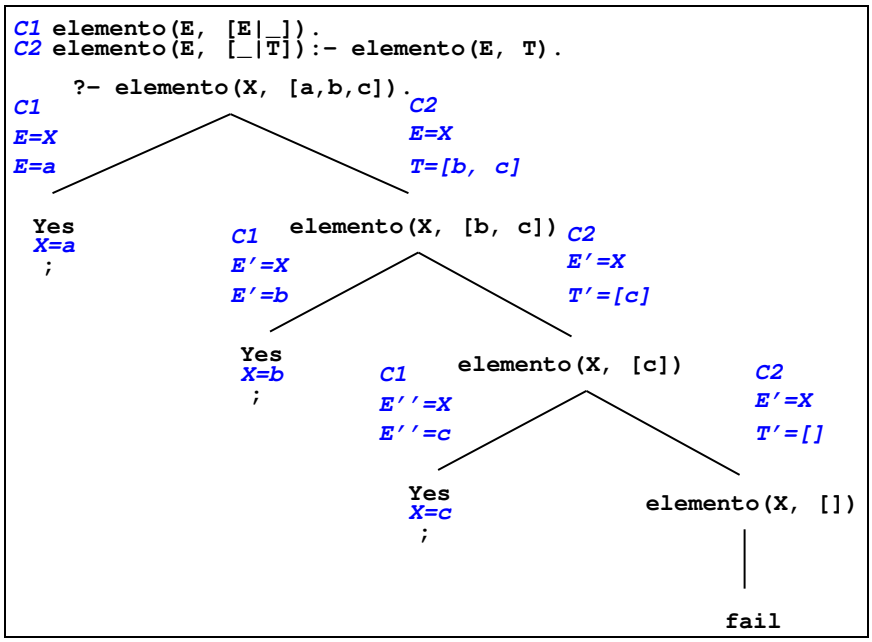

Figura 7.9: Uma árvore de Prova

Na página da disciplina será colocado um exemplo que apresenta, passo a passo, a construção de uma árvore de prova.

7.3 Exercícios

7.3.1 Árvores de Prova simples

Considere os exercícios anteriormente resolvidos na fichas 1 a 3. Escreva as Árvores de Prova para as seguintes *queries* (utilize o `trace` para validar a construção das árvores):

1. `aluno(rui,X).`
2. `aluno(X,ppiii).`
3. `fazppiii(X).`
4. `contacto(rui,Tel).`
5. `a_dar_festa(ze).` — utilizando a seguinte regra para `a_dar_festa`:

   ```
   a_dar_festa(P) :-
       emCasa(P), emcasade(X,P), emcasade(Y,P),
       X\==P, Y\==P, X\==Y.
   ```

   Como poderia tornar a regra mais eficiente? (não necessita construir a árvore toda para responder à pergunta)

7.3.2 Árvores de Prova para predicados recursivos

1. Considerando as Bases de Conhecimento apresentadas na ficha 1, determine as árvores de prova para as seguintes *queries*:

   (a) `descendente(manuel,X).`
   utilizando a seguinte definição para `descendente/2`:

   ```
   descendente(X,Y):- pai(Y,X).
   descendente(X,Y):- descendente(Z,Y),pai(Z,X).
   ```

   (b) `descendente(mário,X).`
   utilizando agora a seguinte definição para `descendente/2`:

   ```
   descendente(X,Y):- pai(Z,X),descendente(Z,Y).
   descendente(X,Y):- pai(Y,X).
   ```

   Qual a diferença para a árvore anterior?

2. Faça árvores de prova para os dois problemas da secção 6.1.2:

   (a) `elemento(X, [a,b,a,c]).`
   (b) `conta(a, [a,b,a,c], X).`

7.3.3 Árvores de Prova para predicados com construção de listas

Considere agora os exercícios da secção 6.2.3. Escreva as Árvores de Prova para as seguintes *queries* (considere a base de conhecimento desenvolvida para representar a figura 3.5 — página 13):

1. `ha_ligacao(lisboa,braga,X).`
2. `ha_ligacao(braga,lisboa,X).`
3. `ha_ligacao(porto,D,X,_).`

Ficha Prática 8

8.1 Objectivos

1. Utilizar os meta-predicados `bagof`, `setof` e `findall`.

8.2 Conceitos

O Prolog possui três meta-predicados dedicados a encontrar todas as soluções para uma dada *query*:

- `findall`
- `bagof`
- `setof`

Considere a seguinte Base de Conhecimento:

```
% pais(Filho, Pai, Mae).
filho(manuel, mario, maria).
filho(teresa, mario, maria).
filho(joao, manuel, joana).
filho(lurdes, joao, joana).
filho(rui, manuel, teresa).
filho(alberto,manuel, teresa).
```

8.2.1 `findall`

A *query*:

```
?- findall(+T, +Objectivo, -L).
```

unifica `L` com a lista de todos os `T` que se obtêm ao procurar todas as soluções de `Objectivo` (`T` deverá conter variáveis que apareçam na expressão `Objectivo`).

Se não existirem instanciações para `T`, `L` unificará com lista vazia.

Considere que se pretende saber quem são os filhos do Manuel e da Teresa. Isso pode ser conseguido com as seguinte *query*:

```
?- findall(Filho,filho(Filho,manuel,teresa),L).

F = _G157
L = [rui, alberto]
```

Neste caso, o `findall` criou a lista `L` com todas as instanciações de `Filho` que satisfazem `filho(Filho,manuel,teresa)`.

Se existirem variáveis no objectivo que não estejam no padrão a lista conterá as instanciações do padrão para *qualquer valor dessas variáveis*. Por exemplo, a *query*

```
?- findall(F,filho(F,manuel, M),L).

F = _G157
M = _G159
L = [joao, rui, alberto]
```

coloca em `L` todas as instanciações de `F` para as quais também exista um `M` tal que se verifique `filho(F,manuel, M)`. Ou seja. calcula a lista de todos os filho de Manuel (independentemente de quem seja a mãe — note como a variável `M` não ficou instânciada).

8.2.2 `bagof`

A *query*:

```
?- bagof(+T, +Objectivo, -L).
```

unifica `L` com a lista de todos os `T` que se obtêm ao procurar todas as soluções de `Objectivo` (`T` deverá conter variáveis que apareçam na expressão `Objectivo`).

Se não existirem instanciações para `T`, o `bagof` falha.

Se existirem variáveis no objectivo que não estejam no padrão o `bagof` calculará uma lista para cada uma das instanciações possíveis para essas variáveis.

Por exemplo, a *query*

```
?- bagof(F,filho(F,manuel, M),L).

F = _G411
M = joana
L = [joao] ;

F = _G411
M = teresa
L = [rui, alberto] ;

No
```

calcula uma lista para `M=joana` (todos os filhos de Manuel e de Joana) e outra para `M=teresa` (todos os filho de Manuel e Teresa). Note que neste caso a variável M fica sempre unificada.

Considere agora a query:

```
?- bagof(F,filho(F,Pai, Mae),L).

F = _G157
Pai = mario
Mae = maria
L = [manuel, teresa] ;

F = _G157
Pai = manuel
Mae = joana
L = [joao] ;

F = _G157
Pai = joao
Mae = joana
L = [teresa] ;
```

```
F = _G157
Pai = manuel
Mae = teresa
L = [rui, alberto] ;

No
```

Neste caso obtemos listas de filhos para cada par de pais/mães.

Se na query anterior não pretendessemos que o Prolog considerasse as mães (se pretendessemos a lista de filhos de cada pai, independentemente das mães) utilizariamos a quantificação existêncial da variável `Mae`:

```
?- bagof(F,Maeˆfilho(F,Pai, Mae),L).

F = _G157
Mae = _G159
Pai = mario
L = [manuel, teresa] ;

F = _G157
Mae = _G159
Pai = joao
L = [teresa] ;

F = _G157
Mae = _G159
Pai = manuel
L = [joao, rui, alberto] ;

No
```

Neste caso as listas são calculadas com bases nas unificações de `F`, para qualquer valor de `Mae`.

Note que a expressao `findall(F, filho(F,Pai,Mae), L)` é equivalente à expressão `bagof(F, PaiˆMaeˆfilho(F,Pai, Mae), L)` excepto quando não exitam resultados a colocar em `L`. Nesse caso, a primeira expressão termina com `L` unificado com a lista vazia, e a segunda expressão falha.

8.2.3 `setof`

Semelhante a bagof, mas a lista é ordenada e sem duplicados:

```
setof(T, G, L) :-
        bagof(T, G, Laux),
        sort(Laux, L).
```

8.3 Exercícios

1. Relembre a Secção *Factos, queries e regras* da Ficha Prática 1:
   (a) Escreva o predicado `alunos_de_ppiii/1` que define a lista de alunos inscritos a `ppiii`.
   (b) Escreva agora o predicado `alunos_de/2` que define a lista de alunos de uma cadeira[9].

[9] `alunos(C,L)` se `L` é a lista de alunos inscritos à cadeira `C`.

(c) Escreva ainda o predicado `cadeirão/1` que define qual a cadeira com maior número de alunos inscritos.

(d) Escreva os predicados `cadeiras_do_rui/1` (a que cadeiras está o `rui` inscrito), `cadeiras_de/2` (a que cadeiras está um aluno inscrito) e `atarefado/1` (quem é que está inscrito a maior número de cadeiras).

2. Considere agora uma Base de Conhecimento onde são armazenados factos `consultou/2` com informação sobre as páginas Web que cada utilizador de um dado ISP consultou. Tome como exemplo a seguinte Base de Conhecimento:

```
consultou(util1, p1).
consultou(util1, p2).
consultou(util1, p3).
consultou(util1, p4).
consultou(util2, p1).
consultou(util2, p1).
consultou(util2, p1).
consultou(util3, p2).
consultou(util4, p2).
```

(a) Escreva o predicado `mais_consultada/1` que define qual a página mais consultada[10].

(b) Escreva o predicado `melhor_cliente/1` que define qual o utilizador que fez maior número de consultas.

(c) Escreva o predicado `com_mais_clintes/1` que define a página com maior número de utilizadores diferentes.

(d) Escreva o predicado `util_por_pagina/1` que define uma lista de pares página/lista de utilizadores que consultaram a página.

[10] Lembre-se que o `setof` cria uma lista ordenada.

Ficha Prática 9

9.1 Objectivos

1. Exercícios com *cut* e *fail*.
2. Exercícios com Árvores de Prova (continuação).

9.2 Conceitos

9.2.1 cut — !

O *cut* (!) é um predicado que impede o *backtracking*. Dito de outro modo, o cut permite "cortar" ramos de uma árvore de prova. Assim, pode ser utilizado para melhorar a eficiência de um programa Prolog, cortando ramos que à partida se sabe que irão falhar.

Considere o seguinte predicado:

```
maior(X,Y,X) :- X>=Y.
maior(X,Y,Y) :- X<Y.
```

A árvore de prova para a *query* `?- maior(4,2,R).` é a apresentada na figura 9.10. Como

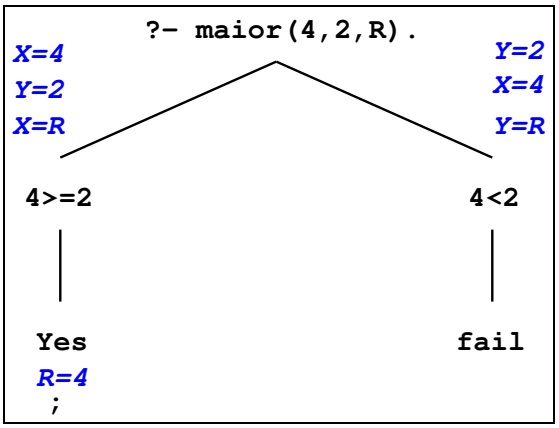

Figura 9.10: Árvore de prova para maior

se pode verificar pela árvore apresentada, o interpretador tenta aplicar a segunda regra durante o processo de *backtracking* para procurar soluções alternativas à solução inicial (`R=4`). Como seria de esperar a segunda regra falha. Na verdade as duas regras são mutuamente exclusivas.

Numa situação de regras mutuamente exclusivas, é possível utilizar o *cut* para indicar que, quando uma das regras é aplicável, não deverão ser tentadas as restantes. Neste caso o predicado ficaria:

```
maior2(X,Y,X) :- X>=Y, !.
maior2(X,Y,Y) :- X<Y.
```

e a árvore de prova passaria a ser a paresentada na figura 9.11. Neste caso o ramo direito

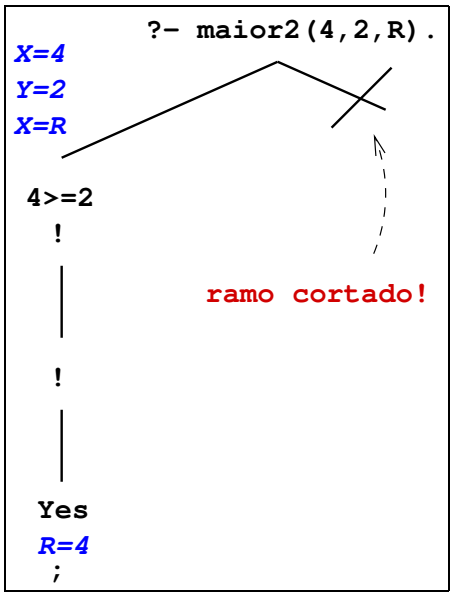

Figura 9.11: Árvore de prova para maior2

da árvore é cortado pois, quando o *backtracking* é iniciado e se tenta fazer o *redo* do *cut*, o objectivo falha sem se tentarem outras soluções.

green cut

É importante notar que, no caso apresentado, o *cut* apenas corta ramos da árvore que vão falhar. Se o *cut* fosse removido o predicado produziria os mesmos resultados mas calcularia árvores de prova maiores. A este tipo de *cut*, que não altera a semântica do predicado, chama-se um *green cut*.

red cut

Considere-se agora a seguinte versão do predicado:

```
maior3(X,Y,X) :- X>=Y, !.
maior3(_,Y,Y).
```

Nesta nova versão, procura aproveitar-se o facto de o *cut* da primeira regra impedir o *backtracking* para a segunda regra para simplificarmos esta última. No entanto, se agora removermos o *cut* da primeira regra o significado do predicado é afectado. A este tipo de *cut*, que altera a semântica do predicado, chama-se *red cut*.

Deve evitar-se a utilização de *red cuts* pois, para além de tornarem a compreensão do código mais difícil, podem também dar origem a resultados inesperados quando mal aplicados (e aplicar bem um *red cut* é mais difícil que aplicar bem um *green cut*!). Tente, por exemplo, a seguinte *query*:

```
?- maior3(4,2,2).
```

O que se passou?!

9.2.2 fail — negação por falha

O *fail* é um predicado que falha sempre. Utilizando o *cut* e o *fail*, é possível escrever o predicado negação:

```
negacao(A) :- A, !, fail.
negacao(_).
```

Note-se que este predicado não produz unificações. Porquê?!

9.3 Exercícios

9.3.1 Cut e Árvores de Prova

1. Para a definição:

   ```
   p(1).
   p(2):- !.
   p(3).
   ```

   Diga qual o resultado das seguintes queries:

   - `p(X).`
   - `p(X),p(Y).`
   - `p(X),!,p(Y).`

   Para cada uma das queries faça a respectiva árvore de prova.

2. Para o progama Prolog que se apresenta:

   ```
   q(a).
   q(b).
   q(c).
   r(b,b1).
   r(c,c1).
   r(a,a1).
   r(a,a2).
   p(X,Y):- q(X),r(X,Y).
   p(d,d1).
   p1(X,Y):- q(X),r(X,Y),!.
   p1(d,d1).
   p2(X,Y):- q(X),!,r(X,Y).
   p2(d,d1).
   p3(X,Y):- !,q(X),r(X,Y).
   p3(d,d1).
   ```

   efectue as árvores de prova para as queries seguintes:

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

9.3.2 Exercícios com cuts

Para cada um dos seguintes enunciados, escreva predicados recorrendo a *green cuts* sempre que possível:

1. Crie um predicado `classe/2`, que determine se um determinado número é `positivo`, `negativo` ou `zero`.
2. Utilize o predicado definido anteriormente por forma a criar um outro, denominado `split/3`, que dada uma lista de números inteiros, origine duas outras listas, uma com os números positivos e zero e uma outra com os números negativos.

   Faça a árvore de prova para a *query*:
   ```
   ?- split([1,-1,3,0,-5,-2], LPositivosZero, LNegativos).
   ```
3. Crie um predicado `ifthenelse/3`, que implemente a estrutura condicional if...then...else.
4. Recorrendo ao uso de *cuts*, formule o predicado `interseccao/3` que implementa a intersecção de listas (vistas como conjuntos).
5. Crie um predicado que dada uma lista de inteiros, devolva a lista dos inteiros que ocorrem mais do que duas vezes nessa lista.
6. Crie um predicado `diferentes/2`, que dê verdadeiro se os dois parâmetros do predicados forem de facto diferentes.
7. Crie um predicado `diferenca/3`, que efectue a diferença de listas, isto é, cria uma lista com os elementos da primeira lista que não se encontram na segunda. Utilize o predicado `negação` definido na secção 9.2.2.
8. Considere o seguinte predicado:

   ```
   % norep(L1,L2) :- L2 é a lista L1 sem elementos repetidos
   norep([],[]).
   norep([H|T1],L) :- member(H,T1), norep(T1,L).
   norep([H|T1],[H|T2]) :- \+ member(H,T1), norep(T1,T2).
   ```

   O predicado funciona mas tem um problema:

   ```
   ?- norep([a,d,a,d,a],L).
   L = [d, a] ;
   L = [d, a] ;
   No
   ```

   O predicado calcula o resultado certo, mas mais que uma vez!

   Utilizando o `trace` e/ou a árvore de prova existente na página de PPIII, identifique onde está o problema e procure resolvê-lo utilizando *cuts*.
9. Escreva o predicado `del/3`:

   ```
   % del(E,L1,L2) :- L2 é a lista L1 com todas as ocorrências
   %                 de E removidas
   ```
10. Escreva o predicado `quicksort/2`:

    ```
    % quicksort(L1,L2) :- L2 é uma versão ordenada de L1
    %                     (utilizando o algoritmo quicksort)
    ```

Ficha Prática 10

10.1 Objectivos

1. Praticar a utilização de `assert` e `retract`.

10.2 Conceitos

O Prolog permite a manipulação da base de conhecimento durante a execução de um programa. Para tal disponibiliza dois meta-predicados base: `assert` e `retract`.

Outra utilização que pode ser dada a estes meta-predicados é na simulação de variáveis globais. Em Prolog existem apenas variáveis locais (ao termo em que são utilizadas). Em situações muito específicas poderá ser útil recorrer a variáveis globais. Os meta-predicados `assert` e `retract` podem ser utilizados para simular estas variáveis.

10.2.1 `dynamic/1`

Antes de perceber os meta-predicados `assert` e `retract`, convém entender o conceito de predicados estáticos e predicados dinâmicos.

Por omissão, um predicado que seja carregado através do `consult` fica definido como estático. Ou seja, a sua definição não poderá ser alterada, não sendo possível adicionar ou remover factos e/ou regras a ela associados[11]. Para indicar que a definição de um dado predicado poderá ser alterada (através da adição/remoção de factos e/ou regras), o predicado deverá ser declarado como dinâmico. Isso é conseguido através da directiva `dynamic`.

Por exemplo, a inclusão da directiva:

```
:- dynamic primo/1.
```

no início de um ficheiro (em que o predicado `primo/1` é definido) permite que, após o *consult* do ficheiro, a definição de `primo/1` seja alterada.

10.2.2 `assert/1` (`asserta/1` e `assertz/1`)

O predicado `assert/1` permite adicionar factos ou regras à base de conhecimento. O facto/regra é adicionado como o último facto/regra do predicado em causa. O predicado, caso exista, tem que estar definido como dinâmico.

Por exemplo (e assumindo que os predicados `primo/1` e `impar/1` são dinâmicos ou não existem na base de conhecimento), após

```
assert(primo(5))
```

o facto "`primo(5).`" passará a constar na base de conhecimento (como o último facto referente ao predicado `primo/1`). Após

[11] Existe, no entanto, uma excepção a esta regra. Ver mais à frente.

```
assert(impar(N):-primo(N))
```

a regra "`impar(N):-primo(N).`" passará a constar na base de conhecimento (como a última regra referente ao predicado impar).

Predicados adicionados à base de conhecimento utilizando `assert/1` ficam definidos como dinâmicos.

`asserta/1`

Semelhante a `assert/1`, mas o facto/regra é adicionado como o primeiro facto/regra do predicado.

`assertz/1`

Equivalente a `assert/1`.

10.2.3 `retract/1` (`retractall/1` e `abolish/1`)

O predicado `retract/1` permite remover factos ou regras da base de conhecimento. O predicado a que diz respeito o facto/regra a ser removido tem que estar definido como dinâmico. Como parâmetro deverá ser passado um termo que unifique com o facto ou regra a remover.

Considere

```
retract(primo(X))
```

o primeiro facto que unifique com "`primo(X)`" será removido da base de conhecimento.

`retractall/1`

Semelhante a `retract/1`, mas **todos** os factos ou regras cuja **cabeça** unifique com o termo passado como parâmetro são removidos da base de conhecimento. Note que neste caso, para remover regras, não é preciso indicar toda a regra, mas apenas a sua cabeça.

`abolish/1`

A expressão `abolish(impar/1)` remove da base de conhecimento todos os factos e regras com functor `impar` e aridade 1. Os atributos definidos para o predicado (por exemplo, se é dinâmico) são também removidos.

Atenção: no SWI-Prolog este meta-predicado remove factos e regras mesmo de predicados estáticos.

10.3 Exercícios

1. Com base no que sabe sobre os predicados `assert`, `retract`, `retractall` e `abolish` procure prever o resultado das queries sublinhadas:

   (a)
   ```
   ?- [user].
   impar(1). impar(2). impar(3).
   ^D
   ?- assert(par(2)).
   ?- assert(par(3)).
   ?- assert(par(4)).
   ?- listing(impar).
   ?- listing(par).
   ?- assert(impar(5)).
   ```

```
?- retract(par(3)).
?- retract(impar(2)).
```

(b)

```
?-  [user].
:- dynamic m2/1, m3/1.
m2(2). m2(4). m2(6).
m3(3). m3(6). m3(8).
^D
?- retract(m3(8)).
?- assert(m2(8)).
?- assert(m3(9)).
```

(c)

```
?-  [user].
:- dynamic m23/1.
m23(X) :- m2(X).
m23(X) :- m3(X).
nãoComum(X) :- m2(X), not(m3(X)).
nãoComum(X) :- m3(X), not(m2(X)).
^D
?- assert(comum(X):- m2(X), m3(X)).
?- retract(comum(_)).
?- retract(comum(_):- m2(X), m3(X)).
?- retractall(m23(_)).
?- retractall(nãoComum(_)).
?- abolish(nãoComum(_)).
```

2. Escreva o predicado `regista_pares/1` que, dada uma lista de número inteiros, cria na base de conhecimento o facto `pares/1` cujo parâmetro é uma lista com os números pares presentes na lista original.

3. Escreva o predicado `regista_repetidos/1` que, dada uma lista, cria na base de conhecimento os factos `repetidos/1` e `repeticao/2`. O parâmetro de `repetidos/1` é uma lista com os elementos repetidos da lista original (a lista gerada não deverá ter repetições). Os factos do predicado `repeticao/2` registam, para cada um dos elementos repetidos, quantas vezes ele aparece repetido.

4. Relembre a base de conhecimento da secção 1.3. Escreva o predicado `gera_tios/0` que cria factos `tio/2`, que definem a relação "tio de", com base na informação presente na base de conhecimento.

5. Relembre o predicado `aluno/2` definido na secção 1.4.2. Escreva o predicado `gera_turmas/0` que cria, na base de conhecimento, factos `alunos/2` associando, a cada disciplina, a lista de alunos nela inscritos.

6. Relembre os exercícios da secção 4.3.2. Refaça os predicados para as funções de Fibonacci e Ackerman por forma a que reutilizem valores já calculados. Compare as capacidades de cálculo das versões originais com as das versões agora desenvolvidas.

Ficha Prática 11

11.1 Objectivos

1. Leitura e escrita de informação em ficheiros: `see` e `tell`.
2. Praticar a utilização de `assert` e `retract`.

11.2 Conceitos

11.2.1 `read` e `write`

O SWI Prolog fornece diversos predicados para leitura e escrita. Listam-se de seguida alguns deles:

- `read(-Term)` — lê o próximo termo da *stream* de entrada (*input*) e unifica-o com `Term`. Se a *stream* de entrada actual já não possuir mais informação, `Term` é unificado com o átomo `end_of_file`.
- `write(+Term)` — escreve o termo `Term` na *stream* de saída (*output*) actual.
- `nl` — escreve o caracter de mudança de linha na *stream* de saída.
- `writeln(+Term)` — escreve o termo `Term` seguido de uma mudança de linha (equivalente a: `write(+Term), nl`).
- `writeq(+Term)` — escreve o termo `Term`, sempre que necessário são colocadas plicas nos átomos (um termo escrito com `writeq/1` pode depois ser lido com o `read/1`).

Analise os seguintes exemplos:

```
?- read(N), write('Olá '), write(N).
|: 'Manuel Albertino'.
Olá Manuel Albertino

N = 'Manuel Albertino'

Yes
?- read(N), write('Olá '), writeq(N).
|: 'Manuel Albertino'.
Olá 'Manuel Albertino'

N = 'Manuel Albertino'

Yes
?- read(N), write('Olá '), writeq(N).
|: jose.
```

```
Olá jose

N = jose

Yes
?- read(N), writeln('Olá '), write(N).
|: 'filomena'.
Olá
filomena

N = filomena

Yes
?- read(N), writeln('Olá '), writeq(N).
|: 'filomena'.
Olá
filomena

N = filomena

Yes
```

11.2.2 `tell` (`told`, `telling`)

- `tell(F)` — abre `F` para escrita e torna-o a *stream* de saída actual (se `F` já for uma *stream*, limita-se a torná-la a *stream* de saída actual).
- `told` — fecha a *stream* de saída actual.
- `telling(S)` — `S` é unificado com a *stream* de saída actual.

Para escrever no ficheiro `fich.txt` o facto `impar(1)` e, garantidamente, voltar a deixar o interpretador com a *stream* de saída como estava anteriormente, utiliza-se o código:

```
telling(S),
tell('fich.txt'),
write(impar(1)),
writeln('.').
told,
tell(S).
```

Escreveu-se um ponto (.) no fim da expressão para que ela corresponda a um termo Prolog bem formado. Deste modo, mais tarde será possível ler o termo para o voltar a colocar na base de conhecimento.

11.2.3 `see` (`seen`, `seeing`)

- `see(F)` — abre `F` para leitura e torna-o a *stream* de entrada actual (se `F` já for uma *stream*, limita-se a torná-la a *stream* de entrada actual).
- `seen` — fecha a *stream* de entrada actual.
- `seeing(S)` — `S` é unificado com a *stream* de entrada actual.

Para ler do ficheiro anterior o facto lá escrito e voltar a colocá-lo na base de conhecimento, deixando, garantidamente, o interpretador com a *stream* de entrada como estava anteriormente, utiliza-se o código:

```
seeing(S),
see('fich.txt'),
read(T),
assert(T),
seen,
see(S).
```

11.2.4 `repeat`

O predicado `repeat` sucede sempre, sendo utilizado para controlar o *backtracking*. Um exemplo de utilização é apresentado na próxima secção.

11.2.5 Leitura de informação a partir de ficheiros

A leitura de informação a partir de um ficheiro, e o seu carregamento para a base de conhecimento do Prolog, obedece a um esquema geral que se baseia na repetição da leitura até que se encontre o fim do ficheiro (representado pelo símbolo `end_of_file`). Em simultâneo com a leitura, a informação lida é processada (tarefa que é inerente à complexidade do problema) e gravada em memória (através do predicado `assert`).

O código padrão para este comportamento é apresentado de seguida:

```
carrega(F) :- see(F),
              repeat,
                read(T),
                processaTermoLido(T),
                T==end_of_file,
              !,
              seen.


processaTermoLido(end_of_file).
processaTermoLido(P) :- assert(P).  % P é o resultado da leitura.
                                    % Atenção que pode ser necessário
                                    % fazer algumas operações sobre P,
                                    % dependendo da lógica do problema.
```

11.3 Exercícios

1. Com base no que sabe sobre os predicados `see` e `tell` procure prever o resultado das seguintes queries:

   (a) `?- write('Escreva o seu nome:'), read(N), atom_concat('Olá ', N, S), write(S).`

   (b) `?- write('Escreva o seu nome:'), read(N), tell('Teste.txt'), atom_concat('Olá ', N, S), write(S), told.`
   Veja o conteúdo do ficheiro `Teste.txt`.

(c) Crie o ficheiro `Entrada.txt` com uma linha contendo o seu nome sob a forma de átomo prolog (por exemplo, "`'Manuel Costa'.`"):

```
?- see('Entrada.txt'), read(N), atom_concat('Olá ', N, S),
write(S), seen.
```

(d) `?- tell(t1), write(olá), tell(t2), write(olé), told, write(oli), told, write(olo).`

(e) `?- tell(t1), write(olá), telling(F), tell(t2), write(olé), tell(F), write(oli), told, write(olo).`

2. Num ficheiro encontram-se por ordem aleatória predicados que representam definições de figuras geométricas sob a forma: `quadrado(p, tam)`, `triangulo(p1, p2, p3)`, `rectangulo(p1, p2)` e `circunf(p, r)`:

   (a) Escreva o predicado `lerFiguras/1` que carrega para a Base de Conhecimento todas as figuras presentes no ficheiro.

   (b) Escreva o predicado `lerQuadrados/1` que carrega para a Base de Conhecimento todos os quadrados presentes no ficheiro.

   (c) Escreva o predicado `maiorCircunf/2` que define qual a maior circunferência presente no ficheiro (sem alterar a base de conhecimento).

   (d) Escreva ainda o predicado `listaQuad/2` que define a lista dos quadrados presentes na Base de Conhecimento que possuem como lado um valor dado.

   (e) Escreva o predicado `guardarQuad/1` que permite guardar, num ficheiro cujo nome é dado, todos os quadrados existentes na base de conhecimento.

   (f) Escreva o predicado `guardarTri/2` que permite guardar, num ficheiro cujo nome é dado, todos os triângulos existentes na base de conhecimento que possuam como um dos vértices o ponto `P` passado como parâmetro.

3. Termos da forma `telefonou(pessoa1, pessoa2)` são utilizados numa Base de Conhecimento para indicar que uma dada pessoa contactou outra. Escreva predicados que lhe permitam:

   (a) Carregar para a BC tais factos a partir de um ficheiro F.

   (b) Determinar qual a pessoa que mais vezes ligou para a pessoa X.

   (c) Determinar qual o conjunto de pessoas que a pessoa Y contactou.

   (d) Determinar a lista de pares `(pessoa, lista_de_pessoas_contactadas)`.

   (e) Sabendo que cada chamada custa P, gravar num ficheiro, para cada pessoa, o custo das suas chamadas.

Ficha Prática 12

12.1 Objectivos

1. Criação de programas interactivos.

12.2 Conceitos

12.2.1 Interactividade em programas Prolog

A forma de criação de um predicado que implemente a interacção com o utilizador, segue um padrão reutilizável, e recorre ao predicado `repeat` já anteriormente utilizado.

```
menu :-
       carrega('ficheiros.dat'),
       repeat,
         write('*** MENU ***'),nl,
         write('1 - Opção 1'),nl,
         write('2 - Opção 2'),nl,
         ...
         write('6 - Opção 6'),nl,
         write('7 - Sair'),nl,
         write('Opção:'),nl,
         read(X),
         processa(X),
       X=7.                % se falhar volta ao repeat


processa(1) :-
  .....,
  !.

...
...
...

processa(7) :- !.
processa(_) :- write('Opção Inválida! Tente Novamente!').
```

Se quisermos efectuar validações ao tipo de entrada que queremos obter, pode ser utilizado o predicado `repeat` por forma a forçar leitura de valores que estejam dentro de uma determinada gama.

Para criar um predicado que valide a entrada de números inteiros, podemos escrever o código que se apresenta:

```
lerInteiro(I) :-
   repeat,
     read(I),
   integer(I),
   !.
```

12.3 Exercícios

1. Relembre o predicado `aluno/2` apresentado na secção 1.4.2. Escreva um programa que permita, de forma interactiva, realizar as seguintes operações:
   - inscrever um aluno a uma disciplina;
   - remover a inscrição de um aluno a uma disciplina;
   - saber se um aluno está inscrito a uma disciplina;
   - saber a que disciplinas está um aluno inscrito;
   - saber quais os alunos inscritos a uma disciplina;
   - saber qual a disciplina com mais alunos inscritos;
   - saber qual o aluno inscrito a mais disciplinas.
2. Considere que tem, numa base de conhecimento, predicados `bib/2` com informação relativa a livros e respectivos autores:

   ```
   bib(asterix,[uderzo, gosciny]).
   bib(relatorio_minoritario,[philip_k_dick]).
   bib(maias,[eca_queiroz]).
   bib(farpas,[eca_queiroz,ramalho_ortigao]).
   bib(ubik,[philip_k_dick]).
   ```

   (a) escreva o predicado `autores/1` que constrói uma lista que permita mais facilmente responder à questão "Que livros escreveu determinado autor?". Para a base de conhecimento anterior a lista resultante deverá ter o seguinte formato:

   ```
   [auth(eca_queiroz,[maias,farpas]),
    auth(gosciny,[asterix]),
    auth(philip_k_dick,[ubik,relatorio_minoritario]),
    auth(ramalho_ortigao,[farpas]),
    auth(uderzo,[asterix])
   ]
   ```

   (b) desenvolva uma interface que permita ao utilizador inserir, consultar e remover livros, bem como consultar a lista de livros por autor.
3. Considere agora que tem a informação relativa aos livros numa lista. Para a base de conhecimento anterior essa lista assume a forma que a seguir se exemplifica:

   ```
   [bib(asterix,[uderzo, gosciny]),
    bib(relatorio_minoritario,[philip_k_dick]),
    bib(maias,[eca_queiroz]),
    bib(farpas,[eca_queiroz,ramalho_ortigao]),
    bib(ubik,[philip_dick])
   ]
   ```

Utilizando os conhecimentos adquiridos durante o semestre apresente duas implementações diferentes para um predicado que transforma a lista de livros na lista de autores referida no ponto anterior. As implementações devem obedecer aos requisitos que se apresentam de seguida:

- `bibtoauth_meta/2` — deve recorrer aos meta-predicados (ex: setof, bagof, etc.);
- `bibtoauth_recur/2` — deve recorrer apenas à recursividade sobre listas.