PIPCA – Técnicas de Programação – 2017/01 – Prof. João Gluz

Atividade de Avaliação 1 – Resolução de Problemas de Programação em Prolog

1. Listas em Prolog

Listas são estruturas lineares para representação de informações. Em Prolog uma lista é uma estrutura no formato:

[$Elem_1$, $Elem_2$, ..., $Elem_n$]

onde cada um dos $Elem_i$ é um átomo ou termo Prolog. Listas são manipuladas com o operador '|', que separa o primeiro elemento (a "cabeça") do resto (a "cauda") da lista (que também é uma lista). Assim, a seguinte expressão:

[1, b, c, 4, e] = [H | T]

irá resultar nas unificações:

H = 1

T = [b, c, 4, e]

ou seja, a seguinte expressão também resulta verdadeiro:

[1, b, c, 4, e] = [1 | [b, c, 4, e]]

A lista vazia é identificada pela seguinte expressão:

[]

Usando estes operadores desenvolver e testar os seguintes predicados sobre listas:

- (a) *pertence*(E, L) – verdadeiro se o elemento E pertence a lista L, falso caso contrário.
- (b) *somatorio*(L,S) – verdadeiro se a lista é formada por valores numéricos e se S é o resultado da soma destes valores.
- (c) *indice*(E, L, I) – verdadeiro se o elemento E está na posição de índice I da lista L, falso caso contrário. Faça uma implementação não-determinística deste predicado, ou seja, se o índice I é deixado livre e o elemento E está fixo (unificado com um valor), este predicado retorna o índice do elemento na lista L, se ele existir na lista, por outro lado, se o elemento E é livre e o índice I está fixo então retorna em E o elemento que está na posição I.
- (d) *altera*(E, L1, I, L2) – verdadeiro se a lista L2 for igual a lista L1, exceto (possivelmente) pela posição de índice I, que deverá ser igual ao elemento E. Este predicado efetivamente altera o elemento armazenado na posição de índice I da lista L1, resultando na lista L2.
- (e) *inverso*(L1, L2) – verdadeiro se a lista L2 é o inverso da lista L1 (ou vice-versa).
- (f) *concatena*(L1, L2, L3) – verdadeiro se a lista L3 é feita pela concatenação das listas L1 e L2. Faça uma implementação não-determinística de forma que se L1 e L2 estão fixados e L3 está livre, então retorna em L3 a concatenação de L1 com L2, por outro lado se L1 (ou L2) está livre e L2 (ou L1) e L3 estão fixos, então L2 deve ser sufixo (ou L1 deve ser prefixo) de L3 e L1 retorna o prefixo (ou L2 retorna o sufixo) da lista L3.
- (g) *insere_elem*(L1, E, I, L2) – verdadeiro se o elemento E for inserido na posição de índice I da lista L1, resultando na lista L2. Falso caso contrário. Note que este predicado também deverá poder ser usado para *eliminar* um elemento da posição de índice I da lista L2, se esta lista for dada como fixa, enquanto L1 e E estiverem livres.

2. Tabelas

Tabelas podem ser facilmente representadas por listas de listas. Assim uma tabela bidimensional com 3 linhas e 2 colunas pode ser representada pela lista de listas:

[[a1, a2], [b1, b2], [b1, b2]]

Quando a tabela possui um número igual de elementos em cada linha, então a tabela efetivamente se transforma em uma **matriz**.

Porém, esse formalismo é bastante flexível porque não há restrições de tamanho em cada dimensão – no caso acima, por exemplo, a lista que representa uma linha pode ter tamanhos variáveis, ou pode ser vazia ([]). Isso permite criar estruturas como:

[[a1,a2,a3], [b1,b2], [], [c1]]

que é uma tabela *irregular* com 3 colunas na primeira linha, 2 na segunda linha, 0 colunas na linha 3 e 1 coluna na linha 4.

Observação: nada impede que este formalismo seja estendido para listas de listas de listas, para tabelas com 3 dimensões e assim por diante.

Agora desenvolva e teste os seguintes predicados para tratar com as tabelas bidimensionais:

- (a) *indice*(E,I,J,T) – verdadeiro se o elemento E está na linha I e coluna J da tabela T, falso caso contrário.
- (b) *altera*(E, T1, I, J, T2) – verdadeiro se a tabela T1 for igual a tabela T1, exceto (possivelmente) pela posição que está na linha I e coluna J, que deverá ser igual ao elemento E. Este predicado efetivamente altera o elemento armazenado na posição que está na linha I e coluna J da tabela T1, resultando na tabela T2.
- (c) *transposta*(M1,M2) – verdadeira se M1 e M2 são matrizes e M2 é a matriz transposta de M1, falso caso contrário. Por exemplo, supondo M1 = [[a1, a2], [b1, b2], [b1, b2]], então *transposta*(M1,M2) deve resultar na unificação M2=[[a1, b1, c1], [a2, c2, b2]].
- (d) *insere_lin*(T1, I, L,T2) – verdadeiro se a linha L for inserida na linha de índice I na tabela T1, resultando na tabela T2. Falso caso contrário. Note que, da mesma forma que no caso do predicado *insere_elem*(), este predicado também deverá poder ser usado para *eliminar* uma linha na posição de índice I da tabela T2, se esta tabela for dada como fixa, enquanto T1 e L estiverem livres.

3. Árvores Binárias

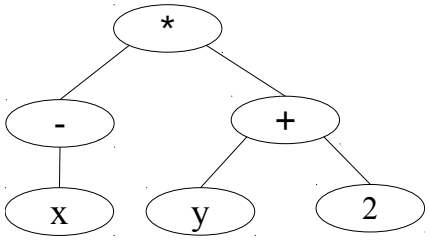
Árvores binárias podem facilmente ser representadas em Prolog através de termos com estrutura similar a seguinte:

arvbin(V, E, D)

onde V é o valor armazenado na raiz da árvore, E é o termo que representa o ramo esquerdo da árvore e D é o termo que representa o ramo direito da árvore. Árvores vazias (ou ramos vazios) são representados pelo termo:

nil

Assim a árvore



pode ser representada pelo termo A1 definido como:

```
A1 = arvbin('*',
        arvbin('-',
                arvbin('x', nil, nil),
                nil),
        arvbin('+',
                arvbin('y', nil, nil),
                arvbin('2', nil, nil) ) )
```

Note que, coincidentemente, árvores binárias podem ser utilizadas para representar expressões aritméticas (ou outros tipos de expressões como lógicas, relacionais, etc.). No caso a árvore acima pode ser utilizada para representar a expressão:

(-x) * (y – 2)

Note que a unificação pode ser usada para percorrer e extrair os componentes de uma árvore. Assim no caso da árvore A1 definida acima, a seguinte expressão:

arvbin(V,E,D) = A1

resulta nas unificações:

V='*'
E= *arvbin*('-', *arvbin*('x', *nil*, *nil*), *nil*)
D = *arvbin*('+', *arvbin*('y', *nil*, *nil*), *arvbin*('2', *nil*, *nil*))

De maneira similar, a expressão:

A1 = *arvbin*('*',*arvbin*('-',E1,D1), *arvbin*(O,E2,D2))

resultará nas unificações:

E1= *arvbin*('x', *nil*, *nil*),
D1= *nil*
O = '+'
E2 = *arvbin*('y', *nil*, *nil*)
D2 = *arvbin*('2', *nil*, *nil*)

Agora desenvolva e teste os seguintes predicados para tratar de árvores binárias:

- (a) *percurso_pre*(A, L) – verdadeiro se A é uma árvore e L é uma lista que contém os elementos armazenados em cada nodo da árvore A ordenada na forma de um percurso pré-ordem da árvore A. Percursos pré-ordem são definidos recursivamente da seguinte maneira:
  - Visite a raiz da árvore
  - Percorra o ramo (subárvore) da esquerda em pré-ordem
  - Percorra o ramo (subárvore) da direita em pré-ordem

- (b) *percurso_pos*(A, L) – verdadeiro se A é uma árvore e L é uma lista que contém os elementos armazenados em cada nodo da árvore A ordenada na forma de um percurso pós-ordem da árvore A. Percursos pós-ordem são definidos recursivamente da seguinte maneira:
  - Percorra o ramo (subárvore) da esquerda em pós-ordem
  - Percorra o ramo (subárvore) da direita em pós-ordem
  - Visite a raiz da árvore

- (c) *percurso_sim*(A, L) – verdadeiro se A é uma árvore e L é uma lista que contém os elementos armazenados em cada nodo da árvore A ordenada na forma de um percurso simétrico da árvore A. Percursos simétricos são definidos recursivamente da seguinte maneira:
  - Percorra o ramo (subárvore) da esquerda em ordem simétrica
  - Visite a raiz da árvore
  - Percorra o ramo (subárvore) da direita em ordem simétrica
- (d) Suponha que L é uma lista formada por termos com o seguinte formato:

  val(X,V)

  onde X são identificadores de variáveis formatos por átomos do Prolog (p.ex. *x*, *y*, *z*, *x1*, *x2*, *a*, *b* ,*c*, ...) e V são valores numéricos. Dessa forma a lista L serve como uma memória armazenando os valores de um conjunto de variáveis. Por exemplo, se L fosse igual a lista:

  L = [*val*(*x*, 10), *val*(*y*, -3), *val*(*a*, 2.3)]

  isso representaria o fato que o valor da variável *x* é 10, o valor de *y* é -3 e o valor da variável *a* é 2.3. Agora implemente o predicado:

calc(L,A,V)

onde L é uma lista de valores, tal como definida acima, A é uma árvore de expressões aritméticas e V o valor numérico calculado para esta expressão. Cada nodo da árvore A pode conter:

- um identificador de variável ou
- um valor numérico ou
- um identificador de operação aritmética que pode ser:
  '+' (soma), '-' (subtração ou complemento), '*' (multiplicação), '/' (divisão), '^' (exponenciação)

- Uma árvore binária de busca é uma árvore binária, onde os valores contidas em cada nodo podem ser comparados e postos em uma ordem específica (ou seja, existe um operador de comparação que permite saber se um valor é maior, menor ou igual à outro). Além disso para cada valor V de qualquer nodo de uma árvore binária de busca é garantido que todos os valores V_e contidos nos nodos do ramo (subárvore) esquerdo são menores que V e que todos os valores V_d contidos nos nodos do ramo (subárvore) direito são maiores que V. Agora implemente o predicado:

  insere_abb(A1, V, A2)

  onde A1 é uma árvore binária de busca (possivelmente vazia, isto é, pode ser apenas nil), V é um valor e A2 é a arvore binária de busca correspondente à inserção do valor B na posição correta da árvore A1.

4. Árvores Genéricas

Árvores genéricas são árvores onde os nodos podem possuir um número qualquer de filhos e este número de nodos filhos pode variar de nodo para nodo da árvore.
Uma árvore genérica pode ser representada facilmente em Prolog combinando listas e árvores através de termos que representam os nodos da árvore genérica e que tem uma estrutura similar a seguinte:

arv(V, L)

onde V é o valor armazenado no nodo e L é uma lista de nodos filhos deste nodo. Note que a ordem da lista L é importante, o primeiro elemento de L é o nodo mais a esquerda, enquanto que o último nodo da lista L é o nodo mais à direita da árvore.

No gerenciamento de projetos, uma *Estrutura Analítica de Projetos* (*EAP*) (do Inglês, *Work breakdown structure - WBS*) é um processo de subdivisão das entregas e do trabalho do projeto em componentes menores, formando uma estrutura em árvore exaustiva, hierárquica (de mais geral para mais específica) orientada aos objetivos que precisam ser cumpridos (ou entregas que precisam ser feitas) para completar o projeto. Também podem ser usados para representar subprojetos do projeto.

Uma árvore genérica pode ser utilizada para representar uma EAP. O projeto é o nodo raiz da árvore. Os nodos filhos representam os objetivos, entregáveis ou subprojetos do projeto. Cada um destes podendo também ser decompostos da mesma forma. Nodos que não contém filhos são os nodos terminais da EAP. Usando termos Prolog pode-se armazenar vários tipos de informações úteis nos nodos, além do nome do objetivo. Em particular, no caso dos nodos terminais pode-se armazenar informações como estimativas de necessidade de recursos, de orçamento e de tempor para atingir o objetivo ou tarefa correspondente ao nodo terminal. No caso dos nodos intermediários pode-se armazenar informações que definam, por exemplo, se os subobjetivos ou subprojetos deste nodo devem ser executados sequencialmente ou podem ser executados em paralelo.

Supondo que cada nodo *terminal* de uma EAP representada como uma árvore genérica contenha como valor, termos com uma estrutura similar a seguinte:

objterm(O,R,C,T)

onde O identifica o objetivo correspondente ao nodo, R é uma lista de identificadores de recursos (pessoas, equipamentos, materiais, etc.) necessários para atingir o objetivo, C é um valor numérico com a estimativa de custo necessário para atingir o objetivo e T uma estimativa em dias do tempo necessário para alcançar o objetivo.

Além disso suponha que cada nodo intermediário da EAP contenha como valor termos com a seguinte estrutura:

objint(O,SP)

onde O identifica o objetivo correspondente ao nodo e SP identifica se os subobjetivos ou subprojetos deste objetivo podem ser buscados em paralelo (*par*) ou em sequência (*seq*).

Agora desenvolva e teste os seguintes predicados:

- *(a) total_recursos*(P, R) – verdadeiro se P é uma árvore genérica que representa a EAP de um projeto e R é a lista de recursos usada no projeto. Note que R não deve conter elementos repetidos, ou seja, se dois objetivos do projeto usarem o mesmo recurso ele deve ser registrado apenas uma vez em R.
- *(b) recursos_obj*(P, O, R) – verdadeiro se P é uma árvore genérica que representa a EAP de um projeto, O é um objetivo deste projeto e R a lista de recursos deste objetivo e de seus subobjetivos.
- *(c) custo_total*(P, C) – verdadeiro se P é uma árvore genérica que representa a EAP de um projeto e C é o somatório dos custos dos objetivos deste projeto.
- *(d) custo_obj*(P, O, C) – verdadeiro se P é uma árvore genérica que representa a EAP de um projeto, O é um objetivo deste projeto e C é o somatório dos custos para atingir este objetivo.
- *(e) prazo_total*(P, O, T) – verdadeiro se P é uma árvore genérica que representa a EAP de um projeto e T é o prazo total estimado para atingir os objetivos deste projeto.
- *(f) prazo_obj*(P, O, T) – verdadeiro se P é uma árvore genérica que representa a EAP de um projeto, O é um objetivo deste projeto e T é o prazo total estimado para atingir este objetivo.

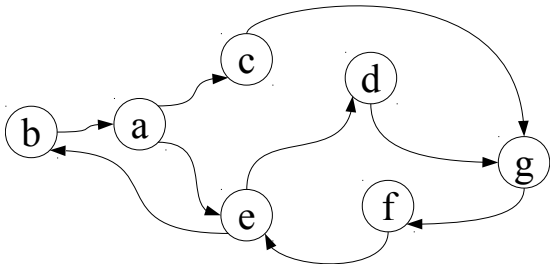
5. Grafos

Um grafo dirigido é formado por uma coleção nodos (ou vértice) e de um conjunto de arcos (ou arestas) dirigidos de um nodo à outro do grafo. Grafos podem ser representados em Prolog através de fatos armazenados na base de fatos com uma estrutura similar a seguinte:

arco(G,N1, N2)

que indica que há um arco do nodo N1 para o nodo N2 no grafo G.

Assim o grafo g1 apresentado abaixo:



Poderia ser definido pelo seguinte conjunto de fatos:

arco(g1, b, a). *arco*(g1, a, c). *arco*(g1, c, g). *arco*(g1, a, e). *arco*(g1, e, b). *arco*(g1, e, d). *arco*(g1, f, e). *arco*(g1, g, f).

Supondo que a base de fatos possua um grafo armazenado no formato descrito acima, implemente e teste os seguintes predicados sobre grafos:

- *nodos_do_grafo*(G, L) – verdadeiro se L é a lista de nodos do grafo G.
- *conectado*(G,N1,N2) – verdadeiro se o nodo N1 está conectado ao nodo N2 de forma direta ou indireta no grafo G.
- *ciclo*(G,N) – verdadeiro se há um ciclo no grafo G que parte e retorna ao nodo N.
- *grafo_ciclico*(G) – verdadeiro se há um ciclo no grafo G.
- *caminho*(G, N1,N2,C) – verdadeiro se C é uma lista de nodos que forma um caminho do nodo N1 ao nodo N2 dentro do grafo G.
- *caminho_mais_curto*(G, N1,N2,C) - verdadeiro se C é uma lista de nodos que forma o caminho mais curto do nodo N1 ao nodo N2 dentro do grafo G.
- *hamiltoniano*(G) – verdadeiro se o grafo G tem um ciclo Hamiltoniano.
- *euleriano*(G) – verdadeiro se o grafo G tem um ciclo Euleriano
- *planar*(G) – verdadeiro se o grafo G é planar.
- *isomorfos*(G1,G2) – verdadeiro se os grafos G1 e G2 são isomórficos.