Predicados de Sistema do Prolog

- Mecanismos para manipulação de Estruturas
- Input, Output, ficheiros,
- All-Solutions (findall), assert, retract.

Como qualquer outra linguagem, o Prolog oferece um conjunto de funcionalidades pré-definidas para fortalecer as potencialidades da mesma. Neste módulo vamos estudar algumas destas funcionalidades, nomeadamente predicados e operadores para a manipulação de estruturas, manipulação de ficheiros, computação de conjuntos de soluções e manipulação de cláusulas.

Manipulação de termos

Vejamos alguns predicados de sistema que ajudam nesta importante tarefa de manipulação de termos ou estruturas. Estes termos podem representar estruturas de dados clássicas, cláusulas ou qualquer outro tipo de estrutura lógica. Assim, é de vital importância ter ferramentas adequadas para a manipulação dos mesmos.

Predicados de tipos:

Predicados que identificam o tipo de dados associado a um termo. Exemplos:

- integer(X) identifica um inteiro, e.g. 5
- atom(X) identifica um termo atómico, e.g. foo
- constant(X) identifica uma constante, e.g. foo
- compound(X) identifica um termo composto, e.g. pai(X, Y)

Uma constante tanto pode ser um termo atómico como um inteiro. Estes nomes variam de sistema Prolog para Prolog.

Notar que as variáveis (que são termos) são tratadas por predicados especiais que vão ser estudados mais a frente (i.e. predicado var).

Acesso a termos Compostos

Por vezes é interessante ter acesso (manipular) à estrutura dos termos compostos. Por exemplo contar e/ou ter acesso aos parâmetros de um termo.

Um predicado especialmente útil é o functor. Tem o formato

```
functor(Termo, Nome, Aridade).
```

O predicado sucede se o termo Termo tiver aridade Aridade e se o seu principal functor tiver como nome Nome.

Duas formas típicas da sua utilização:

Na primeira forma usa-se para determinar nome e aridade de um termo

```
?functor(pai(joao, ana), X, Y)
```

Tem como resposta X = pai e Y = 2.

A outra forma permite, dado um nome e uma aridade, construir um termo:

```
?functor(T, pai, 2)
```

tem como resposta T = pai(X, Y).

Acesso a termos Compostos (Cont)

Outro predicado bastante útil conjuntamente com functor é o arg. Tem o formato

arg(N, Termo, Arg)

Este predicado em vez de construir termos permite aceder a argumentos de um termo. Por exemplo

?arg(N, Termo, Arg)

sucede se Arg for o N -ésimo argumento do termo Termo.

Novamente temos várias formas de uso. Numa, podemos encontrar um argumento especifico de um termo:

?arg(2, pai(joao, ana), X)

respostas X = ana.

O outro permite instanciar uma variável num termo:

?arg(1, pai(X, ana), joao)

A resposta é X = joao.

Exemplo do uso de functor e arg

subtermo(X, Y) se X for um subtermo incluído no termo Y.

```
subtermo(X,X).
subtermo(S,T) :-
        compound(T),
        functor(T,_,N),
        subtermo2(N,S,T).

subtermo2(N,S,T) :-
        N > 1,
        N1 is N - 1,
        subtermo2(N1,S,T).
subtermo2(N,S,T) :-
        arg(N,T,Arg),
        subtermo(S,Arg).
```

subtermo/2 identifica o número de parâmetros que o dado termo tem (usando functor). Verifica também os casos em que os termos são iguais. subtermo2/3 verifica para cada um dos parâmetros se S é subtermo desse parâmetro. Notar que arg obtém os vários parâmetros do termo em questão.

Este programa pode ser usado de duas formas. Uma para verificar se um termo é subtermo de um outro, e noutra para gerar um subtermo de um dado termo e.g. subtermo(Z, p(f(a), Y))

O operador univ (= ..)

O operador binário $= ..$ permite converter termos em listas e vice versa. Assim, a operação

$$?\mathsf{Termo} = ..\mathsf{Lista}$$

sucede se Lista for uma lista cuja cabeça é o nome do termo Termo e a cauda é a lista de parâmetros do termo.

Por exemplo

$$?\mathsf{pai(joao, ana)} = ..\mathsf{L}$$

tem como resposta

$$\mathsf{L} = [\mathsf{pai, joao, ana}]$$

Tal como os predicados functor e arg este também tem dois possíveis usos. Pode construir uma lista dado um termo ou vice versa. Exemplos:

$?\mathsf{X} = ..[\mathsf{pai, A, B}]$

$$\mathsf{X} = \mathsf{pai(A, B)}$$

$?\mathsf{pai(joao, ana)} = ..\mathsf{Xs}$

$$\mathsf{Xs} = [\mathsf{pai, joao, ana}]$$

Input/Output

Como qualquer outra linguagem, o Prolog também inclui facilidades de Entrada e Saída de dados. Estes processos de input/output são implementados pelos predicados read e write.

Exemplos:

write('ola mae'), write(X), write(5).

read(X), nl, write(X), writeln(['O valor de X e', X]).

Nenhum destes dois predicados (obviamente) dá valores alternativos via backtracking. Existem outras variantes de write (versões mais sofisticadas) que permitem, por exemplo, imprimir listas de termos.

Outros exemplos interessantes: (ciclo de entrada de dados até fim aparecer).

```
go :- repeat, read(X), nl, write(X), nl, X == fim.
go.

repeat.
repeat :- repeat.
```

Operadores

Podemos declarar operadores que vão ser definidos pelos nosso predicados. Todos os operadores em Prolog e.g. $=$, mod, $+$, etc estão predefinidos.

Por exemplo, num programa de tratamento de expressões lógicas os operadores lógicos devem estar definidos como operadores infixos (exceptuando a negação que é um operador unário).

Formato do comando:

$$\mathsf{op(precedencia, tipo, operador)}$$

onde precedencia é um número que indica o nível de precedência do operador (varia entre 1 e 1200), tipo pode ser as expressões xfy, xfx, yfx, xf, yf, fx, fy e operador o nome do operador a definir.

As expressões tem o seguinte significado:

Para operadores prefixos e posfixos

xf e fx significa que a precedência do argumento tem de ser mais baixo do que a do operador

yf e fy significa que a precedência do argumento pode ser igual a do operador.

para Infixos,

xfx ambos os argumentos tem menor precedência que o operador,

xfy apenas o argumento à esquerda,

yfx apenas o argumento à direita.

Exemplo:

$$: -\mathsf{op(300, xfx, and)}$$

Manipulação de Ficheiros

Os predicados que permitem a manipulação de ficheiros são:

- see(File), define File como o novo ficheiro de input,
- seeing(File), File unifica com o nome do ficheiro de input actual,
- seen, fecho o ficheiro de input actual.
- tell(File), define File como o novo ficheiro de output,
- telling(File), File unifica com o nome do ficheiro de output actual,
- told, faz flush do ficheiro de output actual.

Uma vez definido qual o novo *stream* de input/output podemos usar os predicados *read* e *write* para ler/escrever nesse novo stream (tal como faziamos para ler e escrever no stdin/stdout).

Programação de Segunda Ordem

Apresentamos agora um conjunto de técnicas que permitem ir além do formalismo de primeiro ordem. Pretende-se apurar a possibilidade de manipular conjuntos de soluções, cláusulas e outros termos de ordem superior.

Assumimos até agora que os programas são residentes em memória. Nunca foi discutido se haveria necessidade de aceder ao programa i.e. às cláusulas que compõem o programa. O predicado clause(Head, Body) permite aceder às várias cláusulas que constituem um programa em memória (i.e. registado na base de dados interna do Prolog).

Por exemplo, assumindo que temos o programa append em memória, a query

$$?\mathsf{clause}(\mathsf{append}(\_, \_, \_), \mathsf{Body})$$

tem como soluções possíveis as substituições

$$\mathsf{Body} = \mathsf{true}$$

e

$$\mathsf{Body} = \mathsf{append}(\mathsf{Xs}, \mathsf{Ys}, \mathsf{Zs}).$$

Podemos assim manipular (usando os predicados para manipular termos já estudados) e até mesmo construir novas cláusulas e queries no programa, à custa das já residentes em memória.

Retirar e adicionar cláusulas

O Prolog possui predicados de adição e remoção de cláusulas. Podemos assim adicionar novas cláusulas ao programa em runtime. Claro está que isto é equivalente a programação com side-effects o que é altamente não recomendável. No entanto, podemos fazer uso destes predicados para simular estruturas de dados e registo de informação que seja possível de eliminar "fisicamente" i.e. simular *destructive assigment*.

Adicionar cláusulas com assert:

```
assert(horario(turma_1,10:00,programacao,lab_10)).
```

assert ou assertz adiciona cláusulas ao fim do programa.

Uma variante deste predicado (asserta) adiciona ao início.

Adição de novas cláusulas deve ser da seguinte forma (notar parênteses):

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

Em Prologs como a implementação SICStus e SWI é necessário adicionar o comando

:− dynamic nome_da_clausula/aridade

para indicar que o predicado especificado é dinâmico i.e. pode-se remover e adicionar cláusulas para este predicado.

Retract e Exemplos

A remoção de cláusulas é feito pelo predicado retract. Este predicado remove a primeira cláusula que unificar com o parâmetro fornecido. Por exemplo

```
retract(horario(turma_1,_,_,_)).
```

remove a primeira cláusula de horários que representa a informação da turma 1.

Existe uma variante que permite remover todas as cláusulas que é o rectractall.

Este predicado podia ser implementado da seguinte forma:

```
retractall(C):- retract(C), fail.
retractall(_).
```

assert e retract serve para implementar enumeros predicados de sistema. Por exemplo copiar um termo pode ser implementado como (assumindo que $tmp é um termo não usado pelo nosso programa):

```
copy(X,Y) :- asserta($tmp(X)), retract($tmp(Y)).
```

Devido aos efeitos não esperados/desejados, estes predicados devem ser usados com precaução e o menos possível! Notar que estes predicados pode ser computacionalmente bastante "caros".

Múltiplas soluções

Por vezes há necessidade de trabalhar com conjuntos de soluções em vez de uma única solução tal como o modelo computacional do Prolog fornece. No entanto, sabemos que o Prolog, durante backtracking, perde toda a informação respeitante a um ramo especifico da árvore de prova. Sendo assim, não é simples usar Prolog puro para calcular o conjunto de todas as soluções de uma query ou até contar o número de soluções para essa mesma query.

Podemos, no entanto, fazer uso de outros predicados que permitam estas desejadas funcionalidades. Estes predicados de sistema são conhecidos por *set-predicates* que podem ser vistos como novas primitivas. Estes predicados podem ser implementados em Prolog fazendo uso dos já conhecidos predicados assert e retract.

Existem vários destes predicados (e.g. setof, bagof, findall). Talvez o mais popular seja o findall. Este predicado colecciona o conjunto de soluções de uma dada query num formato pré-definido. Neste predicado os conjuntos são representados por listas. O predicado tem o seguinte formato:

```
findall(Formato,Query,Lista)
```

Query é a query dada a responder e Formato é o formato a representar as soluções na lista Lista.

Notar que este predicado não devolve alternativas em backtracking e consequentemente não produz falhanços. i.e. falhanço equivale a resposta ser Lista = [ ].

Findall

Exemplos do seu uso na relação pai

```
pai(luis,ana).      pai(rui,olga).
pai(joao,luis).     pai(luis,ze).
pai(joao,rui).      pai(rui,ana).
```

```
filhos_de(X,L) :- findall(Y,pai(X,Y),L).
```

Assim, por exemplo,

$$?\mathsf{filhos\_de}(\mathsf{rui}, \mathsf{L})$$

tinha como resposta

$$\mathsf{L} = [\mathsf{olga}, \mathsf{ana}].$$

outro exemplo,

$$?\mathsf{findall}(\mathsf{filho}(\mathsf{Y}), \mathsf{pai}(\mathsf{luis}, \mathsf{Y}), \mathsf{L}),$$

a resposta é

$$\mathsf{L} = [\mathsf{filho}(\mathsf{ana}), \mathsf{filho}(\mathsf{ze})]$$

Usando findall conjuntamente com a cláusula clause apresentada anteriormente.

$$?\mathsf{findall}(\mathsf{Tail}, \mathsf{clause}(\mathsf{member}(\_, \_), \mathsf{Tail}), \mathsf{L}),$$

resposta

$$\mathsf{L} = [\mathsf{true}, \mathsf{member}(\mathsf{X}, \mathsf{Ys})]$$

Bagof

É o caso geral de findall e tem o mesmo formato. Mas agora é possível fazer backtracking sobre o próprio set-predicate. Ou seja, há backtracking sobre as variáveis ”livres” (que não estão no primeiro parâmetro).

Exemplos na relação pai anterior (onde X é livre).

```
?- bagof(Y,pai(X,Y),L).
Y = _G308, X = joao, L = [luis,rui] ;
Y = _G308, X = luis, L = [ana,ze] ;
Y = _G308, X = rui,  L = [olga,ana] ;
no
```

No entanto é possível expressar que as variáveis não são livres. A construção Var $\wedge$ Goal diz ao bagof para não criar substituições para Var em Goal.

Assim, por exemplo:

```
?- bagof(Y,X^pai(X,Y),L).
Y = _G308, X = _G300, L = [ana,luis,rui,olga,ze,ana] ;
no
```

o que é equivalente a fazer

$$? - \mathsf{findall}(\mathsf{Y}, \mathsf{pai}(\mathsf{X}, \mathsf{Y}), \mathsf{L}).$$

O predicado setof produz os mesmos resultados que bagof mas a lista final aparece ordenada e sem elementos repetidos.

Exemplos (implementar findall/3)

```
findall(Form,Query,_) :-
        asserta($instancia($fim)),
        Query,
        asserta($instancia(Form)),
        fail.

findall(Form,Query,Lista-T) :-
        retract($instancia(Form)),
        repete(Form,Lista-T), !.

repete(X,L-T) :-
        X \== $fim, retract($instancia(X1),
        !, repete(X1,L-[X|T]).)

repete($fim,L-L).
```

A forma de interpretar esta implementação é procedimental pois o jogo de assert, retract faz o desejado efeito de colecção de conjunto de soluções. Form contém pelo menos uma variável comum com Query. $fim funciona como flag de fim de conjunto. Numa primeira fase, por backtracking, calcula-se todas as soluções de Query. Depois o predicado repete colecciona todas as soluções encontradas e adicionadas na forma de cláusulas $instancia através de uma diferença de listas.

Exemplos (implementar forall)

```
forall(Goal,Condicao) :-
       findall(Condicao,Goal,Casos),
       check(Casos).


check([Case|Cs]) :-  Case, check(Cs).
check([]).
```

Este predicado recorre a findall para calcular todas as soluções de Goal. Depois verifica se estas satisfazem a condição condicao com o predicado check. Notar o uso de uma "meta-variável" em check. Case é usada como uma invocação de um predicado.

Exemplo de um uso:

?forall(pai(X, C), homem(C)).

Mais usos de predicados de segunda ordem

Lógica de primeira ordem permite quantificação sobre indivíduos. Lógica de segunda ordem permite expandir a linguagem para ter quantificação sobre predicados. Incorporar esta funcionalidade em Prolog implica usar regras com literais no corpo cujos nomes de predicados são variáveis.

O exemplo mais geral desta noção é o predicado apply. Isto é, aplica um conjunto (lista) de argumentos a um dado predicado. Pode ser implementado como:

```
apply(Predicado,Lista) :-
     Query =..[Predicado|Lista],
     Query.
```

Podemos usar apply no predicado que verifica se os elementos de uma lista possuem uma dada propriedade.

```
has_property([X|Xs],P) :-
      apply(P,[X]),
      has_property(Xs,P).
has_property([],_).
```

Ou também o bastante conhecido (nas linguagem funcionais) map_list.

```
map_list([X|Xs],P,[Y|Ys]) :-
      apply(P,[X,Y]),
      map_list(Xs,P,Ys).
map_list([],_,[]).
```