Prestador:

Na nossa base de conhecimento o predicado prestador representa um indivíduo que presta um determinado cuidado, podemos então deduzir, que no mundo real o prestador seria um profissional de saúde. Sendo assim tem de possuir um **nome**, e como o nome de uma pessoa não é suficiente para a identificar unicamente num serviço, foi necessário definir um **identificador** para cada prestador. Para além destes campos, um prestador deve de possuir uma **especialidade** e uma instituição onde pratica os cuidados. Todavia, um prestador pode trabalhar em **várias instituições**, caso assim deseje e seja permitido.

Posto isto usamos o predicado prestador para descrever a seguinte relação: **prestador( ID,Nome,Especialidade,L\_Instituicoes).** Segue alguns exemplos:

* PRIMEIRO EXEMPLO
* SEGUNDO EXEMPLO

Invariantes

Não é possível um prestador ser contratado por uma instituição onde, nessa, não tenha disponível a especialidade que o prestador possui. Desta forma, decidimos definir um invariante, aquando a inserção de um novo predicado *prestador* que impedisse que este fosse adicionado caso alguma das instituições indicadas não possuísse a respetiva especialidade.

A manipulação do invariante foi feita da seguinte forma:

+prestador(IDp,\_,Esp,LInst)::(validaEspecialidadeInstituicao( Esp,LInst )).

validaEspecialidadeInstituicao( Esp,[] ).

validaEspecialidadeInstituicao( Esp,[C|L] ) :-

instituicao( C,\_,\_,E ),

pertence( Esp,E ),

validaEspecialidadeInstituicao( Esp,L ).

O predicado validaEspecialidadeInstituicao apenas verifica se uma especialidade pertence a todos os elementos de uma lista de instituições. Para tal, “pegámos” na instituição que está à cabeça da lista e fazemos match com o predicado instituição cujo o nome unifica com a cabeça, desta forma conseguimos obter as listas de especialidades. Posteriormente é necessário verificar se a Especialidade pertence à lista obtida, algo que é feito com o predicado pertence. Por fim, é só preciso verificar para o resto da lista.

Caso alguma lista não possua a Especialidade indicada a resposta deste predicado será no e consequentemente a resposta do invariante será no pelo que não deixará acrescentar à base de conhecimento o novo prestador. Se, se obtiver a lista vazia então, a resposta do predicado será yes e consequentemente será também yes pelo que é permitido adicionar o novo prestador à base de conhecimento, havendo uma evolução desta.

invariante de o prestador ter de ter as instituições todas que pertençam à base de conhecimento

Não deve ser possível adicionar um predicado prestador onde alguma das instituições indicadas não pertença à base de conhecimento.

+prestador(ID, \_, \_, LI) :: validaInstituicao(LI).

validaInstituicao([]).

validaInstituicao([C|L]) :-

instituicao(C, \_, \_, \_),

validaInstituicao(L).

Para fazer esta validação apenas se tornou necessário de verificar se cada elemento da lista do predicado prestador unifica com um nome de um predicado instituição que está na base de conhecimento. Isso foi feito com o auxilio de um predicado *validaInstituição*. Caso o predicado dê a resposta *yes* então vai ser possível adicionar um novo predicado de prestador caso a resposta seja *no* então tal predicado não será válido para adicionar à base de conhecimento.

Predicado Extra

Após terminar todos os predicados indicados, decidimos criar uns predicados extras que poderiam ter uma resposta interessante sobre a base de conhecimento.

Utentes com mais custos:

Decidimos implementar um predicado que nos desse o top de clientes com mais custos: utentesMaisCusto(N , R), onde N é o número de utentes que queremos visualizar (caso seja 10, então daria o top10 de clientes que mais gastam) e R seria a lista ordenada por ordem decrescente dos utentes com os seus respetivos custos.

utentesMaisCusto(N , R) :-

solucoes( ( C,ID ),( utente(ID, \_, \_, \_),totalCustoUtente(ID,C) ),L ),

ordenaPar(L, Rs),

take(Rs, N, R).

Para isso, precisávamos de uma lista completa com todos os utentes e o valor total gasto. Através do predicado *soluções* obtivemos esta lista – L. Em seguida, tornou-se necessário ordená-la por forma decrescente. Como cada elemento da lista é um par, foi preciso escrever um novo predicado que fosse capaz de ordenar uma lista de pares. EXPLICAR A ORDENAPAR AQUI? Depois da lista estar ordenada, foi só aplicar um predicado, *take*, que mantém apenas os primeiros N elementos de uma lista.