UNIVERSIDADE DE SÃO PAULO
CAMPUS DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS DE SÃO CARLOS
I.C.M.S.C.

Implementação lógica de um módulo coletor de dados para a construção de sistemas especialistas

RODRIGUES, S.R.; MONARD, M.C.

Nº 63

Notas do ICMSC - USP

ISSN 0103-2577

Implementação lógica de um módulo coletor de dados para a construção de sistemas especialistas

RODRIGUES, S.R.; MONARD, M.C.

Nº 63

São Carlos (SP)
1990

IMPLEMENTAÇÃO LÓGICA DE UM MÓDULO COLETOR DE DADOS PARA A CONSTRUÇÃO DE SISTEMAS ESPECIALISTAS [1]

Solange Rezende Rodrigues

Maria Carolina Monard

Universidade de São Paulo
Instituto de Ciências Matemáticas de São Carlos
Departamento de Computação e Estatística
São Carlos - SP

Instituto de Lógica Filosofia e Teoria da Ciência - ILTC

Sumário

A construção de Núcleos de Sistemas Especialistas pode ser facilitada se ela for realizada dentro de um ambiente que permita ligar, como também alterar, os diversos subsistemas que o constituem, tal que estes possuam características apropriadas para manipular Bases de Conhecimento com características diferentes.

Neste trabalho apresentamos a descrição da implementação, em Prolog, de um desses subsistemas denominado Módulo Coletor de Dados, que é responsável pela aquisição de informação de um meio externo. Este subsistema, que é parte de um ambiente que estamos desenvolvendo para facilitar a construção de Núcleos de Sistemas Especialistas, tem capacidade para processar diversas categorias de perguntas que podem ser feitas ao usuário, como também possui facilidades para validar as respostas por ele fornecidas.

[1] Trabalho realizado com auxílio do CNPq e FINEP

Conteúdo

1 Introdução

Um Sistema Especialista -SE- é um programa que se comporta como um especialista em algum domínio específico de conhecimento. Sendo assim, os SE devem assumir um comportamento semelhante ao dos especialistas humanos em suas áreas de conhecimento.

Na consulta de um cliente a um especialista que é uma pessoa que possue experiência em resolver problemas em uma área limitada de, por exemplo, ciência ou tecnologia, se desenvolve um processo de interação através de uma série de perguntas e respostas efetuadas por ambos. O especialista faz perguntas para reunir informações que são usadas, juntamente com o conhecimento que ele possue, para chegar a alguma conclusão. O cliente também pode questionar o especialista, durante a consulta, para esclarecer suas dúvidas.

Para simular um especialista humano, os SE devem ser capazes de fazer perguntas aos usuários de modo que possam reunir informações necessárias à obtenção de conclusões. Do mesmo modo, o SE deve ter a capacidade de responder perguntas de modo a tornar a sua linha de "raciocínio" o mais clara possível para o usuário. Isto, além de aumentar a confiança do usuário nas conclusões do sistema, permite que o usuário detecte eventuais erros na linha de "raciocínio" do SE.

Devido à pouca flexibilidade dos Núcleos de Sistemas Especialistas, aos quais temos acesso, e tentando minimizar o tempo de implementação, decidimos construir um ambiente para desenvolver Núcleos de Sistemas Especialistas. Para isto, estamos realizando várias implementações dos diversos subsistemas que o constitue.

Estas implementações são abertas no sentido de que é possível, para um usuário não leigo na área de Sistemas Especialista e Prolog, ligar, como também alterar, os subsistemas que ele considera mais apropriados para construir o Núcleo de Sistema Especialista adequado para manipular a Base de Conhecimento de seu interesse.

Entre os subsistemas que compõem um SE encontra-se o Módulo Coletor de Dados. Sempre que o SE necessitar de alguma informação que deve ser fornecida pelo usuário, é responsabilidade do Módulo Coletor de Dados fazer as perguntas referentes à obtenção dessa informação, como também ler e analizar as respostas dadas pelo usuário.

O objetivo deste trabalho é mostrar a idéia usada na implementação Prolog do subsistema Módulo Coletor de Dados. Este subsistema tem capacidade para

processar diversas categorias de perguntas que podem ser feitas ao usuário, como também possui facilidades para validar as respostas por ele fornecidas.

O trabalho está organizado da seguinte forma. Na seção 2 são mostrados os subsistemas que integram o Núcleo do Sistema Especialista. Na seção 3 é descrita a idéia usada na implementação do Módulo Coletor de Dados, como também a interação com os outros subsistemas. A seção 4 contém os tipos de perguntas implementadas, junto com alguns exemplos que mostram a informação que necessita ser dada ao Módulo Coletor de Dados. A seção 5 contém a descrição da implementação e na seção 6 é mostrada, em detalhes, a comunicação entre o Módulo Coletor de Dados e os outros módulos implementados. Finalmente, no apêndice é mostrada a listagem completa da implementação do Módulo Coletor de Dados.

2 Estrutura Básica de um Sistema Especialista

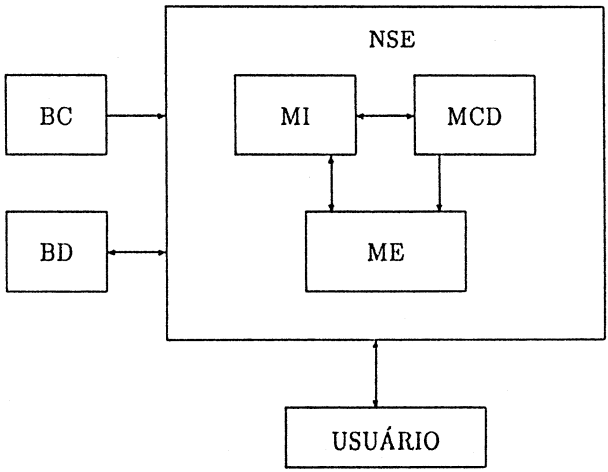
Um Sistema Especialista, como mostra a figura 1, é composto, fundamentalmente, pelos seguintes subsistemas:

- Uma Base de Conhecimento -BC- que contém a modelagem do conhecimento e, em alguns casos, heurísticas para manipular este conhecimento.
- Uma Base de Dados -BD- que constitue a área de trabalho do sistema.
- Um Núcleo do Sistema Especialista -NSE- que é responsável pelo processamento do conhecimento usando alguma linha de "raciocínio", pela justificação ou explicação das novas conclusões obtidas usando este raciocínio e pela interação com o usuário ou equipamentos externos. O NSE pode ser subdividido nos seguintes três módulos:

  Um Motor de Inferência -MI- que processa a BC e a BD usando uma linha de "raciocínio", a fim de propor alguma solução ao problema que está sendo analisado.

  Um Módulo de Explicação -ME- que é responsável pela explicação de *como* o MI chega a certas conclusões, *por que* faz certas perguntas, etc.

  Um Módulo Coletor de Dados -MCD- que é responsável pela comunicação com o usuário e/ou outros sistemas. Este módulo, ativado pelo MI, fará as perguntas ao usuário e obterá as respostas por ele fornecidas. Estas respostas serão usadas pelo MI para inferir novas asserções.



BC: Base de Conhecimento
BD: Base de Dados
NSE: Núcleo do Sistema Especialista
MCD: Módulo Coletor de Dados
MI: Motor de Inferência
ME: Módulo de Explicação

Figura 1: Estrutura Básica de um SE

Note que a figura 1 mostra a estrutura geral de um SE do ponto de vista do usuário do sistema, ao qual não é permitido, em princípio, alterar a BC implementada pelo Engenheiro de Conhecimento que é o responsável pela transferência de conhecimento do perito para o computador.

Dentro do ambiente proposto para desenvolver diversos Núcleos de Sistemas Especialistas já foi implementado um Motor de Inferência com raciocínio "backward chaining", que também manipula dados e regras imprecisas [Monard-89]. O Módulo de Explicação está na fase final de implementação. A seguir são realizadas algumas considerações sobre a implementação do MCD e a interação com estes outros subsistemas. Várias das idéias incorporadas na implementação do MCD derivam de outros trabalhos [Hammond-83], [Niblett-84].

3 Considerações Gerais sobre o Módulo Coletor de Dados Implementado

Deve ser ressaltado que o Módulo Coletor de Dados implementado é geral, isto é, ele pode ser usado como parte de um outro Núcleo de SE. A fim de exemplificar é mostrado, resumidamente, como é a interação com o MCD no sistema que esta sendo implementado.

Neste sistema, a Base de Conhecimento está internamente representada como relações Prolog e o Motor de Inferência está implementado como um meta-interpretador que manipula estas relações usando encadeamento regressivo ou "backward chaining". Considerando as semelhanças e diferenças entre o MI do Prolog e o MI do Núcleo do SE, dividimos as relações da base Base de Conhecimento em cinco grandes grupos. Esta divisão é realizada usando tanto nome de relações como argumentos específicos, que permitem que o MI do NSE reconheça, facilmente, a que grupo pertence cada relação [Monard-89].

Um destes grupos é o das `Relações Perguntáveis` que são as relações que exigem interação com um meio externo ao SE, tais como usuário, equipamentos ou interação com um outro sistema.

As relações perguntáveis são reconhecidas por um predicado da forma

```
perguntavel(<atomo>).
```

onde `<atomo>` denota um predicado na sintaxe de Prolog de Edinburgh.

Na implementação realizada do MI, esta explicação está baseada na cadeia de

regras e metas que conectam esta informação com a interrogação original ao Sistema Especialista. Esta cadeia de regras e metas é usualmente chamada de "trace". O "trace" pode ser visualizado como uma cadeia de regras que conecta a meta que está sendo explorada no momento pelo Motor de Inferência e a meta original que interroga o SE, em uma árvore AND/OR.

Quando o Módulo Coletor de Dados faz uma pergunta e obtém do usuário a resposta por_que, ele ativa o Módulo de Explicação fornecendo-lhe a informação contida no "trace". Na primeira vez que é ativado, o ME mostra, em uma forma apropriada, a regra corrente, isto é, a regra de conhecimento que o Motor de Inferência está tentando provar e, logo após, volta o controle para o MCD que repete a pergunta. Se o usuário voltar a responder por_que, o ME é novamente ativado e a regra imediatamente acima no espaço de busca é mostrada e o controle volta para o MCD que faz a pergunta novamente.

Este processo pode, eventualmente, ser repetido até que a meta original seja alcançada. Neste caso, o ME informa ao usuário que não tem mais explicações a fornecer. Observe que este processo permite verificar a história da prova realizada até o momento.

É importante ressaltar que, se as explicações às perguntas do usuário forem de outro tipo, é muito simples reescrever o código correspondente a fim de acomodar outra estrutura de informação.

4 Base Algorítmica Manipulada pelo Módulo Coletor de Dados

Na estrutura do Sistema Especialista na qual o ambiente implementado está apoiado — figura 3 — a BC é considerada como um todo, ou seja, ela contém toda a informação necessária para a correta execução do sistema. Porém, algumas dessas informações, especificamente programas algorítmicos de escrita e crítica de dados, são manipulados somente pelo MCD. Portanto, é conveniente destacar, por motivos de clareza, esta distinção na Base de Conhecimento. Na figura 2, a área nomeada base Ba refere-se aos programas algorítmicos que são manipulados somente pelo MCD do Núcleo de um Sistema Especialista.

Para cada relação perguntável declarada na BC, deve existir informação específica na base Ba a fim de informar ao MCD o tipo de pergunta a ser realizada, qual a pergunta a ser feita, se a meta e a pergunta a serem feitas admitem variáveis instanciadas, responsabilidade de leitura e aceitação das respostas fornecidas pelo

usuário, etc.

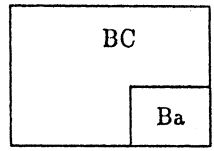


Figura 2: Divisão da Base de Conhecimento

Podemos dividir a informação requerida na base Ba em dois grandes grupos — figura 3 — tendo em conta se a responsabilidade da leitura e validação das respostas do usuário é do implementador da base Ba ou da rotina de leitura e funções de crítica do Módulo Coletor de Dados. Por sua vez, cada um destes grupos pode ser novamente subdividido dependendo da meta e da pergunta a serem realizadas admitirem ou não variáveis instanciadas. Cada um destes grupos correspondem a um predicado na base Ba como é mostrado a seguir.

No caso do MCD ser responsável pela leitura e crítica das respostas do usuário, deve existir na base Ba um predicado `pergunta/3` ou `pergunta/4` com os seguintes argumentos:

```
   pergunta(Meta,Perguntas,Critica)
ou
   pergunta(Meta,VarsInst,Perguntas,Critica)
```

onde

- `Meta` é o argumento da relação perguntável, que é a meta que está sendo provada pelo MI e que deverá ser preenchida com as respostas dadas pelo usuário;
- `Perguntas` corresponde a uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Estes argumentos são programas que devem estar definidos na base Ba e são ativados pelo MCD. Por exemplo, programas para criar janelas, escrever dentro de uma janela determinada, etc. No mínimo, um destes programas deve escrever o texto da pergunta a ser realizada;

- `Critica` determina o tipo de pergunta — afirmativa-negativa, única etc ... — e as funções de crítica das respostas que serão ativadas pelo MCD para validar as respostas fornecidas pelo usuário;
- `VarsInst` determina quais as variáveis de `Meta` que devem estar instanciadas para serem usadas nas perguntas realizadas.

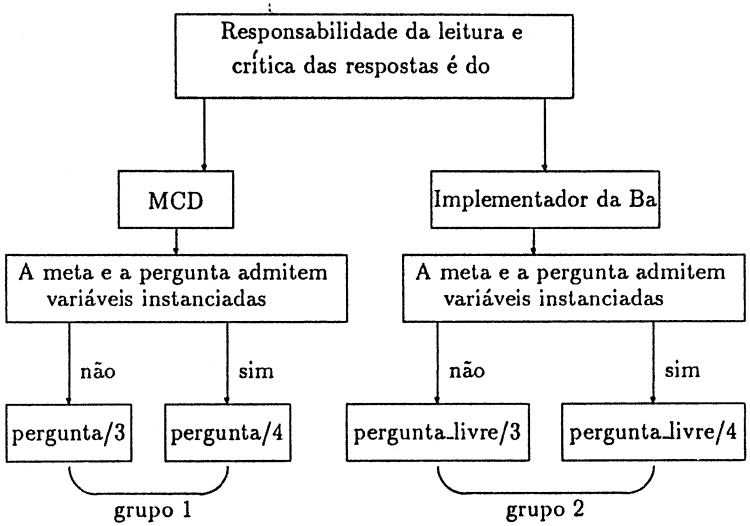


Figura 3: Informação na base Ba

Quando não é responsabilidade direta do Módulo Coletor de Dados realizar a leitura e criticar as respostas do usuário, então, deve existir na base Ba um predicado `pergunta_livre/3` ou `pergunta_livre/4`. Estes predicados devem ser declarados com os seguintes argumentos:

```
pergunta_livre(Meta,Perguntas,Tipo_Resps).
```

ou

```
pergunta_livre(Meta,VarInst,Perguntas,Tipo_Resps).
```

onde

- `Meta` e `VarsInst` são como explicadas anteriormente;

- `Perguntas` corresponde a uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Analogamente ao caso anterior, estes argumentos são programas que devem estar definidos na base Ba e que são ativados pelo MCD. A diferença com o caso anterior é que estes programas devem escrever o texto da pergunta a ser realizada, bem como ler e validar as respostas fornecidas pelo usuário. Além disso, as respostas do usuário devem ser unificadas com variáveis que sejam visíveis para o argumento `Tipo_Resps`;

- `Tipo_Resps` é uma lista onde o primeiro elemento determina o tipo de pergunta — afirmativa-negativa, única ou várias — como no caso anterior, mas os outros elementos são agora nome de variáveis Estas variáveis devem ser idênticas às usadas pelos programas de leitura definidos em `Perguntas`. É responsabilidade desses programas unificar estas variáveis com as respostas fornecidas pelo usuário.

Portanto, os predicados `pergunta/3` e `pergunta/4` correspondem, respectivamente, aos mesmos casos dos predicados `pergunta_livre/3` e `pergunta_livre/4`, com a diferença que nos casos em que `pergunta_livre` for usado, é responsabilidade do projetista da base Ba, definir a pergunta a ser feita bem como obter as respostas para esta pergunta.

O motivo de ter `pergunta` ou `pergunta_livre` com aridade 3 e 4 é o seguinte: suponha que a relação perguntável é

```
perguntavel(peso_altura_idade(Peso,Altura,Idade)).
```

e a pergunta a ser feita é

```
Qual o peso, a altura e a idade do paciente?
```

ou seja, a pergunta não carrega nenhuma informação adicional ligada a esta meta. Neste caso, `pergunta/3` ou `pergunta_livre/3` deve ser usada. Suponha, entretanto, que desejamos realizar uma pergunta personalizada, pois se está perguntando o peso a altura e a idade de um determinado paciente, por exemplo

```
Qual o peso, a altura e a idade de Jose Silva ?
```

neste caso, a relação perguntável seria, por exemplo

```
perguntavel(peso_altura_idade(Paciente,Peso,Altura,Idade)).
```

e `pergunta/4` ou `pergunta_livre/4` deve ser usada, onde o segundo argumento é uma lista de um elemento que contém a variável que deve estar instanciada na chamada, neste caso, a correspondente a `Paciente`.

A razão é que, ao ser ativado, o MCD faz com que todas as variáveis de `Meta` sejam variáveis livres ("desinstancia") exceto aquelas que estão declaradas no segundo argumento de `pergunta/4` ou `pergunta_livre/4`, a fim de preencher estas variáveis livres com as respostas fornecidas pelo usuário.

A regra geral a ser usada é a seguinte: se a pergunta a ser realizada não carrega nenhuma informação, `pergunta/3` ou `pergunta_livre/3` deve ser definida, caso contrário `pergunta/4` ou `pergunta_livre/4` deve ser definida. Na próxima seção são descritos, informalmente, os argumentos das relações `perguntavel/1`, `pergunta/3`, `pergunta_livre/3`, `pergunta/4` e `pergunta_livre/4` bem como são mostrados alguns exemplos juntamente com os programas de crítica dos dados correspondentes.

Resumindo, o Motor de Inferência deve ser capaz de reconhecer as relações perguntáveis e, se a pergunta ainda não tiver sido realizada, o MI deve ativar o Módulo Coletor de Dados para realizar as perguntas correspondentes a esta meta. Portanto, as metas que são perguntáveis devem ser declaradas como tal na Base de Conhecimento, isto é

```
perguntavel(Meta).
```

Além disso, devem estar declarados na base Ba o modo como a pergunta correspondente a esta `Meta` deve ser formulada, o tipo da pergunta (explicado a seguir), bem como funções — explícitas no caso das perguntas do grupo 1 ou implícitas no caso das perguntas do grupo 2 — para criticar as respostas do usuário, ou seja, verificar se elas são ou não aceitáveis.

5 Tipos de Perguntas Implementadas

Os tipos de perguntas implementadas são:

- *afirmativas ou negativas*, para perguntas que necessitam de respostas sim ou não;
- *únicas*, para perguntas que devem ser feitas somente uma vez;

- *várias*, para perguntas que admitem várias respostas. Neste caso, a pergunta é feita várias vezes até que o usuário responda `basta` ou `pare`.

A seguir, são mostrados alguns exemplos descompromissados correspondentes a cada tipo de pergunta, que mostram a necessidade dos diversos tipos de informações na base Ba. Lembre que o Módulo Coletor de Dados permite ao projetista da base Ba decidir se a leitura das respostas fornecidas pelo usuário, para uma meta que foi declarada como perguntável na Base de Conhecimento, é realizada pelo Módulo Coletor de Dados ou é de exclusiva responsabilidade do implementador das perguntas. Esta diferença é feita usando as relações de nome `pergunta` e `pergunta_livre` declaradas na base Ba.

A fim de não confundir, são definidos programas muito simples para escrever e criticar dados. Porém, estes programas podem ser, eventualmente, bastante complexos. É mostrado também o resultado da execução do programa `faz_pergunta/4`, que ativa o Módulo Coletor de Dados onde as respostas fornecidas pelo usuário estão precedidas pelo símbolo '>'.

Deve ser salientado que a resposta `nao_sei` é aceita nos três tipos de perguntas bem como a resposta `por_que` que, como já foi visto, ativa o ME para explicar o motivo desta pergunta sempre que o MCD fizer parte de um NSE. No caso em que `pergunta_livre/3` ou `pergunta_livre/4` for definido, fica por conta do projetista da base Ba fornecer ao usuário a opção de responder `nao_sei` `por_que` e `basta` ou `pare` no caso em que o tipo de pergunta é várias.

5.1 Tipo de Pergunta que Admite Resposta Afirmativa ou Negativa

Suponha que seja necessário saber se um fiador de um cliente de uma imobiliária conhece suas obrigaçoẽs como fiador. Neste caso poderiamos ter:

- na BC

```
perguntavel(obrigacoes_fiador).
```

- na Ba

```
pergunta(obrigacoes_fiador,pg(perg1),[s-n]).

perg1:-
```

```
write('Voce sabe das suas obrigacoes como fiador ?'),
nl.
```

O argumento de perguntavel/1 é sempre a meta a ser perguntada e deve ser igual ao primeiro argumento da relação pergunta/3 correspondente. O segundo argumento é uma estrutura onde o funtor é um átomo qualquer — pg neste caso — e possui um número variável de argumentos. Estes argumentos correspondem aos programas de escrita que serão ativados pelo MCD. O terceiro argumento é uma lista onde o primeiro elemento — também o único elemento no caso de respostas afirmativas ou negativas, pois neste caso o MCD já possui a função de crítica destas respostas — indica o tipo de pergunta.

Segue-se um exemplo de execução correspondente a este caso.

```
?-faz_pergunta(obrigacoes_fiador,MSaida,Inf_PorQue,Classe).
Voce sabe das suas obrigacoes como fiador ?
>sim.
```

o programa é bem sucedido com

```
MSaida = obrigacoes_fiador
Classe = sim
```

No caso de ser necessário saber se um fiador específico conhece suas obrigações, a relação pergunta/4 deve ser usada, isto é:

- na BC

```
perguntavel(obrigacoes_fiador(Cliente)).
```

- na Ba

```
pergunta(obrigacoes_fiador(Cliente),[Cliente],
        pg(perg2(Cliente)),[s-n]).

perg2(Cliente):-
  write(Cliente),
  write(' voce sabe das suas obrigacoes como fiador ?'),
  nl.
```

A seguir, são usados estes mesmos exemplos para ilustrar os casos onde a leitura das respostas é de exclusiva responsabilidade do projetista das perguntas.

Para o primeiro caso onde o predicado pergunta/3 é substituído por pergunta_livre/3 poderiamos ter:

- na BC

```
perguntavel(obrigacoes_fiador).
```

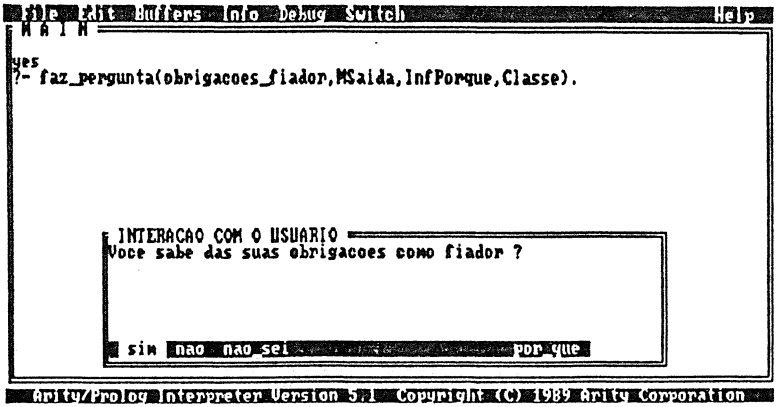
- na Ba

```
pergunta_livre(obrigacoes_fiador,pg(perg1(X)),[s-n,X]).

perg1(X):-
  faz_janela,
  write('Voce sabe das suas obrigacoes como fiador ?'),
  ativa(7,0,X),
  exit_popup.

faz_janela:-
  create_popup(' INTERACAO COM O USUARIO ',
               (14,10),(22,68),(79,-79)).

begin_menu(resposta,50,
          colors((113,31),(113,31),(123,27),(123,116))).
item($sim$,sim).
item($nao$,nao).
item($nao_sei$,nao_sei).
item(right($por_que$),por_que).
end_menu(resposta).

ativa(L,C,Resposta):-
    send_menu_msg(activate(resposta,(L,C)),Resposta).
```

Neste caso, o segundo argumento de pergunta_livre/3 é uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Estes argumentos correspondem a programas de escrita e leitura que serão ativados pelo MCD. Neste exemplo a pergunta foi escrita em uma janela do tipo "popup" e a leitura, que é de exclusiva responsabilidade do projetista da base Ba, é realizada usando um menu que somente aceita respostas sim, nao, nao_sei e por_que. O

terceiro argumento é uma lista onde o primeiro elemento indica o tipo de pergunta (s-n) e o outro é uma variável que vai ser instanciada com a resposta válida do usuário, para ser posteriormente avaliada, se for o caso, pelo MCD.

Segue-se um exemplo de execução correspondente a este caso.



após o usuário responder `sim` o programa é bem sucedido com

```
MSaida = obrigacoes_fiador
Classe = sim
```

No caso de saber se um fiador específico conhece suas obrigações, o predicado `pergunta_livre/4` deve ser usado, isto é:

- na BC

```
perguntavel(obrigacoes_fiador(Cliente)).
```

- na Ba

```
pergunta_livre(obrigacoes_fiador(Cliente),[Cliente],
                pg(perg2(Cliente,X)),[s-n,X]).

perg2(Cliente,X):-
  faz_janela,
```

```
  write(Cliente),
  write(' voce sabe das suas obrigacoes como fiador ?'),
  ativa(7,0,X),
  exit_popup.
```

5.2 Tipo de Pergunta a ser Realizada Somente uma Vez

Suponha que seja necessário saber de um fiador, que não é explicitamente identificado, o seu salário e se ele já é fiador de outra pessoa. Neste caso poderiamos ter:

- na BC

```
perguntavel(salario_fiador(Salario,Fiador)).
```

- na Ba

```
pergunta(salario_fiador(Salario,Fiador),pg(perg3),
         [u,critica_salario,critica_e_fiador]).

perg3:-
  write('Qual a sua renda mensal em salarios minimos)?'),
  nl,
  write('Voce ja e fiador de alguem ?'),
  nl.

critica_salario(X):-
  integer(X),
  X > 2,
  !.
critica_salario(_):-
  write('O valor aceitavel para renda deve ser maior que 2.'),
  nl,
  nova_resposta,
  fail.

critica_e_fiador(X):-
  pertence(X,[sim,nao]),
  !.
critica_e_fiador(_):-
```

```
    write('Os valores validos sao sim ou nao .'),
    nl,
    nova_resposta,
    fail.

nova_resposta:-
    write('Responda novamente, por favor.'),
    nl.
```

O primeiro e o segundo argumento de `pergunta/3` são como explicados anteriormente. O terceiro argumento é uma lista onde o primeiro elemento é um átomo, u, que identifica este tipo de pergunta. Os outros elementos da lista são os nomes das relações das funções de crítica na ordem de entrada dos dados. Todas as funções de crítica tem aridade 1.

No exemplo acima, o primeiro nome é `critica_salario`. Isto indica que o predicado `critica_salario/1` é aquele que verifica que a resposta do usuário para salário é um elemento do domínio de valores definido para a variável `Salario` (um número inteiro e maior que 2). O segundo nome indica que o predicado `critica_e_fiador/1` realiza tarefa análoga para a variável `Fiador`. Se a pergunta a ser feita for:

```
Voce ja e fiador de algem ?
Qual a sua renda mensal em salarios minimos ?
```

então, a ordem dos nomes das funções de crítica nesta lista também devem ser trocadas. Neste caso, o terceiro argumento de `pergunta/3` seria

```
[u,critica_e_fiador,critica_salario]
```

Observe que uma função de crítica do tipo

```
critica_nada(_).
```

permite processar respostas que admitem qualquer tipo de valor.

No caso de um fiador explicitamente identificado poderiamos ter:

- na BC

```
perguntavel(salario_fiador(Cliente,Salario,Fiador)).
```

- na Ba

```
pergunta(salario_fiador(Cliente,Salario,Fiador),[Cliente],
         pg(perg4(Cliente)),
         [u,critica_salario,critica_e_fiador]).

perg4(Cliente):-
  write(Cliente),
  write(' qual o sua renda mensal em salarios minimos ?'),
  nl,
  write('Voce ja e fiador de alguem ?'),
  nl.
```

com as mesmas funções de crítica definidas anteriormente. Segue-se um exemplo de execução deste último caso.

```
?-faz_pergunta(salario_fiador(pedro,Salario,Efiador),
               MSaida,Inf_PorQue,Classe).
pedro qual a sua renda mensal em salarios minimos ?
Voce ja e fiador de alguem ?
>-10.
>nao_sei.

O valor aceitavel para renda deve ser maior que 2.
Responda novamente, por favor.
>10.
```

o programa é bem sucedido com

```
MSaida = salario_fiador(pedro,10,nao_sei)
Classe = nao_sei
```

O Módulo Coletor de Dados classifica a resposta na classe `nao_sei` sempre que o usuário responder `nao_sei` a pelo menos uma das informações requeridas na pergunta em questão. Porém, pode ser observado no exemplo acima que as respostas

válidas são processadas pelo MCD e somente as variáveis correspondentes à resposta `nao_sei` são instanciadas com este átomo. Assim, o Motor de Inferência que eventualmente estiver usando este MCD, tem liberdade para analisar individualmente a informação fornecida pelo usuário, bem como processar a classe `nao_sei` de respostas como um todo.

A seguir são considerados os mesmos exemplos anteriores para ilustrar o uso de `pergunta_livre`. No caso de `pergunta_livre/3` poderiamos ter:

- na BC

```
perguntavel(salario_fiador(Salario,Fiador)).
```

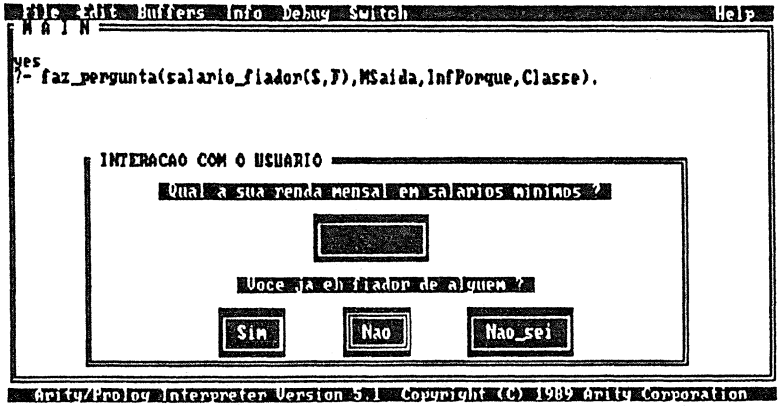
- na Ba

```
pergunta_livre(salario_fiador(Salario,Fiador),
               pg(perg3(Salario,Fiador)),
               [u,Salario,Fiador]).

perg3(Salario,Fiador):-

perg3(Salario,Fiador):-
   dialog_run(j2,trataj2),
   busca_resposta(Salario,Fiador).

begin_dialog(j2,' INTERACAO COM O USUARIO ',(9,8),
             (22,70),(79,-79),112,popup).
ctrl(text,0,$ Qual a sua renda mensal em salarios minimos ? $,
     (1,7),112,47).
ctrl(efield,1,_,(3,23),(112,112),9,$$).
ctrl(text,0,$ Voce ja eh fiador de alguem ? $,(7,15),112,31).
ctrl(push,1,$Sim$,(9,13),(112,112),sim).
ctrl(push,1,$Nao$,(9,26),(112,112),nao).
ctrl(push,1,$Nao_sei$,(9,39),(112,112),ns).
end_dialog(j2).

trataj2(command(nobutton,ok),K):-
   send_control_msg(ef_set_text(Salario,Salario),2,K),
   verif_aceitavel(Salario,NSalario,K),
   assertz(salario(NSalario)),
   !,
```

```
    send_dialog_msg(def_dialog_fn,next_ctrl(2,4),K).
trataj2(command(_,sim),K):-
    !,
    assertz(fiador(sim)),
    exit_dbox(K).
trataj2(command(_,nao),K):-
    !,
    assertz(fiador(nao)),
    exit_dbox(K).
trataj2(command(_,ns),K):-
    !,
    assertz(fiador(nao_sei)),
    exit_dbox(K).
trataj2(Msg,K):-
    def_dialog_fn(Msg,K).

verif_aceitavel(Salario,Salario,K):-
    (pertence(Salario,[por_que,nao_sei]);
    integer(Salario)),!.
verif_aceitavel(Salario,NS,K):-
    send_control_msg(ef_set_text(Salario,$$),2,K),
    send_control_msg(focus(_,2),2,K),
    send_control_msg(ef_set_text(_,NS),2,K).

busca_resposta(Salario,Fiador):-
    salario(Salario),
    fiador(Fiador),
    abolish(salario/1),
    abolish(fiador/1).
```

O primeiro e o segundo argumento do predicado `pergunta_livre/3` são como explicados anteriormente. Neste exemplo, o segundo argumento contém somente o programa `perg3(Salario,Fiador)`, que deve escrever a pergunta e ler as respostas instanciando assim as variáveis da meta que está sendo perguntada — `salario_fiador(Salario,Fiador)` neste caso. Neste exemplo a pergunta é realizada com auxílio de uma caixa de diálogo. O terceiro argumento é uma lista onde o primeiro elemento indica o tipo de pergunta (u) e os outros são as mesmas variáveis que serão instanciadas com as respostas fornecidas pelo usuário.

Segue-se um exemplo correspondente a este caso.



após o usuário fornecer as respostas, o programa é bem sucedido com

```
MSaida = salario_fiador(8,nao)
Classe = sim
```

Para o caso correspondente a `pergunta_livre/4` poderiamos ter:

- na BC

```
perguntavel(salario_fiador(Cliente,Salario,Fiador)).
```

- na Ba

```
pergunta_livre(salario_fiador(Cliente,Salario,Fiador),
               [Cliente],
               pg(perg4(Cliente,Salario,Fiador)),
               [u,critica_salario,Critica_fiador]).
```

onde `perg4(Cliente,Salario,Fiador)` deve ser definido.

5.3 Tipo de Pergunta que Pode ser Feita mais de uma Vez

Suponha que seja necessário saber o tipo e a quantidade de imóveis que um fiador, não explicitamente identificado, possui. Neste caso poderiamos ter:

- na BC

```
perguntavel(imovel_quantidade(Tipo,Quantidade)).
```

- na Ba

```
pergunta(imovel_quantidade(Tipo,Quantidade),pg(perg5),
            [v,critica_tipo,critica_quantidade]).

perg5:-
  write('Que tipo de imovel voce possui ?'),nl,
  write('Qual a quantidade?'),
  nl.

critica_tipo(X):-
  pertence(X,[casa,apartamento,fazenda,lote,outros]),
  !.
critica_tipo(_):-
  write('Os valores validos para tipo de imovel sao:'),
  nl,
  write('casa, apartamento, lote, fazenda e outros.'),
  nl,
  nova_resposta,
  fail.

critica_quantidade(X):-
  integer(X),
  X > 0.
  !.
critica_quantidade(_):-
  write('A quantidade de imoveis deve ser maior que 0.'),
  nl,
  nova_resposta,
  fail.
```

Os argumentos de **pergunta/3** são como descritos no exemplo anterior exceto o do primeiro elemento da lista (terceiro argumento), que é agora o átomo **v** que identifica este tipo de pergunta. A pergunta será repetida até que o usuário responda **basta** ou **pare**.

Segue-se um exemplo de execução deste caso:

```
?-faz_pergunta(imovel_quantidade(Imovel,Quantidade),
                MSaida,Inf_PorQue,Classe).
Que tipo de imovel voce possui ?
Qual a quantidade ?
>casa.
>-3.
A quantidade de imoveis deve ser um numero maior que 0.
Responda novamente, por favor.
>3.
```

o programa é bem sucedido com

```
MSaida = imovel_quantidade(casa,3)
Classe = sim
```

no retrocesso, a pergunta é repetida

```
Que tipo de imovel voce possui ?
Qual a quantidade ?
>apartamento.
>5.
```

o programa é bem sucedido com

```
MSaida = imovel_quantidade(apartamento,5)
Classe = sim
```

novamente, no retrocesso, a pergunta é repetida

```
Que tipo de imovel voce possui e qual a quantidade ?
>pare.
```

e a execução termina.

No caso de um fiador explicitamente identificado poderiamos ter:

- na BC

```
perguntavel(imovel_quantidade(Cliente,Tipo,Quantidade)).
```

- na Ba

```
pergunta(imovel_quantidade(Cliente,Tipo,Quantidade),[Cliente],
         pg(perg6(Cliente)),
         [v,critica_tipo,critica_quantidade]).

perg6(Cliente):-
  write(Cliente),
  write(' que tipo de imovel voce possui ?'),nl,
  write('Qual a quantidade ?'),
  nl.
```

com as mesmas funções de crítica definidas anteriormente.

Os mesmos exemplos são considerados a seguir a fim de ilustrar o uso de pergunta_livre. Para o caso em que o predicado pergunta_livre/3 é usado teria:

- na BC

```
perguntavel(imovel_quantidade(Tipo,Quantidade)).
```

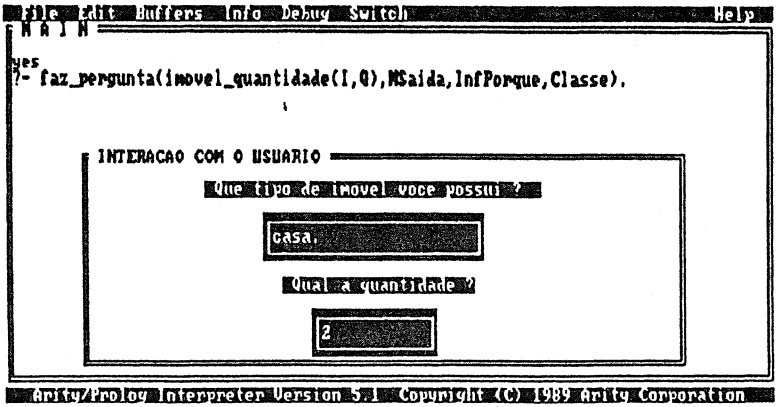
- na Ba

```
pergunta_livre(imovel_quantidade(Tipo,Quantidade),
               pg(perg5(Tipo,Quantidade)),
               [v,Tipo,Quantidade]).


perg5(Tipo,Quantidade):-
   dialog_run(j3,trataj3),
   buscaj3(Tipo,Quantidade).

begin_dialog(j3,' INTERACAO COM O USUARIO ',(9,8),(22,70),
```

```
                (79,-79),112,popup).

ctrl(text,0,$ Que tipo de imovel voce possui ? $,
     (1,12),112,35).
ctrl(efield,1,_,(3,18),(112,112),20,$$).
ctrl(text,0,$ Qual a quantidade ? $,(7,20),112,20).
ctrl(efield,1,_,(9,23),(112,112),10,$$).
end_dialog(j3).

trataj3(command(nobutton,ok),K):-
   send_control_msg(ef_set_text(Tipo,Tipo),2,K),
   verif_aceitavel_tipo(Tipo,NTipo,K),
   assertz(tipo(NTipo)),
   !,
   send_dialog_msg(def_dialog_fn,next_ctrl(2,4),K).

trataj3(command(nobutton,ok),K):-
   send_control_msg(ef_set_text(Quant,Quant),4,K),
   verif_aceitavel_quant(Quant,NQuant,K),
   assertz(quantidade(NQuant)),
   !,
   exit_dbox(K).

trataj3(Msg,K):-
   def_dialog_fn(Msg,K).

buscaj3(Tipo,Quantidade):-
   tipo(Tipo),
   quantidade(Quantidade),
   abolish(tipo/1),
   abolish(quantidade/1).

 onde os programas
      verif_aceitavel_tipo(Tipo,NTipo,K) e
      verif_aceitavel_quant(Quant,NQuant,K)
 devem ser definidos.
```

Os argumentos de `pergunta_livre/3` são como descritos no exemplo de tipo de pergunta a ser realizada somente uma vez — subseção 5.2 — a única diferença é o primeiro elemento da lista (terceiro argumento) que agora é o átomo `v` que indica este tipo de pergunta. O resultado da execução de `faz_pergunta/4` é semelhante

ao mostrado na seção 5.3, somente que neste caso a pergunta é realizada com auxílio de uma caixa de diálogo, mostrada a seguir.



No caso de pergunta_livre/4 ser usado poderiamos ter:

- na BC

```
perguntavel(imovel_quantidade(Cliente,Tipo,Quantidade)).
```

- na Ba

```
pergunta_livre(imovel_quantidade(Cliente,Tipo,Quantidade),
               [Cliente],
               pg(perg6(Cliente,Tipo,Quantidade)),
               [v,Tipo,Quantidade]).

onde perg6(Cliente,Tipo,Quantidade) deve ser definido.
```

A implementação deste tipo de pergunta, que admite várias respostas, está baseada na solução apresentada em [Costa-85] e seu mérito é melhor apreciado quando o Módulo Coletor de Dados for parte integrante de um Núcleo de SE. Neste caso, quando o Motor de Inferência ativa o Módulo Coletor de Dados para fazer uma pergunta que admite várias respostas, o MCD retorna o controle para o MI após a primeira resposta do usuário. O MI é livre — dependendo de como está implementado — para continuar seu raciocínio com esta informação e até pode

encontrar uma solução do problema que está analizando sem ter necessidade de outras respostas. A pergunta será repetida somente se o MI necessitar de mais informações para resolver o problema em questão.

Isto é realizado da seguinte forma: quando a pergunta é de tipo várias, o MCD fornece ao MI a resposta do usuário mas fica pendente uma outra ativação a si próprio. Portanto, se o MI não conseguir encontrar uma solução, no retrocesso ("backtracking"), a pergunta será repetida. A resposta **basta** ou **pare** termina a chamada recursiva dentro do MCD.

6 Descrição da Implementação

A seguir apresentamos a implementação do Módulo Coletor de Dados. Quando for apropriado, os argumentos dos programas são descritos da seguinte forma.

[E]<arg_n>:n_ésimo argumento é de entrada

[S]<arg_n>:n_ésimo argumento é de saida

[A]<arg_n>:n_ésimo argumento é de entrada e de saida

O programa **faz_pergunta**, responsável pela ativação do MCD, possui quatro argumentos

```
faz_pergunta(MEnt,MSaida,Regras,Classe)
```

onde

[E]<arg_1>: a meta a ser provada

[S]<arg_2>: quando o programa faz_pergunta é bem sucedido , este argumento contém a meta que foi provada.

[A]<arg_3>: uma lista que contém as metas anteriores a esta meta. Se o MCD está sendo usado como parte de um NSE a informação contida neste argumento deve ser passada ao ME pelo MCD sempre que o usuário perguntar o por_que de uma pergunta

[S]<arg_4>: a classe da resposta dada pelo usuário. As respostas se classificam em

- `sim` se o usuário responder a todas as perguntas ou se a resposta for positiva

- nao para as respostas negativas
- nao_sei se o usuário responder nao_sei a algumas ou todas as perguntas

O programa faz_pergunta/4 está definido pelas seguintes quatro cláusulas, onde cada uma delas atende a um grupo de quatro perguntas definidas na seção 3.

A primeira cláusula do programa

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta(MEnt,Perg,Tcritica),
   !,
   desinstancie(MEnt,MInt),
   MInt=..[Rel|MSai],
   pergunte(MSai,Perg,GoalAnt,Tcritica,Classe),
   MSaida=..[Rel|MSai].
```

verifica se está definido na Ba um predicado pergunta(MEnt,Perg,TCritica), para obter as informações necessárias para fazer a(s) pergunta(s) ao usuário e as funções de crítica para validar as respostas. Uma vez que existe uma cláusula pergunta/3, o programa faz_pergunta/4 ativa o programa desinstancie/2 da seguinte forma

```
desinstancie(MEnt,MInt)
```

cuja função é simplesmente tomar a meta de entrada (MEnt) e deixar todos os argumentos dessa meta livres. Por exemplo:

```
?-desinstancie(salario_fiador(S,nao),MSai).

S = _005D
MSai = salario_fiador(_01E9,_01DD) ->;

no
?-
```

Após a execução de desinstancie/2, o programa faz_pergunta/4 terá em MInt a meta de entrada com todos os seus argumentos livres. O programa pergunte/5, explicado mais adiante, é ativado para fazer a interação com o usuário. Logo em seguida, usando a pré-definida "univ"(=..), o nome da relação de MEnt e a(s) resposta(s) do usuário, o programa constroi a meta de saida MSaida.

A segunda cláusula

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta(MEnt,Lista,Perg,Tcritica),
   !,
   testa_instanciacao(MEnt,Lista,MSai,Tcritica),
   MEnt=..[Rel|LEnt],
   pergunte(MSai,Perg,GoalAnt,Tcritica,Classe),
   MSaida=..[Rel|MSai].
```

é ativada somente se não existir um predicado pergunta/3 correspondente a esta MEnt. Ela verifica se existe alguma cláusula pergunta/4 na base Ba para obter as informações necessárias para fazer a(s) pergunta(s) ao usuário, as funções de crítica para as respostas e a lista das variáveis que devem permanecer instanciadas. O programa testa_instanciacao/4 verifica quais argumentos de entrada devem permanecer instanciados, que é o caso dos argumentos que estão presentes no segundo argumento de pergunta/4, e deixa os demais argumentos livres. Por exemplo:

```
?-testa_instanciacao(salario_fiador(pedro,S,nao),
                     [pedro],Msai,[u|_]).

MSai = salario_fiador(pedro,_01FE,_01D6) ->;

no
?-
```

Uma vez que testa_instanciacao/4 é bem sucedido, o programa faz_pergunta/4 ativa o programa pergunte/5 e em seguida constroi a meta de saida MSai.

A terceira claúsula

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta_livre(MEnt,Perg,Tresp),
   !,
   desinstancie(MEnt,MInt),
   MInt=..[Rel|MSai],
   obter_resposta(MEnt,MSai,Perg,GoalAnt,Tresp,Classe),
   MSaida=..[Rel|MSai].
```

é ativada somente se não estiverem definidos na base Ba os predicados pergunta/3 ou pergunta/4 correspondentes a meta de entrada. Ela verifica se existe o predicado pergunta_livre/3 na base Ba, a fim de obter as informações necessárias para fazer a pergunta e ler as respostas fornecidas pelo usuário. Em seguida, o programa desinstancie/2, explicado anteriormente, é ativado. Finalmente, o programa obter_resposta/6, explicado mais adiante, é ativado para fazer a interação com o usuário.

A quarta claúsula do programa

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta_livre(MEnt,Lista,Perg,Tresp),
   testa_instanciacao(MEnt,Lista,MSai,Tresp),
   MEnt=..[Rel|_],
   obter_resposta(MEnt,MSai,Perg,GoalAnt,Tresp,Classe),
   MSaida=..[Rel|MSai].
```

executada somente se nenhuma das anteriores tiver sido bem sucedida, procura na base Ba pelo predicado pergunta_livre/4 e deixa livre todos os argumentos da meta de entrada, que não estão presentes no segundo argumento de pergunta_livre/4, usando o programa testa_instanciação explicado anteriormente. O programa obter_resposta/6 é ativado para fazer a interação com o usuário e obter a classe da(s) resposta(s).

O programa pergunte possui cinco argumentos

```
pergunte(MSai,Perg,GoalAnt,Tcritica,Classe).
```

onde

[A] <arg_1>: uma lista que contém os argumentos da meta de entrada. Esta lista pode ter alguns elementos instanciados no caso em que pergunta/4 é encontrada na base Ba.

[E] <arg_2>: uma estrutura com um ou mais argumentos que são programas que contém as informações necessárias para fazer a(s) pergunta(s) ao usuário.

[E] <arg_3>: as regras usadas até o momento. O conteúdo deste argumento é usado para justificar o porque da pergunta.

[E] <arg_4>: uma lista cuja cabeça é o tipo de pergunta e a cauda é composta dos nomes das funções de crítica.

[S]<arg_5>: a classe da resposta dada pelo usuário.

Ele consiste de uma única cláusula

```
pergunte(MSai,Perg,GoalAnt,Tcritica,Classe):-
  escreve_texto(Perg),
  leia_tudo(MSai,GoalAnt,Tcritica,Resp,Perg),
  resposta(MSai,Perg,GoalAnt,Tcritica,Resp,Classe).
```

O programa `pergunte/5` faz a pergunta ao usuário através de `escreve_texto/1`. Feita a pergunta, o programa `pergunte/5`, através de `leia_tudo/5`, lê as respostas dadas pelo usuário e verifica se elas estão de acordo com as funções de crítica definidas na base Ba. Durante esta verificação, se o usuário responder algo que não está de acordo com a função de crítica, o programa informa as respostas válidas para esta pergunta e pede que o usuário responda novamente, verificando em seguida se a nova resposta é um valor aceitável. Por exemplo, em uma pergunta como

```
Que tipo de imóvel voce possui e qual a quantidade ?
```

se o usuário responde um valor negativo para a quantidade e um valor válido para o tipo de imóvel, o programa informa ao usuário os valores aceitáveis para a quantidade e solicita uma nova resposta para quantidade, ou seja, este programa somente solicita a repetição de valores inválidos.

O programa `resposta/6` classifica as respostas do usuário de acordo com as classes de respostas implementadas (`sim`, `nao`, `nao_sei`) ou faz novamente a pergunta ao usuário no caso em que o tipo de resposta é "várias" e o programa ainda não foi informado que pode para de perguntar (pare ou basta). Isto é realizado pelo programa `resposta/6` deixando uma solução pendente. Quando o MI, ligado com o MCD, está tentendo provar uma meta perguntável que admite várias respostas e a resposta dada pelo usuário através do MCD não for suficiente para resolver todo o problema então, no retrocesso, o MCD busca uma nova resposta usando a solução pendente da cláusula `resposta/6`.

O programa `leia_tudo` possui cinco argumentos

```
leia_tudo(MSai,GoalAnt,Tcritica,Resp,Perg)
```

onde

[S]<arg_4>: é uma lista contendo a(s) resposta(s) do usuário já validadas pelas funções de crítica.

Os outros são argumentos de entrada instanciados pelo programa `pergunte/5`.

O programa `leia_tudo/5` é constituido de uma única cláusula

```
leia_tudo(Goal,MetaAnt,[Tipo|Critica],Resp,Perg):-
  faz_leitura(Resp1),
  transfira(MetaAnt,Resp1,Resp2),
  leia_resto(Resp2,Tipo,Critica,Resp3,Perg),
  teste(Goal,Tipo,Critica,Resp3,Resp).
```

Ele lê a primeira resposta do usuário através de `faz_leitura/1`. O programa `transfira/3` verifica a resposta lida. Se a resposta for `porque`, `transfira/3` permite ativar o ME através de `explica_porque/1` com a informação necessária para explicar por que a pergunta foi feita e em seguida repete a pergunta a fim de obter uma nova resposta. O programa `leia_resto/5` é ativado para verificar se ainda existe alguma resposta para ser lida. Esta verificação é feita usando as funções de crítica, pois para cada função de crítica uma resposta deve ser lida. Se o tipo de resposta for diferente de `s-n` ou do tipo "para", no caso em que a pergunta admitir várias respostas, as respostas são armazenadas em uma lista.

O programa `teste/5` é ativado para verificar se as respostas que estão na lista construida por `leia_resto/4` estão corretas segundo as funções de crítica definidas na base Ba. Durante esta verificação, se houver alguma resposta errada, é dada ao usuário uma mensagem informando-o do erro cometido e o usuário deve entrar com uma nova resposta que, se estiver correta , substitui o valor incorreto fornecido anteriormente.

O programa `resposta/6`, que faz parte do corpo do programa `pergunte/5`, classifica a resposta lida nas classes sim, nao ou nao-sei e posteriormente repete a pergunta ao usuário no caso em que a pergunta admitir várias respostas e o usuário ainda não informou ao sistema para parar de perguntar. O programa

```
resposta(MSai,Perg,GoalAnt,Tcritica,Resp,Classe)
```

é constituido de seis cláusulas, onde:

`[A]`<arg_1>:é uma lista que contém os argumentos da meta de entrada. Estes argumentos serão instanciados com as respostas do usuário.

`[S]`<arg_6>: a classe da resposta dada pelo usuário.

Os outros argumentos são argumentos de entrada instanciados pelo programa pergunte/5.

A primeira cláusula

```
resposta(Goal,Perg,GoalAnt,Tcritica,Resp,nao_sei):-
  (nao_sei(Resp),!);(on(nao_sei,Resp),Goal=Resp),
  !.
```

é bem sucedida se a resposta dada pelo usuário é nao-sei ou se ele responder nao-sei a algumas perguntas. Nestes casos, $<arg_6>$ é instanciado com nao-sei e os argumentos da lista de entrada, $<arg_1>$, unificam com as correspondentes respostas da lista das respostas já criticadas,$<arg_5>$.

A segunda cláusula

```
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,Classe):-
  tsim_nao(Tipo),
  ifthenelse(sim(Resp),Classe=sim,Classe=nao),
  !.
```

verifica se a resposta é do tipo sim-nao. Se for bem sucedido, $<arg_6>$ unifica com sim ou nao dependendo da resposta do usuário.

A terceira cláusula

```
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,para):-
  tpara(Resp),
  !,
  fail.
```

verifica se a resposta é do tipo "tpara", neste caso a cláusula falha.

Na quarta cláusula

```
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,sim):-
  tunica(Tipo),
  !,
  Goal=Resp.
```

se a resposta esperada é do tipo "tunica", <arg_1 > unifica com a lista das respostas já criticadas e <arg_6 > unifica com `sim`.

A quinta cláusula

```
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,sim):-
  tvarias(Tipo),
  Goal=Resp.
```

é semelhante a anterior para respostas do tipo "tvarias".

Se esta cláusula é executada, isto significa que as cláusulas anteriores falharam e que a resposta é do tipo "tvarias". Como esta cláusula não tem corte, quando o MI ou o usuário receber a solução dada pelo sistema e desejar outra solução, no retrocesso ("backtraking") esta sexta cláusula será ativada.

A sexta cláusula

```
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,Classe):-
  tvarias(Tipo),
  pergunte(Goal,Perg,GoalAnt,[Tipo|Critica],Classe).
```

repete a pergunta ao usuário, pois, se uma pergunta admite várias respostas, o sistema repete a pergunta até que o usuário responda pare ou basta.

O programa `obter_resposta` possui seis argumentos

```
obter_resposta(MEnt,MSai,Perg,GoalAnt,TResps,Classe)
```

onde:

[E] <arg_1 >: a meta a ser provada;

[A] <arg_2 >: uma lista que contém os argumentos da meta de entrada. Esta lista pode ter alguns elementos instanciados no caso em que `pergunta_livre/4` é encontrada na base Ba;

[E] <arg_3 >: contém o predicado que é responsável por fazer a(s) pergunta(s) e todas as leituras que somente são transferidas ao MCD para ele classificar a resposta ou processar a resposta `por_que`;

[E] <arg_4 >: as regras usadas até o momento. O conteúdo deste argumento é usado para justificar o porque da pergunta;

[E]<arg_5>: uma lista cuja cabeça é o tipo de pergunta e a cauda é composta das variáveis que serão instanciadas com as respostas fornecidas pelo usuário;

[S]<arg_6>: a classe da resposta dada pelo usuário;

O programa `obter_resposta/6` consiste de uma única cláusula.

```
obter_resposta(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe):-
   escreve_texto(Perg),
   verifique_instanciacao_LEnt(Resps),
   obter_classe(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe).
```

O programa `obter_resposta/6` faz a pergunta e lê as respostas fornecidas pelo usuário através da execução do argumento de `escreve_texto/1`. Em seguida, o programa `verifique_instanciacao_LEnt/1` verifica se o usuário respondeu por_que a alguma pergunta ou se todos os elementos da lista `Resps`, que são os mesmos que foram instanciados com as respostas do usuário, estão instanciados. Se algum elemento não estiver instanciado o programa envia uma mensagem de erro avisando que a base Ba esta com problemas. O programa `obter_classe` é ativado para verificar as respostas do usuário e obter a classe destas respostas (sim, nao ou nao_sei).

O programa `obter_classe` está constituido de sete cláusulas.

A primeira cláusula

```
obter_classe(MEnt,MSai,Perg,[Cab|Cauda],[T|Resps],Classe):-
  verifica_porque(Resps),
  !,
  explica_porque(Cab),
  desinstancie_porque(MEnt,NMSai,NPerg,NTResp),
  obter_resposta(MEnt,NMSai,NPerg,Cauda,NTResps,Classe).
```

verifica se o usuário respondeu por_que a alguma pergunta usando o programa de `verifica_porque/1`. Se este programa for bem sucedido, o MCD ativa o Módulo de Explicação através de `explica_porque/1` e, em seguida, deixa livre todas as variáveis que foram instanciadas com as respostas fornecidas pelo usuário usando o programa `desinstancie_porque/4`. Finalmente, ativa o programa `obter_resposta` para repetir a pergunta e obter novas respostas.

A segunda cláusula

```
obter_classe(_,MSai,_,_,[T|Resps],nao_sei):-
  verifica_naosei(Resps),
  !,
  verifica_unificacao(MSai,[T|Resps]).
```

é bem sucedida se a resposta dada pelo usuário é `nao_sei` ou se ele responder naosei a alguma(s) pergunta(s). Isto é verificado pelo programa `verifica_naosei/1`. Nestes casos, $<arg_6>$ é instanciado com `nao_sei` e os argumentos da lista de entrada, $<arg_2>$, unificam com as correspondentes respostas da lista das respostas, que encontram-se na cauda do $<arg_5>$.

A terceira cláusula

```
obter_classe(_,MSai,_,_,[T,Resp],Classe):-
  tsim_nao(T),
  ifthenelse(sim(Resp),Classe=sim,Classe=nao),
  !.
```

verifica se a resposta é do tipo `sim-nao`. Se for bem sucedido, $<arg_6>$ unifica com `sim` ou `nao` dependendo da resposta do usuário.

A quarta cláusula

```
obter_classe(_,MSai,_,_,[T|Resps],para):-
  verifica_tpara(Resps),
  !,
  fail.
```

verifica se a resposta é do tipo "tpara", neste caso a cláusula falha.

Na quinta cláusula

```
obter_classe(_,MSai,_,_,[T|Resps],sim):-
  tunica(T),
  !,
  unifica(MSai,Resps).
```

se a resposta esperada é do tipo "tunica", $<arg_1>$ unifica com a lista das respostas usando o programa `unifica/2` e $<arg_6>$ unifica com `sim`.

A sexta cláusula

```
obter_classe(_,MSai,_,_,[T|Resps],sim):-
  tvarias(T),
  unifica(MSai,Resps).
```

é semelhante a anterior para respostas do tipo "tvarias". Se esta cláusula é executada, isto significa que as cláusulas anteriores falharam e que a resposta é do tipo "tvarias". Como esta cláusula não tem corte, quando o MI ou o usuário receber a solução dada pelo sistema e desejar outra solução, o sistema no retrocesso(backtraking) ativa a sétima cláusula.

A sétima cláusula

```
obter_classe(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe):-
  tvarias(T),
  obter_resposta(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe).
```

que repete a pergunta ao usuário, pois, se uma pergunta admite várias respostas, o sistema repete a pergunta até que o usuário responda pare ou basta.

Como foi explicado anteriormente, no caso onde a leitura das respostas é responsabilidade do MCD, a pergunta a ser realizada pelo sistema é executada através da informação contida no segundo ou terceiro argumento de `pergunta/3` ou `pergunta/4` respectivamente. Este argumento é uma estrutura onde o funtor é um átomo qualquer e possui um número variável de argumentos. Veja que estes argumentos podem ser qualquer programa, por exemplo, programas para criar janelas, escrever dentro de alguma janela determinada, etc. A leitura é realizada através de `faz_leitura/1` que é constituido de duas cláusulas. A primeira cláusula

```
faz_leitura(Resp):-
  rotina_leitura,
  !,
  read(Resp).
```

chama uma rotina de leitura que deve estar definida na base Ba. O programa `rotina_leitura/0` somente informa ao usuário que está esperando pela sua resposta. É responsabilidade do projetista do Núcleo do SE definir este programa. Após a execução da `rotina_leitura/0` o programa lê a resposta. A segunda cláusula

```
faz_leitura(Resp):-
  read(Resp).
```

simplesmente lê a resposta. Esta cláusula só vai ser executada se não existir uma `rotina_leitura/0` definida na base Ba.

Observe que a forma mais elementar de implementar este programa é sem definir `rotina_leitura/0`. Nos exemplos mostrados o programa `rotina_leitura/0` foi definido, na base Ba, da seguinte forma:

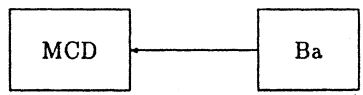
```
rotina_leitura:-
  write(>).
```

A listagem completa do Módulo Coletor de Dados encontra-se no apêndice.

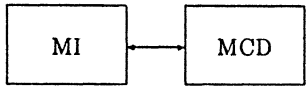
7 Comunicação entre os Módulos

A figura 4 mostra a estrutura do Sistema que estamos implementando considerando a divisão da Base de Conhecimento citada anteriormente. Isto é, a base Ba refere-se aos programas algorítmicos da BC, que são manipulados somente pelo MCD.

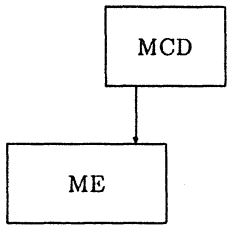
A seguir, é mostrado quais são os programas responsáveis pela comunicação com o Módulo Coletor de Dados, quando este faz parte do Núcleo de um Sistema Especialista.



Esta comunicação é realizada através de `pergunta/3`, `pergunta/4`, `pergunta_livre/3` e `pergunta_livre/4`. No caso de ser usado `pergunta/3` ou `pergunta/4` o MCD também faz uso do programa `rotina_leitura/0` se ele estiver definido na base Ba para informar ao usuário que está esperando pela sua resposta.



Esta comunicação é realizada através de `faz_pergunta/4`. O MI é responsável pela chamada e o MCD simplesmente fornece as respostas.



O MCD ativa o ME, sempre que o usuário responder `por_que`, através de `explica_porque/1`.

Os detalhes da comunicação entre os outros módulos fogem ao objetivo deste trabalho, estes detalhes podem ser encontrados em [Monard-89].

8 Conclusões

Neste trabalho foi apresentado em detalhe a implementação completa do subsistema Módulo Coletor de Dados que é parte integrante de um ambiente que está sendo desenvolvido em Prolog, especificamente Arity_Prolog [Arity-88], para auxiliar na construção de Núcleos de Sistemas Especialistas com características definidas pelo projetista do sistema.

A interação deste subsistema com o usuário admite várias categorias de perguntas que, em alguns casos, sobrepoem-se como é o caso de respostas do tipo `s-n` que podem também ser definidas como de tipo `unica`.

A fim de exemplificar, considere que é necessário confirmar se o voto de um cidadão é nulo. Isto pode ser realizado de duas formas. Na primeira, declarando

- na BC

```
perguntavel(voto_nulo).
```

- na Ba

```
pergunta(voto_nulo,pg(perg_voto_nulo),[s-n]).

perg_voto_nulo:-
  write('O voto e nulo ?'),nl.
```

Uma outra forma é declarando

- na BC

```
perguntavel(voto_nulo(X)).
```

- na Ba

```
pergunta(voto_nulo(X),pg(perg_voto_nulo),[u,critica_sn]).

perg_voto_nulo:-
  write('O voto e nulo ?'),nl.

critica_sn(X):-
  pertence(X,[sim,nao]),!.
critica_sn(X):-
  write('Os valores validos sao sim, nao.'),
  fail.
```

Ainda nestes casos, pode ser observado que não há conflito no tipo de perguntas implementadas.

Um outro aspecto que deve ser ressaltado na implementação deste subsistema é que não há restrição no número de variáveis que podem ser preenchidas com a(s) resposta(s) do usuário a uma pergunta específica.

Na implementação realizada a entrada e a saida de dados pode ser programada usando um ambiente amigável, por exemplo com janelas, menus, caixas de dialogo, etc. A idéia usada é a de deixar liberdade para o projetista das perguntas determinar o formato apropriado. Para isto, ele pode optar por definir na base Ba relações de nome `pergunta` ou `pergunta_livre` dependendo de como ele deseja implementar a interação com o usuário.

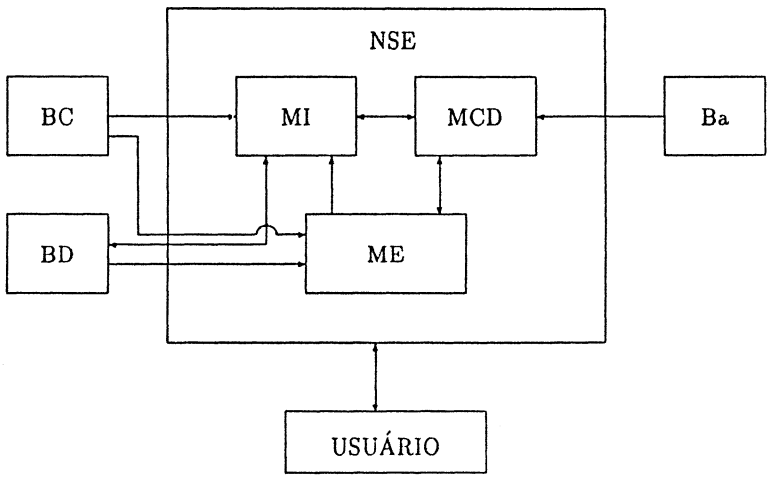


Figura 4: Comunicação entre os Módulos

9 Apêndice: Listagem da Implementação do Módulo Coletor de Dados

```
faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta(MEnt,Perg,Tcritica),
   !,
   desinstancie(MEnt,MInt),
   MInt=..[Rel|MSai],
   pergunte(MSai,Perg,GoalAnt,Tcritica,Classe),
   MSaida=..[Rel|MSai].

faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta(MEnt,Lista,Perg,Tcritica),
   testa_instanciacao(MEnt,Lista,MSai,Tcritica),
   MEnt=..[Rel|_],
   pergunte(MSai,Perg,GoalAnt,Tcritica,Classe),
   MSaida=..[Rel|MSai].

faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta_livre(MEnt,Perg,Tresp),
   !,
   desinstancie(MEnt,MInt),
   MInt=..[Rel|MSai],
   obter_resposta(MEnt,MSai,Perg,GoalAnt,Tresp,Classe),
   MSaida=..[Rel|MSai].

faz_pergunta(MEnt,MSaida,GoalAnt,Classe):-
   pergunta_livre(MEnt,Lista,Perg,Tresp),
   testa_instanciacao(MEnt,Lista,MSai,Tresp),
   MEnt=..[Rel|_],
   obter_resposta(MEnt,MSai,Perg,GoalAnt,Tresp,Classe),
   MSaida=..[Rel|MSai].
```

```
desinstancie(Meta,Meta1):-
   functor(Meta,Rel,A),
   functor(Meta1,Rel,A).

testa_instanciacao(MEnt,Lista,MSai,[Tipo|Critica]):-
   tsim_nao(Tipo),
   !,
   MEnt=..[Rel|LEnt],
   verifique_LEnt(LEnt),
   MSai=LEnt.
testa_instanciacao(MEnt,Lista,MSai,_):-
   teste(MEnt,ListaTrocas),
   MEnt=..[Rel|LEnt],
   instancie(49,LEnt,ListaTrocas,MSai).

pergunte(Goal,Perg,GoalAnt,Tcritica,Classe):-
   !,
   escreve_texto(Perg),
   leia_tudo(Goal,GoalAnt,Tcritica,Resp,Perg),
   resposta(Goal,Perg,GoalAnt,Tcritica,Resp,Classe).

obter_resposta(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe):-
   escreve_texto(Perg),
   verifique_instanciacao_LEnt(Resps),
   obter_classe(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe).

leia_tudo(Goal,MetaAnt,[Tipo|Critica],Resp,Perg):-
   faz_leitura(Resp1),
   transfira(MetaAnt,Resp1,Resp2,Perg),
   leia_resto(Resp2,Tipo,Critica,Resp3),
   teste(Goal,Tipo,Critica,Resp3,Resp).

faz_leitura(Resp):-
   rotina_leitura,
   !,
   read(Resp).
faz_leitura(Resp):-
   read(Resp).

transfira([],Resp1,Resp,Perg):-
   por_que(Resp1),
```

```
    !,nl,
    escreve_mensagem,
    nl,
    escreve_texto(Perg),
    faz_leitura(Resp2),
    transfira(CaudaGoal,Resp2,Resp,Perg).
transfira([CabMeta|CaudaMeta],Resp1,Resp,Perg):-
    por_que(Resp1),
    !,nl,
    explica_porque(CabMeta),nl,
    escreve_texto(Perg),
    faz_leitura(Resp2),
    transfira(CaudaGoal,Resp2,Resp,Perg).
transfira(MetaAnt,Resp,Resp,Perg).

leia_resto(Resp,Tipo,Critica,Resp):-
  (tsim_nao(Tipo);tpara(Resp)),
  !.
leia_resto(Resp,Tipo,[Critica],[Resp]):-
  atomic(Critica),
  !.
leia_resto(Resp1,Tipo,[Crit1|Crit2],[Resp1|Resp2]):-
  atomic(Crit1),
  !,
  faz_leitura(Resp),
  leia_resto(Resp,Tipo,Crit2,Resp2).

teste(Goal,Tipo,Critica,Resp,Resp):-
  tvarias(Tipo),
  tpara(Resp),
  !.
teste(Goal,Tipo,Critica,Resp,Resp):-
  tsim_nao(Tipo),
  !,
  (nao_sei(Resp);sim(Resp);nao(Resp)).
teste(Goal,Tipo,Critica,Resp1,Resp):-
  teste_resto(Goal,Critica,Resp1,Resp).

teste_resto(Goal,[],[],Goal):-
  !.
teste_resto([Var|Varc],LCrit,LResp1,[Var|LResp]):-
```

```
  not(var(Var)),
  !,
  teste_resto(Varc,LCrit,LResp1,LResp).
teste_resto([Var|Varc],[Crit1|Crit2],[Resp|Resp1],
            [nao_sei|Resp2]):-
  nao_sei(Resp),
  !,
  teste_resto(Varc,Crit2,Resp1,Resp2).
teste_resto([Var|VarC],[Crit1|Crit2],[Resp|Resp1],[Resp|Resp2]):-
  Fc=..[Crit1,Resp],
  call(Fc),
  !,
  teste_resto(VarC,Crit2,Resp1,Resp2).
teste_resto([Var|VarC],[Crit1|Crit2],[Resp|Resp1],LResp):-
  faz_leitura(Resp2),
  teste_resto([Var|VarC],[Crit1|Crit2],[Resp2|Resp1],LResp).

resposta(Goal,Perg,GoalAnt,Tcritica,Resp,nao_sei):-
  (nao_sei(Resp),!);(on(nao_sei,Resp),Goal=Resp),
  !.
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,Classe):-
  tsim_nao(Tipo),
  ifthenelse(sim(Resp),Classe=sim,Classe=nao),
  !.
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,para):-
  tpara(Resp),
  !,
  fail.
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,sim):-
  tunica(Tipo),
  !,
  Goal=Resp.
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,sim):-
  tvarias(Tipo),
  Goal=Resp.
resposta(Goal,Perg,GoalAnt,[Tipo|Critica],Resp,Classe):-
  tvarias(Tipo),
  pergunte(Goal,Perg,GoalAnt,[Tipo|Critica],Classe).

obter_classe(MEnt,MSai,Perg,[Cab|Cauda],[T|Resps],Classe):-
  verifica_porque(Resps),
```

```
  !,
  explica_porque(Cab),
  desinstancie_porque(MEnt,NMSai,NPerg,NTResp),
  obter_resposta(MEnt,NMSai,NPerg,Cauda,NTResps,Classe).
obter_classe(_,MSai,_,_,[T|Resps],nao_sei):-
  verifica_naosei(Resps),
  !,
  verifica_unificacao(MSai,[T|Resps]).
obter_classe(_,MSai,_,_,[T,Resp],Classe):-
  tsim_nao(T),
  ifthenelse(sim(Resp),Classe=sim,Classe=nao),
  !.
obter_classe(_,MSai,_,_,[T|Resps],para):-
  verifica_tpara(Resps),
  !,
  fail.
obter_classe(_,MSai,_,_,[T|Resps],sim):-
  tunica(T),
  !,
  unifica(MSai,Resps).
obter_classe(_,MSai,_,_,[T|Resps],sim):-
  tvarias(T),
  unifica(MSai,Resps).
  obter_classe(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe):-
  tvarias(T),
  obter_resposta(MEnt,MSai,Perg,GoalAnt,[T|Resps],Classe).

verifica_unificacao(MSai,[T|Resps]):-
  tsim_nao(T),
  !.
verifica_unificacao(MSai,[T|Resps]):-
  unifica(MSai,Resps).

desinstancie_porque(MEnt,MSai,NPerg,NTResp):-
  pergunta_livre(MEnt,Perg,Tresp),
  !,
  desinstancie(MEnt,MInt),
  pergunta_livre(MInt,NPerg,NTresp),
  MInt=..[Rel|MSai].
desinstancie_porque(MEnt,MSai,NPerg,NTResp):-
  pergunta_livre(MEnt,Lista,Perg,Tresp),
```

```
  testa_instanciacao(MEnt,Lista,MSai,Tresp),
  MEnt=..[Rel|_],
  MLivre=..[Rel|MSai],
  pergunta_livre(MLivre,NLista,NPerg,NTResp).

verifica_tpara([X|R]):-
  tpara(X),
  !.
verifica_tpara([_|R]):-
  verifica_tpara(R).

verifica_porque([X|R]):-
  por_que(X),
  !.
verifica_porque([_|R]):-
  verifica_porque(R).

verifica_naosei([X|R]):-
  nao_sei(X),
  !.
verifica_naosei([_|R]):-
  verifica_naosei(R).

unifica([],[]):-!.
unifica([Cab|Cauda],[Resp|RResp]):-
  var(Cab),
  !,
  Cab=Resp,
  unifica(Cauda,RResp).
unifica([Cab|Cauda],Resps):-
  unifica(Cauda,Resps).

nao(Resp):-
  on(Resp,[nao,n]),
  !.

sim(Resp):-
  on(Resp,[s,sim,yes]),
  !.

nao_sei(Resp):-
```

```
  on(Resp,[nao-sei,ns,nao_sei]),
  !.

por_que(Resp):-
  on(Resp,[por-que,por_que,pq,why]),
  !.

tvarias(Tipo):-
  on(Tipo,[v,varias]),
  !.

tpara(Resp):-
  on(Resp,[basta,pare]),
  !.

tsim_nao(Tipo):-
  on(Tipo,[s-n,sn]),
  !.

tunica(Tipo):-
  on(Tipo,[u,unica]),
  !.

verifique_instanciado(Cab):-
   var(Cab),
   !,
   write($Sua Ba nao esta bem definida.$),
   nl,
   abort.
verifique_instanciado(Cab).

instancie(_,[],[],[]).
instancie(N,[Cab|Cauda],[Cab1|Cauda1],[Cab|Lista1]):-
   var(Cab1),
   !,
   verifique_instanciado(Cab),
   instancie(N,Cauda,Cauda1,Lista1).
instancie(N,[Cab|Cauda],[Cab1|Cauda1],[Var|Lista1]):-
   gere_var(N,Var),
   inc(N,NN),
   instancie(NN,Cauda,Cauda1,Lista1).
```

```
verifique_LEnt([]):-!.
verifique_LEnt([Cab|Cauda]):-
   verifique_instanciado(Cab),
   verifique_LEnt(Cauda).

verifique_instanciacao_LEnt([]):-!.
verifique_instanciacao_LEnt(Resps):-
   verifica_porque(Resps),!.
verifique_instanciacao_LEnt([Cab|Cauda]):-
   verifique_instanciado(Cab),
   verifique_instanciacao_LEnt(Cauda).

execute([]).
execute([X|Y]):-
   call(X),
   execute(Y).

escreve_texto(Perg):-
   Perg=..[Rel|Args],
   execute(Args).

teste(X,Ltrocas):-
   functor(X,Rel,Arid),
   functor(PredLivre,Rel,Arid),
   (pergunta(PredLivre,Lista,_,_);
    pergunta_livre(PredLivre,Lista,_,_)),
   PredLivre =.. [Rel|Lv],
   compara(Lv,Lista,Ltrocas).

compara([],_,[]).
compara([C|Ca],L,[C|L1]):-
   identica(C,L),
   !,
   compara(Ca,L,L1).
compara([C|Ca],L,[0|L1]):-
   compara(Ca,L,L1).

identica(X,[Y|Z]):-
   X==Y,
   !.
```

```
identica(X,[_|Z]):-
   identica(X,Z).

concatenar([],Lista,Lista).
concatenar([Elem|Lista1],Lista2,[Elem|Lista3]):-
   concatenar(Lista1,Lista2,Lista3).

gere_var(N,X):-
   concatenar['Q,'%],[N],Lista),
   name(X1,Lista),
   string_term(X1,X).

escreve_mensagem:-
   write('Nao tenho mais informacoes para explicar "por-que".'),

on(X,[X|_]):-!.
on(X,[_|Z]):-on(X,Z).
```

Referências

[Arity-88] Arity Corporation. *The Arity/Prolog Programming Language.* 1988.

[Costa-85] Costa, A.E.P. *Comunicação pessoal*, 1985.

[Hammond-83] Hammond, P. *APES: A Prolog Expert System Shell.* Dept. of Computing, Imperial College, 1983.

[Monard-89] Monard, M.C.; Rodrigues, S.R. *Implementação Lógica de um Motor de Inferência com Raciocínio Backward Chaining para a Construção de Sistemas Especialistas.* Notas do ICMSC-USP, Nº 51, 1989.

[Niblett-84] Niblett,T. *YAPES - Yet Another Prolog Expert System.* Technical Report TIRM-84-008, Turing Institute, 38 pg, 1984.

NOTAS DO ICMSC/USP

Nº 62 - MONARD, M.C.; NICOLETTI, M.C. - Método sintático de prova de teorema algorítmo de Wang

Nº 61 - GIONGO, M.A.; TÁBOAS, P.Z. - Roses play a role in some inverse problems from bifurcation theory

Nº 60 - MORABITO, R.N.; ARENALES, M.N.; ARCARO, V.F. - An and-or-graph representation for two-dimensional cutting problems

Nº 59 - ACHCAR, J.A. - A bayesian approach to reparametrization of the exponential distribution with type I censored data

Nº 58 - ACHCAR, J.A. - An useful reparametrization for the extrem value distribution

Nº 57 - MASIERO, P.C.; et al - Um ambiente de desenvolvimento baseado na abordagem operacional

Nº 56 - ANDRADE, E.X.L.; BRACCIALI, C.F. - Um método, assemelhado ao de Francis, para determinação de auto-valores de matrizes

Nº 55 - BRUCE, J.W. - Euler characteristics of real varieties

Nº 54 - FURKOTTER, M.; RODRIGUES, H.M - Symmetry and bifurcation to 2"-periodic solutions on nonlinear second order equations with 2"/m-periodic forcings

Nº 53 - RIEGER, J.H.; RUAS, M.A.S. - Classification of A-simple germs from K^n to K^2

Nº 52 - FURKOTTER, M.; RODRIGUES, H.M. - On harmonic and subharmonic solutions of nonlinear second order equations: symmetry and bifurcation