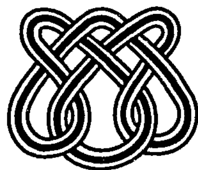
# UNIVERSIDADE DE SÃO PAULO

FERRAMENTA *FLEX* PARA DESENVOLVIMENTO DE SISTEMA BASEADO EM CONHECIMENTO [1]

JAQUELINE BRIGLADORI PUGLIESI
SOLANGE OLIVEIRA REZENDE

Nº 27

## NOTAS DIDÁTICAS



Instituto de Ciências Matemáticas de São Carlos

**Instituto de Ciências Matemáticas de São Carlos**

ISSN - 0103-2577

# FERRAMENTA *FLEX* PARA DESENVOLVIMENTO DE SISTEMA BASEADO EM CONHECIMENTO [1]

**JAQUELINE BRIGLADORI PUGLIESI**
**SOLANGE OLIVEIRA REZENDE**

**Nº 27**

NOTAS DIDÁTICAS DO ICMSC

São Carlos
Mai./1997

# Ferramenta *Flex* para Desenvolvimento de Sistema Baseado em Conhecimento[1]

**Jaqueline Brigladori Pugliesi**
**Solange Oliveira Rezende**

Universidade de São Paulo
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos - SP

e-mail: {jbpuglie, solange}@icmsc.sc.usp.br

Abril de 1997

[1]Trabalho realizado com o auxílio do CNPq e da FAPESP.

# Conteúdo

**1. INTRODUÇÃO** ........ **1**

**2. SISTEMA BASEADO EM CONHECIMENTO** ........ **4**

2.1. Estrutura Básica de um SBC ........ 4
2.2. Domínios de Conhecimento ........ 5
2.3 Etapas de Desenvolvimento de um SBC ........ 5

**3. FLEX** ........ **7**

3.1. Mecanismos de Inferência ........ 7
*3.1.1. Forward Chaining* ........ *7*
*3.1.2. Backward Chaining* ........ *7*
3.2. Representação do Conhecimento ........ 7
*3.2.1. Frames e Herança* ........ *8*
*3.2.2. Regras* ........ *13*
3.3. Perguntas/Respostas e Explicações ........ 17
3.4. Programação Dirigida a Dados ........ 22
3.5. A Linguagem KSL ........ 26

**4. CONCLUSÕES** ........ **31**

**5. REFERÊNCIAS BIBLIOGRÁFICAS** ........ **31**

**APÊNDICE A - SENTENÇAS EM KSL** ........ **32**

**APÊNDICE B - PREDICADOS DO *FLEX*** ........ **35**

**APÊNDICE C - EXEMPLO** ........ **48**

# 1. Introdução

Esse documento objetiva ser um guia para aprendizado e uso da ferramenta *Flex* para desenvolvimento de Sistemas Baseados em Conhecimento. Ele mostra onde o *Flex* se enquadra dentro do software LPA WIN-PROLOG 3.300 e como o *Flex* se integra com a linguagem Prolog que se encontra no nível abaixo.

O diagrama visto na Figura 1 mostra algumas das ferramentas de programação do LPA-Prolog e como elas trabalham juntas. O ferramenta *Flex*, FLINT e Prolog++ são todos implementados em Prolog. A ferramenta *Flex* pode ser usada sozinha ou em conjunto com FLINT, que propicia suporte a inferência em lógica fuzzy. Prolog também proporciona os meios de acesso a base de dados ODBC submissas e facilidades de comunicação com outras linguagens de programação.

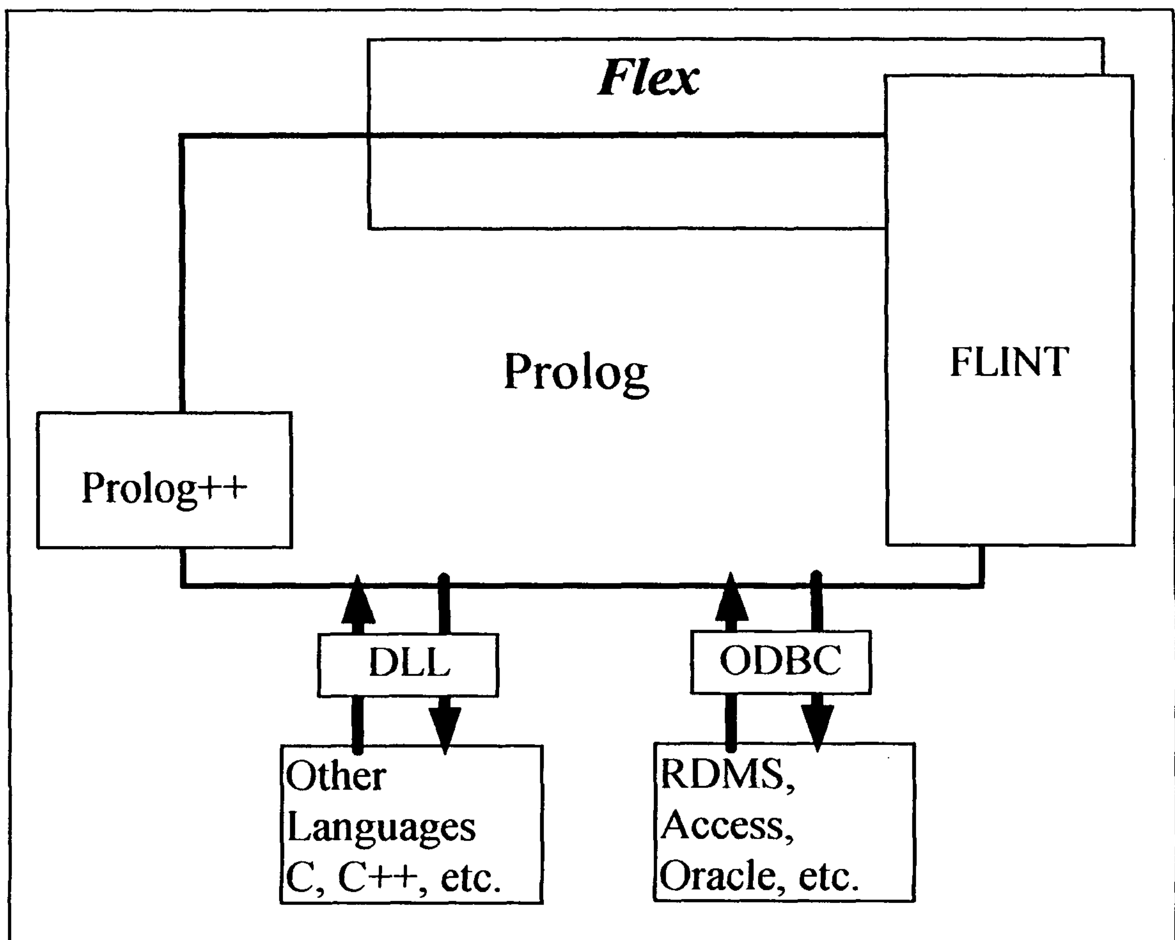



*Figura 1: O conjunto de produtos do LPA-Prolog*

*Flex* foi projetado especialmente para possibilitar o fácil desenvolvimento de Sistemas Baseados em Conhecimento (SBC).

O *Flex* é uma ferramenta expressiva para o desenvolvimento de SBC que suporta raciocínio baseado em frame com herança, programação baseada em regras e

procedimentos dirigidos a dados completamente integrados com um ambiente de programação lógica, e possui uma linguagem própria, *Knowledge Specification Language (KSL)*, que é semelhante ao inglês.

O *Flex* vai além de muitas shells para SBC, pois emprega uma arquitetura aberta e permite acessar, ampliar e modificar o seu comportamento através de uma camada de funções de acesso. Dessa forma, o *Flex* é referenciado como uma ferramenta de Inteligência Artificial (IA). A combinação de *Flex* e Prolog, isto é, uma ferramenta para SBC com uma poderosa linguagem de programação de IA, resulta num rico e versátil ambiente de desenvolvimento de SBC.

As principais características do *Flex* são:

- ferramenta para SBC totalmente funcional com suporte para várias construções, tais como: frames, regras, procedimentos, etc;
- uma linguagem similar ao inglês, a *Knowledge Specification Language (KSL)* para definição dessas construções;
- extensível e configurável através de uma camada de funções de acesso do Prolog.

Além dessa características principais, o *Flex* também possui:

- acesso ao Prolog e outras linguagens;
- portabilidade para a maioria das plataformas de hardware e software;
- suporte opcional para lógica fuzzy;
- interface ODBC opcional;
- acesso de alto nível à funções de interface gráficas de usuário.

O *Flex* possui uma arquitetura baseada em três níveis (Figura 2): o KSL, o suporte a predicados do *Flex* e o motor do *Flex*. O KSL situa-se no nível acima do Prolog e o suporte a predicados do *Flex* e o motor do *Flex* estão integrados diretamente dentro do Prolog. No nível mais alto está o KSL que permite desenvolver SBC numa forma que pode ser facilmente lida por não programadores. Essa linguagem é composta por sentenças que são usadas para descrever as construções do SBC. O suporte a predicados do *Flex*, no próximo nível, são funções de acesso do Prolog que suportam essas construções. Caso o KSL não proporcione exatamente o que se deseja, pode-se estender e configurar o comportamento das construções através dessa camada de suporte a predicados do *Flex*. Isso é necessário para manusear a herança de atributos, o processo forward chaining, programas dirigidos a dados, etc.

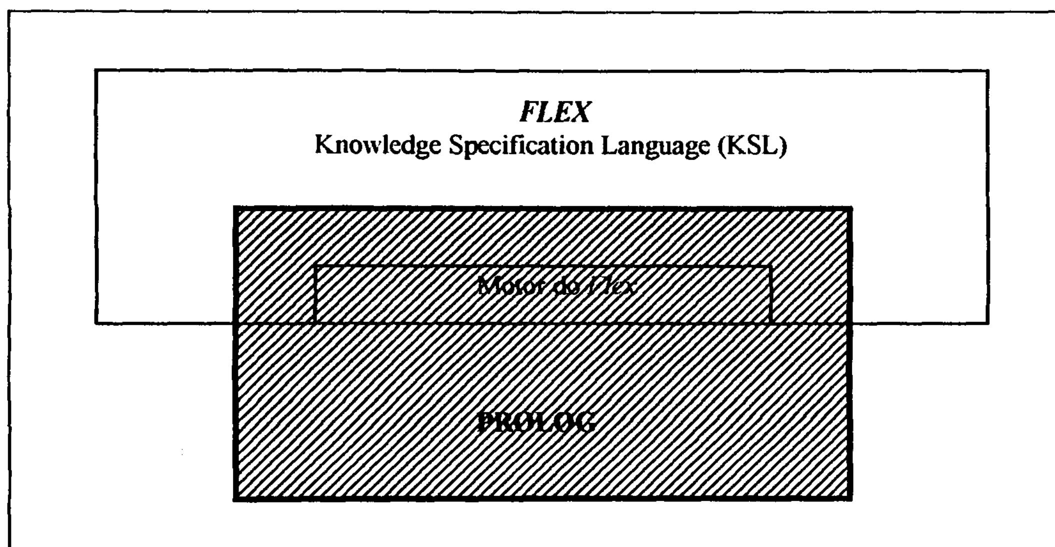



*Figura 2: A arquitetura do* Flex

O *Flex* suporta um conjunto de construções comumente usadas, que são:

- **hierarquia de frames**, onde os dados para um SBC podem ser representados como um número de ligações de frames, atributos e valores. Isso permite que os dados sejam definidos de uma maneira estruturada, onde os atributos comuns são herdados;

- **forward chaining** é um processo interativo pelo qual a cada estágio uma única regra é selecionada de um conjunto de regras de acordo com o estado corrente dos dados. A regra selecionada então gera um novo estado para os dados e todo o processo se repete;

- **backward chaining** é um processo pelo qual a verdade de uma meta é estabelecida pela prova da verdade de suas submetas. Esse processo pode proporcionar soluções alternativas quando requisitadas;

- **questões e respostas**, onde programas *Flex* podem questionar o usuário através de perguntas. A resposta à pergunta é então armazenada para referência futura;

- **programas dirigidos a dados**, onde programas são disparados automaticamente pela criação, acesso ou alteração de tipos específicos dos dados;

- **templates e sinônimos**, onde o programa KSL para um SBC pode ser construído para corresponder à linguagem apropriada para aquele domínio do SBC.

Esta nota didática está organizada da seguinte maneira: a seção 2 apresenta uma idéia geral de Sistema Baseado em Conhecimento. A seção 3 mostra a ferramenta *Flex*, bem como os mecanismos de inferência, a representação do conhecimento, perguntas/respostas e explicações, programação dirigida a dados e sua linguagem KSL. A seção 4 faz algumas conclusões e a seção 5 as referências bibliográficas.

## 2. Sistema Baseado em Conhecimento

Os Sistemas Baseados em Conhecimento (SBC) são de grande interesse nas pesquisas em Inteligência Artificial. Tais sistemas são programas de computador que resolvem ou ajudam a resolver problemas específicos, os quais geralmente requerem inteligência humana na sua solução [Arantes 95].

### *2.1. Estrutura Básica de um SBC*

O SBC pode ser dividido em três módulos principais [Rodrigues 93]:

- *Base de Conhecimento (BC)*: contém a modelagem do conhecimento específico do domínio de aplicação. Isto inclui asserções sobre o domínio de conhecimento, regras que descrevem relações neste domínio e, em alguns casos, heurísticas e métodos de resolução de problemas;
- *Motor de Inferência (MI)*: é responsável pelo processamento do conhecimento da BC, usando alguma linha de raciocínio. Contém as estratégias de inferência e controle;
- *Base de Dados (BD)*: consiste na área de trabalho do sistema, armazenando dados e conclusões intermediárias por ele obtidos.

A estrutura básica de um Sistema Baseado em Conhecimento pode ser visualizada na Figura 3.

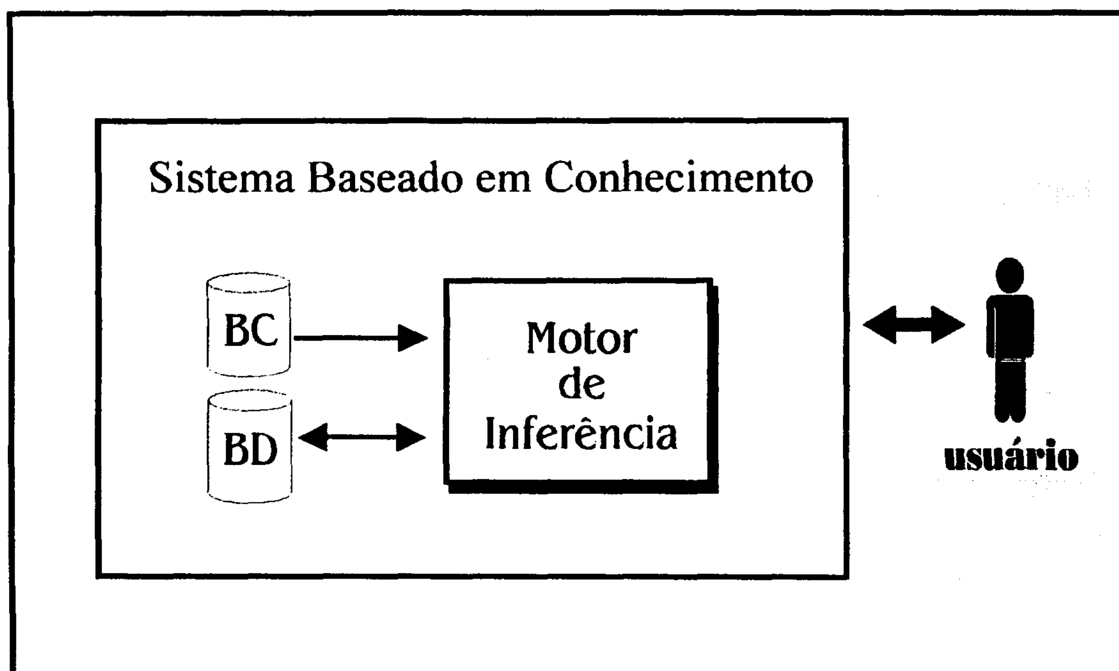



*Figura 3: Estrutura básica de um Sistema Baseado em Conhecimento*

## 2.2. Domínios de Conhecimento

O conhecimento e o processo de resolução de problemas são os pontos centrais no desenvolvimento de um SBC. Quanto ao conhecimento, verifica-se que existem domínios cujos problemas não são adequadamente solucionados através de SBCs. Por isso, uma análise do domínio e das características dos problemas a serem abordados é essencial quando se pretende desenvolver um sistema deste tipo.

Existe uma grande variedade de domínios de conhecimento comumente tratados em SBCs. Entre eles, pode-se citar:

- Interpretação: consiste na análise de dados para determinação de seu significado. O problema central desta classe é que, frequentemente, esses dados possuem ruído. Ex.: espectroscopia de massa, processamento de imagens, reconhecimento de fala, análise de circuitos elétricos, etc.;

- Diagnose: consiste no processo de determinação de falhas em um sistema, dado um conjunto de sintomas. Ex.: a maioria das aplicações médicas relativas à determinação de doenças em seres vivos, determinação de falhas de máquinas, etc.;

- Monitoramento: consiste no processo de observação contínua do comportamento de um sistema a fim de realizar ações quando alguma situação específica acontece. Ex.: monitoramento de centrais de energia nuclear, controle de tráfego aéreo, pacientes após ou durante cirurgia, etc.;

- Planejamento: consiste no processo de determinação da sequência de ações que devem ser realizadas para atingir uma dada meta. Ex.: planejamento de operações de robôs, experimentos em genética molecular, ações militares, planejamento de processos, etc.;

- Instrução: consiste no desenvolvimento de tutoriais para estudantes, com o objetivo de suprir pontos fracos nos seus conhecimentos. Ex.: atividades de ensino em diversas áreas;

- Configuração: consiste na configuração de sistemas, fornecendo estratégias a serem seguidas. Ex: configuração de sistemas de processamento de dados, reprojetos de circuitos visando atender a novas especificações, planejamento de experimentos de laboratório, etc...

## 2.3 Etapas de Desenvolvimento de um SBC

O desenvolvimento de um Sistema Baseado em Conhecimento envolve diversas etapas, como mostra a Figura 4:

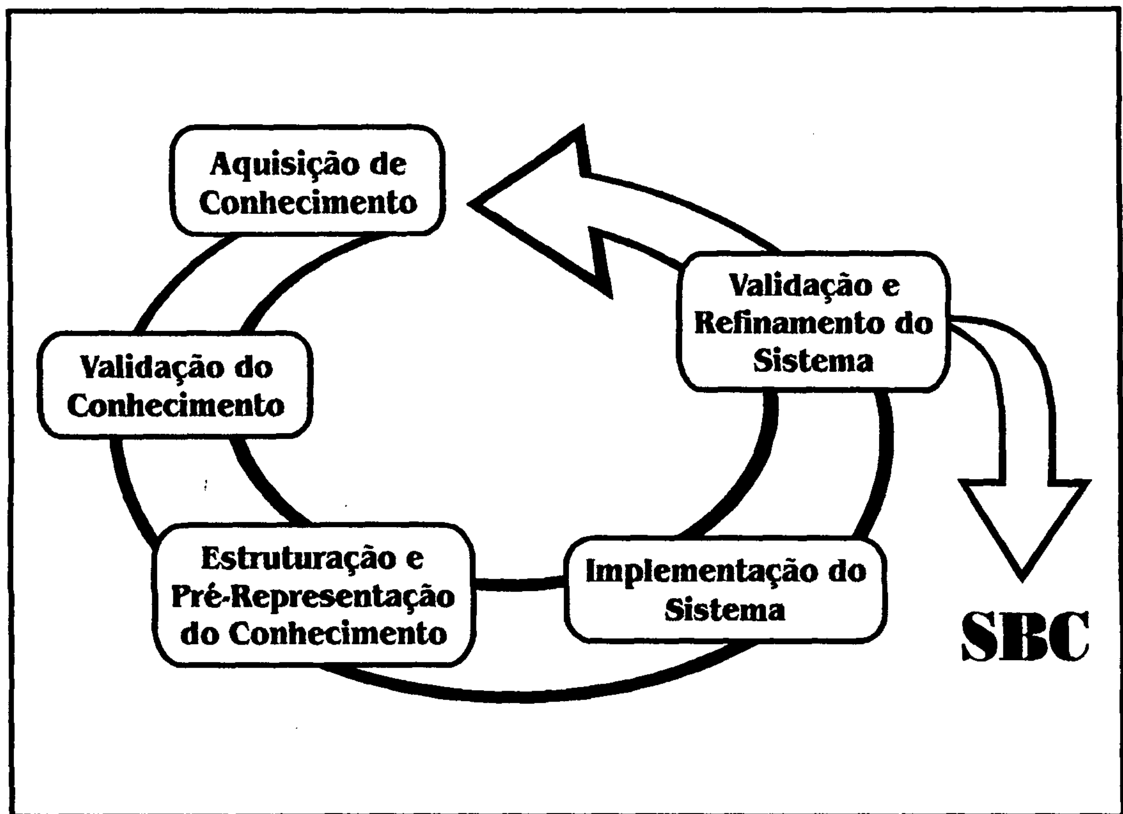



*Figura 4: Etapas de Desenvolvimento de um SBC*

- *Aquisição de conhecimento*: consiste na extração do conhecimento do especialista no domínio, bem como pesquisa em documentos, livros, handbooks, etc.;
- *Validação*: trata-se da validação do conhecimento com especialistas;
- *Estruturação e representação do conhecimento*: consiste em estruturar o conhecimento obtido na forma de regras, redes semânticas, frames, e outras formas de representação;
- *Implementação do sistema*: consiste na representação do conhecimento no computador, em linguagens de programação ou através de *shells* ou ferramentas que auxiliam a construção de SBCs;
- *Validação e Refinamento do sistema*: consiste em submeter o sistema a testes reais de forma a avaliar seu desempenho.

A seguir será mostrada a ferramenta *Flex* para auxiliar o desenvolvimento de um SBC.

# 3. Flex

Para detalhar a ferramenta *Flex* serão abordados os mecanismos de inferência disponíveis, os esquemas de representação do conhecimento utilizados, a forma de definir perguntas e respostas a explicações, o estilo de programação dirigido a dados e a linguagem para especificação de regras, frames e procedimentos.

## *3.1. Mecanismos de Inferência*

O *Flex* usa tanto forward quanto backward chaining como mecanismos de inferência.

### 3.1.1. Forward Chaining

A produção de regras usando forward chaining no *Flex* segue o formato clássico de regras *if-then*. Forward chaining é dirigido a dados e é adequado para problemas que envolvem muitos resultados possíveis para checar por backward chaining, ou onde o resultado final não é conhecido.

O motor de inferência forward chaining circula pelas regras correntes da agenda procurando por regras as quais as condições *if* possam ser satisfeitas, e seleciona uma regra para usar ou ser disparada executando sua parte *then*. Esse lado tipicamente afeta os valores dos dados, o que significa que um conjunto diferente de regras agora possuem suas condições satisfeitas.

O *Flex* estende a produção de regras clássica com uma opcional facilidade de explicação e dinâmico mecanismo de marcação de pontos para resolver conflitos durante a seleção de regras. Regras podem ter múltiplas conclusões ou ações (tanto positiva quanto negativa) em sua parte *then*.

### 3.1.2. Backward Chaining

A produção de regras usando backward chaining correspondem aos predicados Prolog, e são chamadas de **relation** no *Flex*. Elas possuem uma conclusão única que é verdade se todas as condições podem ser provadas. Backward chaining é comumente referida como dirigida a objetivos, e está muito ligada à noção de prova.

## *3.2. Representação do Conhecimento*

O *Flex* utiliza frames e regras para representar o conhecimento.

### 3.2.1. Frames e Herança

A hierarquia de frames permite que dados sejam armazenados de uma maneira abstrata em uma hierarquia aninhada com propriedades comuns que são automaticamente herdadas através da hierarquia. Isso evita a duplicação desnecessária de informações, simplifica o código e proporciona um sistema de fácil leitura e manutenção [Ávila 91].

Um frame é similar a um objeto e é uma estrutura de dados complexa que proporciona um modo útil de modelar objetos do mundo real. Os frames são análogos a registros em bases de dados, porém são mais poderosos e expressivos (Figura 3).

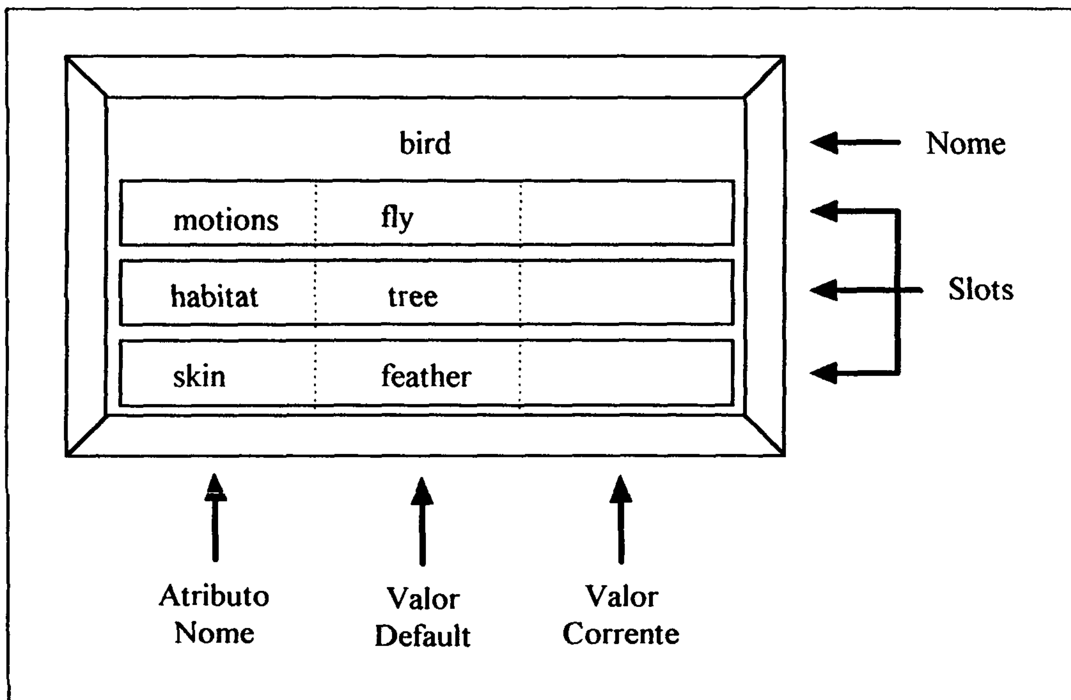



*Figura 5: Exemplo de um frame*

Cada frame possui um nome pelo qual ele é referenciado, detalhes de seu(s) frame(s) pai(s) e uma coleção de slots ou atributos que conterão valores ou ponteiros para valores. Valores dos slots podem ser explicitamente definidos ou implicitamente herdados de um de seus ancestrais.

O código KSL para o frame mostrado na Figura 3 poderia ser:

```
frame bird
    default skin is feather and
    default habitat is a tree and
    default motions are { fly } .
```

É importante notar a diferença entre valor default e corrente, pois algumas operações do *Flex* trabalham apenas nos valores correntes.

Quando um frame é declarado em KSL, o valor inicial dos atributos podem ser declarados. Entretanto, slots adicionais podem ser inseridos dinamicamente, como pode ser visto no exemplo a seguir.

```
frame jug
    default capacity is 15 and
    default contents is 0 .
action jug_update ;
    do the contents of the jug become 7.5 and
    the position of the jug become upright .
```

Valores default são normalmente associados a objetos ou classes gerais, enquanto que valores correntes são normalmente associados a instâncias específicas. Os valores default são usados somente quando o valor corrente não está disponível.

Os links entre os frames determinam a estrutura da hierarquia de frames. Cada link liga um frame pai ao seu filho. O frame filho pode ser entendido como uma especialização do frame pai, ou o frame pai como uma generalização do frame filho.

Um frame filho pode herdar valores (default ou correntes) de qualquer um dos seus frames pais, que por sua vez herdam de seus pais, e assim por diante. Isso permite a distribuição da informação sem duplicação.

Exemplo de declarações de frames em KSL:

```
frame animal .
frame carnivore .
frame mammal is an animal ;
    default blood is warm and
    default habitat is land .
frame rodent is a kind of mammal ;
    default habitat is sewer .
frame feline is a mammal, carnivore .
```

Instâncias aparecem como nós folhas na hierarquia de frames e podem ter apenas um pai. Instâncias podem conter apenas valores correntes em seus slots; elas não podem possuir valores default declarados. Por exemplo:

```
frame feline is a mammal, carnivore ;
    default legs are 4 .
frame cat is a feline ;
    default habitat is house and
    default meal is kit_e_kat .
instance sylvester is a kind of cat .
instance sammy is an instance of cat .
```

*Passando por cima da herança*

Nos exemplos dados até o momento, um frame filho herda automaticamente tudo de seus pais. Entretanto, pode-se querer herdar um atributo particular de um frame fora da hierarquia, ou de um frame particular da hierarquia, ou não herdar nada.

No *Flex*, um link especial pode ser definido de modo que permita que um atributo específico possa ser herdado de um frame específico. Por exemplo:

**instance** sammy **is an instance of** cat ;
  **inherit** comida **from** herbivore .

No *Flex*, também pode-se fazer que um atributo particular de um frame particular possa ser eliminado. Por exemplo:

**frame** cat
  **default** tail **is** furry .
**frame** manx **is a** cat
  **do not inherit** tail .

*Encadeamento de atributos*

Algumas vezes pode ser conveniente para um atributo ter seu próprio conjunto de valores, e nesse caso, slots podem conter ponteiros para outros frames ao invés de simples valores. Por exemplo:

**frame** address
  **default** city **is** 'London' .
**frame** employee
  **default** residence **is an** address .

Nesse exemplo, o valor ligado ao atributo residence do frame employee é um ponteiro para outro frame, chamado address.

Caso queira saber a cidade onde reside um empregado, pode-se referir a ela de três maneiras diferentes:

X **is the** residence **of** employee
**and** Y **is the** city **of** X

ou

Y **is the** city **of the** residence **of** employee

ou usando o operador **`s**

Y **is** employee**`s** residence**`s** city

Por exemplo, se criar uma nova instância empregado chamado phil, então será assumido que phil mora em London.

Para criar uma instância de empregado, de modo que ele more em Glasgow, poderia fazer:

**instance** phil **is an** employee **.**

Então, se phil não mora em London, mas em Glasgow, então pode-se mostrar isso através da seguinte diretiva:

**do the** city **of** residence **of** phil **becomes** 'Glasgow'

*Variáveis Globais*

Um uso especial de frame é para o armazenamento de variáveis globais. Elas são definidas como atributos de um frame especial chamado *global*. Por exemplo:

```
frame global
    default current_interest_rate is 10.3 .
```

Isso cria uma variável global, chamada current_interest_rate, que pode ser referenciada por qualquer proposição KSL.

Os valores das variáveis globais podem ser atualizados em tempo de execução. Variáveis globais também são usadas para armazenar a resposta de uma pergunta do *Flex* (veja seção 3.3).

*Herdando valores através da hierarquia de frames*

Em geral, informação flui na hierarquia da frames, a partir dos frames do topo para os das extremidades. Isso é realizado pela herança.

Sempre que existir um pedido de algum valor de um slot, o algoritmo de herança é automaticamente chamado. O lugar mais óbvio de se procurar primeiro é no próprio frame original, visto que ele pode ter o valor corrente e/ou o valor default para o atributo solicitado. Apenas se tal valor não existir localmente, será necessário procurar em outro lugar. Resta saber onde procurar, em que ordem procurar e quando se deve parar.

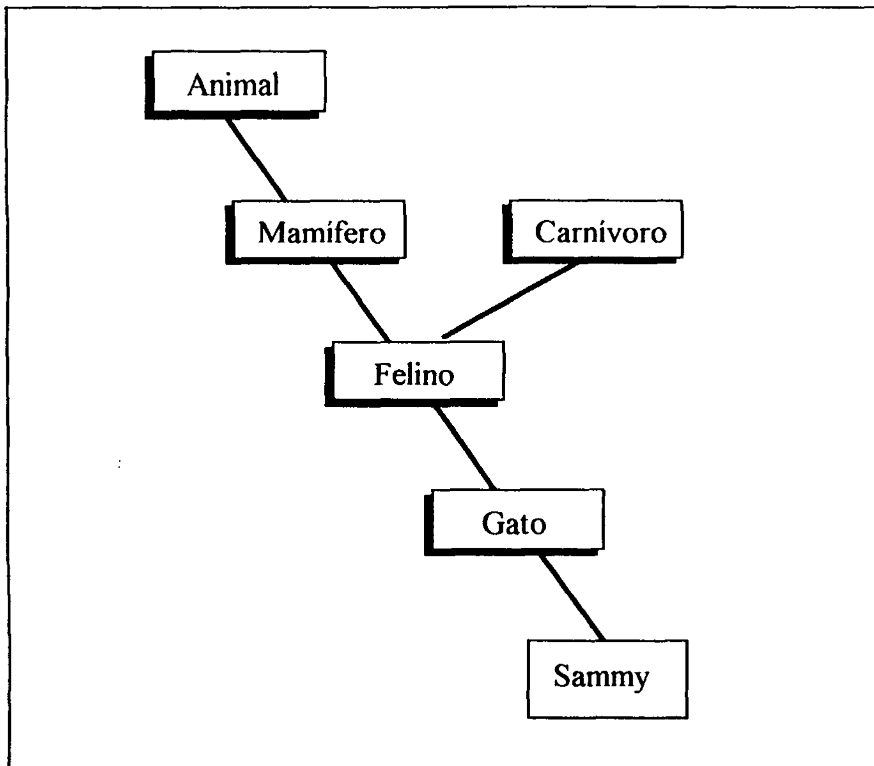



*Figura 6: Hierarquia de Frames*

Os frames a serem considerados, ao se procurar de quem deve ser herdado um atributo específico, são determinados pela hierarquia de frames e o atributo em questão (Figura 6). Sempre que existir lugares alternativos de busca, existirá inevitavelmente caminhos a serem observados.

Em ambos os métodos apresentados a seguir, a busca será da esquerda para a direita de acordo com a ordem dos pais. Uma busca em profundidade numa hierarquia de frames irá investigar todo o ramo de um ancestral antes de considerar um outro ramo ancestral alternativo. Já uma busca em largura irá visitar todos os frames de um particular nível de ancestrais antes de considerar qualquer outro de um nível superior. Uma característica importante desse método é que o valor retornado será do ancestral mais próximo possível do frame original.

A busca em profundidade é a estratégia default utilizada, e a mais eficiente, pois, nesse caso, ela mapeia estritamente o mecanismo de busca do Prolog.

Para grandes hierarquias de frames pode ser desejável limitar a quantidade de esforço usado em uma busca. Por essa razão, *Flex* possibilita que o número (um inteiro não negativo) máximo de níveis a serem buscados seja imposto. Por exemplo, se o limite for 0, não haverá herança; se for 2, apenas os pais e avós serão considerados. O valor default desse limite é 9 níveis.

### 3.2.2. Regras

Regras de produção são pontos centrais na tecnologia de SBCs, e proporcionam significados elegantes, expressivos e intuitivos da expressão do conhecimento.

Uma regra simples poderia ser *'se está chovendo então carregue uma sombrinha'*. Então, dado o fato que *'está chovendo'*, pode-se inferir ou derivar que *'carregue uma sombrinha'*.

Fatos podem ser vistos como um caso degenerado de uma regra, isto é, regras sem nenhuma pré-condição. Fatos podem ser armazenados em uma base de dados local, recuperadas de uma base de dados externa, obtidas do usuário através da interação de perguntas e respostas, armazenadas como varáveis globais, ou derivadas de outras regras.

Uma outra regra poderia ser *'se está chovendo então a grama está molhada'*. Agora, dado o mesmo fato que *'está chovendo'*, pode-se também inferir ou derivar que *'a grama está molhada'*, e armazenar isso como um fato, ou se *grama* for um atributo de algum frame, setar seu valor para *molhada*.

A coleção de todos os fatos é chamada de Base de Fatos. Em programação baseada em regras usa-se um motor de inferência para manipular regras e fatos e produzir novos fatos, o que significa que pode-se depois usar novas regras, até chegar a um resultado final.

A tentativa de provar as pré-condições das regras forward chaining pode envolver algumas avaliações backward chaining de suas sub-metas, isto é, pode haver um encadeamento backward para determinar o estado corrente do tempo. A integração desses dois mecanismos de inferência, em alguns casos, pode ser complexa. Note, que na falta de informação, execuções falham e condições em vigor são julgadas falsas. Isso é conhecido como suposição em mundo fechado.

Por exemplo:

A if B and C.
B if D.
C.
D.

onde, as duas primeiras proposições são regras, e as duas últimas são axiomas (ou fatos).

Regras em forward chaining devem ser indicadas pela palavra chave **rule**; e regras em backward chaining pela palavra chave **relation**.

Ambos os formatos se enquadram no estilo *if-then*, mas enquanto relações backward chaining permite apenas uma simples e positiva conclusão na parte *then*, não exite tal restrição nas regras forward chaining que pode ter múltiplas conclusões, qualquer delas pode ser tanto positiva quanto negativa por natureza.

Regras forward chaining sempre contém ações como parte de suas conclusões. Essa ações normalmente atualizam vários valores de dados (slots), o que significa que diferentes regras serão ou não disparadas no próximo ciclo de forward chaining.

Por outro lado, backward chaining geralmente procura estabelecer uma seqüência lógica de regras e fatos a fim de provar uma cláusula ou objetivo. Isso não envolve disparo de regras ou execução de ações, mas é uma natureza mais dedutiva.

*Prioridade das regras*

Em sistemas baseados em regras pode existir pontos de checagem, onde seleciona-se a regra a ser executada dependendo de sua prioridade. Colocar pesos nas regras é uma opção que pode auxiliar na tomada dessa decisão.

O peso das regras refletem sua importância relativa às outras regras do sistema. Sempre que duas ou mais regras forem simultaneamente aplicáveis, seus pesos relativos podem ser comparados para resolver o conflito.

A maioria dos sistemas são estáticos, com cada regra possuindo um peso específico. Quanto mais importante a regra, maior deve ser o seu peso. Por exemplo:

```
rule drink_tea
    if the hour is late
    then drink_a_cup_of_tea
    score 5 .
rule drink_beer
    if the fridge contains some beer
    and the weather is hot
    then drink_a_can_of_beer
    score 10 .
```

O *Flex* também permite um sistema dinâmico para definir essas prioridades, onde os pontos dados a uma regra não são fixos quando a regra é definida, mas dependem de alguma informação mutável, como pode ser visto no exemplo a seguir onde a prioridade da regra é definida em **score**.

```
rule empty_master_into_slave
    if the master is not empty
    and the slave`s contents > the master`s spare_capacity
    then fill_from( master, slave )
    score master`s contents plus slave`s contents .
```

*Anexando Explicações às Regras*

A parte final de uma regra de produção do *Flex* envolve a união de uma explicação adicional às regras. Essas explicações são utilizadas para explicar porque a regra foi disparada.

A explicação pode ser tanto um texto quanto um ponteiro para um arquivo no disco acessado em tempo de execução através de um *file browser*.

```
rule 'check status'
    if the applicant`s job is officer
    and the applicant`s age is greater than 65
    then ask_for_grading
    because The job grading affects the pension .
rule 'check status'
    if the applicant`s job is principal
    and the applicant`s age is greater than 65
    then check_principal_function
    browse file status .
```

*O Motor Forward Chaining*

O motor forward chaining é implementado como um programa Prolog. A ênfase da implementação está na simplicidade ligada a uma grande flexibilidade, além da extração do melhor desempenho possível.

*Conjunto de Regras*

Regras são agrupadas em conjunto de regras, e o mecanismo forward chaining é iniciado usando a palavra reserva do KSL **invoke ruleset**. Isso proporciona uma construção para formação de regras para as bases de regras e governar o motor forward chaining em termos de métodos de seleção de regras e atualização da agenda.

```
ruleset make_move
    contains corner_move, edge_move, centre_move .
action move ;
    invoke ruleset make_move .
```

*A Agenda de Regras*

A agenda determina as regras correntes disponíveis ao motor de inferência durante o forward chaining (inicialmente especificada pelo conjunto de regras). Ela pode incluir todas as regras do sistema ou apenas um subconjunto. A agenda pode ter duplicações.

Após uma regra ter sido selecionada e disparada, a agenda pode ser atualizada, e será essa agenda revisada que o motor de inferência usa como base para o próximo ciclo.

A agenda inicial pode ser especificada contendo todas as regras, uma lista de regras ou um grupo de regras.

```
ruleset everything
    contains all rules .
```

*Selecionando as Regras*

A parte vital de qualquer motor, quer seja forward ou backward chaining, é o método pelo qual as regras são selecionadas. O *Flex* proporciona três métodos de seleção de regras:

a) Primeiro a chegar primeiro a ser servido

Esse algoritmo de seleção escolhe a primeira regra da agenda que possui todas as condições satisfeitas (a parte *if*). Isso significa que a ordem das regras na agenda é muito importante. O seu maior benefício é eficiência: primeiro que não é necessário considerar todas as regras da agenda, e segundo que não há necessidade de pilha para armazenar um conjunto temporário de regras.

A maior desvantagem é a falta de controle ao escolher as regras, já que a única opção de controle disponível é a reordenação da agenda. Isso, entretanto, nem sempre reflete a importância relativa das regras.

Esse é o algoritmo de seleção default para conjunto de regras. Por exemplo:

```
ruleset mover
    contains push, pull, lift .
ruleset make_tea
    contains all rules ;
    select rule using first come first served .
```

b) Resolução por conflito

Esse é um mais sofisticado e computacionalmente caro esquema de seleção, onde a “melhor” regra é sempre selecionada. O conflito existente quando mais de uma regra pode ser disparada é resolvido escolhendo-se a regra de mais alta prioridade (**score**).

Esse esquema certamente permite um maior controle sobre a fase de seleção das regras, mas isso eleva o custo. A “melhor” regra só pode ser escolhida se todas as regras forem consideradas. Por exemplo:

```
ruleset mover
    contains push, pull, lift
    select rule using conflict resolution .
```

c) Resolução por conflito com um limiar.

Esse esquema de seleção consiste em acrescentar limiares no esquema anterior, que oferecerá a princípio um esquema de resolução por conflito, mas que parará assim que um candidato, com pontos maior que o limiar, for encontrado. Exemplo:

```
ruleset make_tea
    contains all rules ;
    select rule using conflict resolution
                    with threshold 6 .
```

*Atualizando a Agenda*

Ao final de cada ciclo do motor, a agenda pode ser atualizada de acordo com o nome da regra que foi disparada. O default é deixar o conjunto de regras exatamente como está, de modo que as regras sejam sempre consideradas na mesma ordem.

Existem vários procedimentos pré-definidos para atualizar a agenda, e ainda há a possibilidade de se desenvolver em *Flex* outros algoritmos. As opções para atualização do conjunto de regras são as seguintes:

- a cada vez que uma regra é disparada, ela pode ser removida do conjunto de regras;
- a cada vez que uma regra é disparada, ela pode ser movida para o início ou final do conjunto de regras corrente;
- a agenda pode ser vista como uma fila circular;
- o conjunto de regras pode ser atualizado removendo-se da agenda qualquer regra nas quais as condições, no último ciclo do motor forward chaining, não foram satisfeitas;
- finalmente, uma "rede" de regras pode ser especificada possibilitando a agenda trocar entre diferentes grupos de regras para cada ciclo.

É também possível desabilitar ou habilitar certas regras.

### *3.3. Perguntas/Respostas e Explicações*

*Flex* possui um subsistema pré-definido de perguntas (**question**) e respostas (**answer**) que permite às aplicações finais questionar o usuário por entradas adicionais

através de janelas de diálogos interativas. Essas janelas podem ser as pré-definidas. ou sofisticadas janelas construídas usando as facilidades de manuseio das próprias janelas do Prolog e depois ligadas ao subsistema de perguntas e respostas.

*Flex* possui um sistema pré-definido de explicação que suporta explicações *como* e *porque*. Explicações podem ser ligadas tanto a regras quanto a perguntas, usando simples cláusulas **because**.

A maioria das aplicações de SBC possuem alguma comunicação com o usuário. No *Flex*, isso é feito utilizando perguntas pré-definidas. Essas perguntas podem invocar uma escolha de um menu, uma entrada de informações pelo teclado, ou qualquer conjunto de operações que requerem uma reação do usuário.

Interações mais sofisticadas com o usuário podem ser definidas usando as facilidades do Graphical User Interface (GUI) do sistema Prolog básico e depois podem ser facilmente integradas com o subsistema de perguntas e respostas do *Flex* usando relações ou ações.

*Seleção por Menu*

Nesse tipo de perguntas é apresentado ao usuário uma coleção de opções e é oferecido a opção de se fazer única ou múltiplas seleções. Por exemplo:

```
question main_course
    Please select a dish for your main course ;
    choose one of steak, 'lamb chops', trout, 'dover sole' .

group main_courses
    steak, 'lamb chops', trout, 'dover sole' .
question main_course
    Please select a dish for your main course ;
    choose one of main_courses .

question vegetables
    Please select vegetables to accompany your main course ;
    choose some of potatoes, leeks, carrots, peas .
```

*Armazenando as Respostas*

Sempre que uma pergunta é feita, a "resposta" é armazenada como o valor de uma variável global de mesmo nome da pergunta. Por exemplo:

```
action get_main_course ( Main, Vegs ) ;
    do ask main_course and
    ask vegetables and
    check that Main is main_course and
    check that Vegs is vegetables .
```

get_main_course/2 pode ser chamado por um programa Prolog, ou um procedimento *Flex*, ou executado por uma linha de comando do Prolog, por exemplo:

```
?- get_main_course ( M, V ) .

M = steak, V = [leeks, carrots, peas]
```

```
action show_main_course ;
    do ask main_course and
    ask vegetables and
    write ( main_course ) and
    nl and
    write ( vegetables ) .
?- show_main_course .

steak

[leeks, carrots, peas]
```

*Entrada pelo Teclado*

O dado de entrada pode ser restringido a um texto (nome), um número em ponto flutuante, um inteiro, ou um conjunto desses itens como, por exemplo:

```
question name_of_applicant
    Please enter your name ;
    input name .

question height_of_applicant
    Please enter your height (in metres) ;
    input number .

question address1_of_applicant
    Please enter your house number ;
    input integer .

question address5_of_applicant
    Please enter your city and post code ;
    input set .
```

Alguns sistemas podem necessitar de uma entrada diferente de nome, número, inteiro ou conjunto desses objetos. Parar isso é definido um programa Prolog ou uma relação *Flex* para se validar uma resposta, que é indicado pela palavra chave **such that**.

```
question yes_or_no
    Please answer yes or no ;
    input K such that yes_or_no_answer ( K ) .

relation yes_or_no_answer ( yes ) .

relation yes_or_no_answer ( no ) .
```

### *Entrada Personalizada*

O conjunto de perguntas padrões inevitavelmente não irão cobrir todas as possíveis situações. Por essa razão, o *Flex* permite perguntas personalizadas nas quais o programador pode especificar como obter a resposta e qual a forma que a pergunta deve ter. A responsabilidade é totalmente do programador apresentar a pergunta ao usuário (por exemplo, criar uma caixa de diálogo) e retornar a resposta apropriada. Isso é indicado, em KSL, pela palavra reservada **answer**.

```
question my_question
    answer is K such that ask_my_question ( K ) .
```

Nesse caso, nenhuma caixa de diálogo pré-definida será apresentada, mas uma chamada será feita à ask_my_question/1, que deve estar definida, no *Flex*, como uma ação, uma relação ou um predicado Prolog. Será feita uma pergunta, criada uma caixa de diálolgo, se encessáio, e retornado o valor na varável K.

### *Perguntas Default*

Ao desenvolver uma aplicação, é muitas vezes útil adiar a implementação exata das perguntas até algum estágio mais adiante. Durante esse processo de desenvolvimento, o *Flex* permite declarar uma pergunta default que é usada na ausência de uma definição específica. O nome da pergunta default é catchall.

```
question catchall
    Please enter data ;
    input name .
```

Sempre que uma pergunta, que não possui definição, for feita, a definição catchall é usada em seu lugar. No exemplo acima, a pergunta default será do tipo **name**.

### *Explicações às Perguntas*

Além da forma da pergunta, pode-se, opcionalmente, acrescentar uma explicação a qualquer um dos tipos padrões de pergunta usando a cláusula **because**.

A explicação pode ser um texto a ser mostrado, ou um nome de um arquivo a ser carregado. As explicações são apresentadas sempre que o usuário final as requisitar (normalmente existe um botão Explain nas caixas de diálogo das perguntas pré-definidas).

```
question main_course
    Please select a dish for your main course ;
    choose from steak, 'lamb chops', trout, 'dover sole' ;
    because The main course is an integral part of a meal .

question headache
    Have you got a headache ? ;
    answer is K such that yes_or_no_answer ( K ) ;
    browse file medical_symptoms .

question headache
    answer is K such that
            write ( 'Have you got a headache ?' ) and
            yes_no_question ( K, medical_symptoms ) .

action yes_no_question ( K, File ) ;
    do write ( 'Please type Y or N or ESC' )
    and repeat
    and get0 ( Byte )
    and yes_or_no_check ( Byte, K, File )
    and ! .

relation yes_or_no_check (  89, yes, File ) .
relation yes_or_no_check ( 121, yes, File ) .
relation yes_or_no_check (  78,  no, File ) .
relation yes_or_no_check ( 110,  no, File ) .
relation yes_or_no_check (  26,   _, File )
    if browser ( File )
    and fail .
```

*Invocando Perguntas*

Existem dois procedimentos, ask/1 e answer/2, para ativar perguntas. Eles são refletidos em KSL como a diretiva

**ask** <nome da pergunta>

e como o termo

**answer to** <nome do pergunta>

respectivamente. Esses procedimentos pré-definidos se comportam diferente em tempo de execução. Sempre que houver um pedido para **ask**, essa pergunta será sempre

imediatamente feita. Entretanto, um pedido para **answer to** somente será feito se essa pergunta já não foi anteriormente feita.

```
action decide_meal ;
    do ask main_course
    and ask vegetables .

rule prescription1
    if the answer to headache is yes
    and the answer to pregnant is no
    then prescribe ( paracetamol ) .

rule prescription2
    if the answer to headache is yes
    and the answer to pregnant is yes
    then prescribe ( nothing ) .
```

### *3.4. Programação Dirigida a Dados*

*Flex* oferece procedimentos especiais os quais podem ser ligados a coleções de frames (frames individuais ou slots em frames). Esses procedimentos permanecem adormecidos até que sejam ativados pelo acesso ou atualização de uma estrutura particular à qual eles foram ligados. Existem quatro diferentes tipos de procedimentos dirigidos a dados disponíveis no *Flex*, que são: **launches**, **demons**, **watchdogs** e **constraints**.

O sistema de frames do *Flex* pode ser usado para a representação de dados e conhecimento. Na programação dirigida a dados, ao invés de programar o fluxo de controle ou lógica dos sistema, procedimentos são ligados a frames individuais ou grupos de frames.

Esses procedimentos ficam dormentes e são ativados sempre que existe uma requisição de atualização, acesso ou criação de uma instância do frame ou slot aos quais eles estão ligados.

Existem quatro tipos de procedimentos dirigidos a dados: **launches**, **watchdogs**, **constraints** e **demons.** O diagrama da Figura 7 mostra quando e onde os vários procedimentos são invocados.

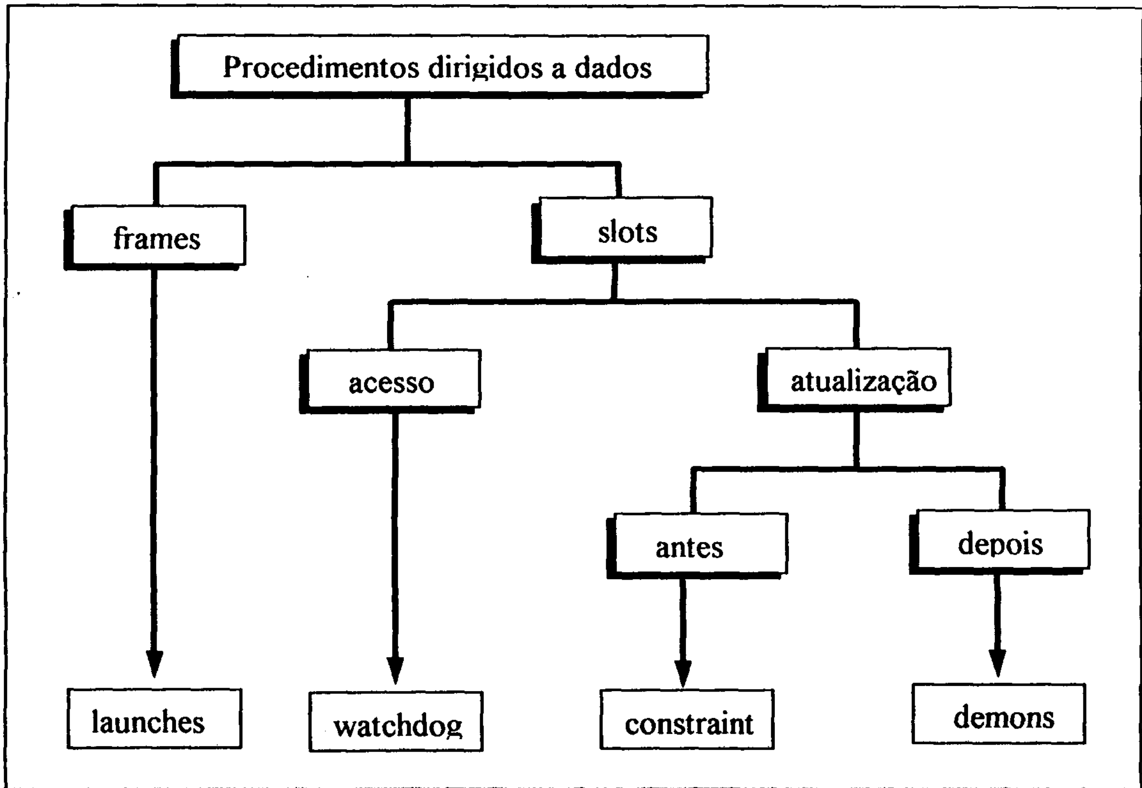



*Figura 7: Procedimentos Dirigidos a Dados*

<u>*Launches*</u>

O procedimento launch é ativado sempre que existir uma requisição para criar uma instância do frame ao qual ele está ligado.

O procedimento launch possui três partes principais:

- contexto: um teste para ver se certas condições são válidas;
- teste: um teste para ver se certas condições são válidas;
- ação: uma série de comandos a serem desenvolvidos.

Uma ação só será invocada se o teste for bem sucedido. O launch é chamado depois que a instância foi criada. Por exemplo:

```
frame employee
    default sex is male .

launch new_employee
    when Person is a new employee
    and sex of Person is male
    then male_enrolment_question ( Person ) .

instance dave is an employee .
```

*Constraint*

Um constraint está ligado a um slot individual, e é designado a restringir o conteúdo de um slot a valores válidos. Ele é ativado sempre que o valor corrente do slot é atualizado, e a ativação ocorre imediatamente antes da atualização.

O constraint possui três partes principais:

- contexto: um teste para ver se certas condições são válidas;
- check: um teste para ver se a atualização é válida;
- erro: uma série de comandos a serem efetuados para uma atualização inválida.

Um check só será feito se o contexto for válido. Se o check for bem sucedido, então a atualização é permitida, caso contrário os comandos de erro são chamados e a atualização não é permitida. Por exemplo:

```
frame jug
    default contents are 0 and
    default capacity is 7 .

instance jug1 is a jug .

constraint check_contents
    when the contents of Jug changes to X
    and Jug is some jug
    then check that number ( X )
    and X =< Jug`s capacity
    otherwise write ( 'contents not update' )
    and nl .
```

Note o uso da variável local *Jug* que irá unificar com qualquer frame ou instância que possui, nesse caso, o atributo *contents*.

*Demon*

Um demon está ligado a um slot individual. Ele é ativado sempre que o valor corrente do slot é atualizado, e a ativação ocorre imediatamente após a atualização.

O demon possui duas partes principais:

- contexto: um teste para ver se certas condições são válidas;
- ação: uma série de comandos a serem executados.

Um demon pode ser construído de modo que ele dispara somente para dados valores e/ou somente sob certas circunstâncias. Por exemplo:

```
frame kettle
    default temp is 0 .

instance kettle1 is a kettle .

demon kettle_boiling
    when the temp changes to T
    and T is greater than 100
    then make_steam
    and nl
    and whistle .
```

*Watchdog*

Uma watchdog está ligada a um slot individual. Ele é ativado sempre que o valor corrente do slot é acessado.

O watchdog possui três partes principais:

- contexto: um teste para ver se certas condições são válidas;
- check: um teste para ver se o acesso é válido;
- erro: uma série de comandos a serem efetuados se o acesso for inválido.

Um check só será feito se o contexto for válido. Se o check for bem sucedido, então o acesso é permitido. Caso contrário, os comandos de erro são chamados e o acesso é negado.

A watchdog pode ser usada para checar o direito de acesso a um atributo de um frame. Ela é chamada sempre que houver um pedido pelo valor corrente (não o valor default) do slot. Por exemplo:

```
frame bank_account
    default balance is 0 .

frame current_user
    default name is ' ' and
    default access is 0 .
```

```
watchdog account_security
    when the balance of Account is requested
    and Account is some bank_account
    then check current_user's access is above 99
    otherwise write ( 'Balance access denied to user ' ) and
    write ( current_user's name ) .
```

### *3.5. A Linguagem KSL*

*Flex* possui sua própria linguagem para definir regras, frames e procedimentos. Essa linguagem é bem expressiva, fácil de programar e fácil de ler. A KSL permite que projetistas escrevam proposições simples e concisas sobre o mundo do especialista, o qual pode depois ser entendido e mantido por não programadores. O conhecimento é explicitamente fixado de um maneira natural e simples fazendo bases de conhecimento virtualmente auto-documentadas. A KSL suporta expressões e funções matemáticas, booleanas e condicionais junto com conjuntos de abstrações; além disso, a KSL é estendida a sinônimos e templates. Por suportar tanto varáveis lógicas quanto globais nas regras, *Flex* evita desnecessária duplicação de regras e necessita de menos regras do que a maioria dos Sistemas Baseados em Conhecimento.

*Termos em KSL*

Os termos e tokens em KSL, essencialmente, seguem a sintase de Edinburgh, com alguns realces.

Existem cinco tipos de tokens de Edinburgh: pontuação, número, átomo, string e variável. Acima dos tokens de Edinburgh foram construídos os conceitos de nome e valor do KSL.

a) Comentário: qualquer texto entre os símbolos /* e */ são tratados como comentário e ignorado pelo compialdor do *Flex*. O símbolo % indica o inicio do comentário que irá até o final da linha.

b) Pontuação: são sempre consideradas como um token de separação (a não ser que esteja entre aspas), e podem ser: ( ) [ ] { } | ! ; ,

c) Número: são tanto inteiros quanto ponto flutuantes. Exemplos: 211327 -32768 0 2.34 10.3e99 -0.81

d) Átomo: podem ser de três tipos: alfanuméricos (letras minúsculas (a-z) seguidas por zero ou mais caracteres alfabéticos (a-z, A-Z ou _) ou dígitos (0-9)), símbolos (seqüência de caracteres, como *, >, #,etc., com exceção dos dígitos, caracteres alfabéticos e pontuação) ou citação (qualquer seqüência de caracteres entre aspas simples). Exemplos: apple aPPLE h45j apple_cart & >= **/ 'Apple' 'the*^' 'home''s'

e) String: qualquer seqüência delimitada por aspas duplas. Exemplos: "A boy"  "h2*"

f) Variável: é uma letra maiúscula (A-Z) ou underscore ( _ ) seguido de uma seqüência de zero ou mais caracteres alfabéticos (a-z, A-Z ou _) ou dígitos (0-9). Um underscore sozinho é uma variável anônima. Exemplos: X  Apple  X_Apple_23

g) Nome: é qualquer átomo de citação ou qualquer átomo que não for uma palavra reservada (isto é, um átomo que não aparece no glossário da KSL). Exemplos: brick brick23  'The'  'the brick'

h) Valor: é qualquer número, string ou nome. Exemplos: 9821  -0.81  "the lake" 'the'

*Objetos em KSL*

Objetos em KSL são construções que correspondem a entidades no seu domínio particular. Eles vão desde variantes que podem mudar de valor com o tempo (por atribuição), até conjunto de abstrações que descreve uma coleção de objetos.

a) Variantes: podem mudar de valor com o tempo (isto é, como uma variável). Existem duas formas de variantes: variável global (que é qualquer nome, opcionalmente precedida pelo determinante **the**) ou slot (especificado pelo nome de um atributo e o nome do frame, utilizando **of** ou **`s**). Ambos possuem valores associados a eles. Exemplos:

    staff
    **the** staff
    'today''s temperature'
    size **of the** collar
    **the** collar**`s** size
    **the** colour **of** money
    money**`s** colour

b) Esquema Variante: é uma generalização dos slots no qual o nome do frame é apenas indiretamente referenciado. Ou seja, o variante é um atributo específico de algum frame não especificado. Exemplos:

    atributo **of anything**
    atributo **of something**
    atributo **of anybody**
    atributo **of somebody**
    atributo **of some** frame
    atributo **of any** frame
    atributo **of some instance of** frame
    atributo **of any instance of** frame

c) Variantes Complexas: esse conceito pode ser extendido com níveis arbitrários de indireção, também referenciado como cadeia de atributos. O valor atribuido a um atributo de um frame pode ser o nome de um outro atributo ou frame. Exemplos:

**the** city **of** ( **the** employee_address **of the** employee )
city **of** employee_address **of some** employee
**some** employee's employee_address's city

Conjuntos: existe duas formas de conjuntos em KSL: explícito (descrição de cada um dos elementos individuais em um conjunto. Os elementos podem ser separados por vírgula, **or** ou **and**; e o conjunto é delimitado por { }.) ou implícito (estabelece como encontrar ou computar cada elemento sem ter que mencionar cada um.).

d) Termos Gerais: inclui nomes, valores, variáveis, variantes, esquemas variantes e conjuntos.

*Expressões Aritméticas*

Expressões aritméticas são formadas por operadores binários ( *, +, -, / e ^ ) e operador unário ( - ) sobre os termos. As palavras **plus, minus, times, divided by** e **to the power of** também podem ser usadas.

A precedência aplicada usualmente aos operadores é o operador ^ maior que a dos operadores * e / que é maior que a dos operadores + e - . Associatividade, quando existe mais de um operador com a mesma precedência na expressão, será sempre da esquerda para a direita.

A precedência dos operadores podem ser escondidas com a utilização de parênteses nas expressões. Exemplos:

1 + 2 * 3 - 4 / 5 ^ 6
**the** number **of** managers **plus the** number **of** secretaries
( 1 - 2 ) * ( 3 + 4 )
( **its** temp **times its** volume ) **to the power of** 2

*Fórmulas em KSL*

As fórmulas em KSL são usadas para estabelecer relação entre objetos KSL. Isso recai em duas áreas distintas: condição ou diretivas.

### Condição

Uma fórmula condição é usada para testar o estado corrente, por exemplo, variáveis globais, frames e fatos.

Condição também testa a existência de um variante ou compara o valor de duas expressões; uma condição pode ser também uma chamada a procedimento.

Condições podem ser do tipo: comparação de igualdade (=, **is**, **are** ou **is equal to**), teste de existência (**known** ou **unknown**), comparação direta (>, <, =<, >=, **greater than**, **greater than or equal to**, **above**, **at or above**, **less than**, **less than or equal to**, **below**, **at or below**, que podem estar precedidas por **not**), comparação relativa (**according to**), membros de um conjunto (**include**, **includes**, **included in**, **does not include**, **do not include**), conjunção e disjunção (**and** e **or**), ou mudança de contexto (**check[that]**).

### Diretivas

Diretivas são usadas para mudar o estado corrente para alguns novos estados, onde um estado consiste de variáveis globais, frames, fatos e exceções.

A mudança de variáveis globais e frames são atribuições, enquanto que a adição e remoção de fatos e exceções são classificadas como manutenção da base de dados.

As diretivas podem ser: atribuição direta, incremento e decremento, membros de um conjunto, novas instâncias, manutenção da base de dados, perguntas, ou chamada a procedimento.

## *Estruturas de Controle em KSL*

As estruturas de controle no *Flex* pertencem a duas categorias: condicional e repetição.

a) If-Then-Else

> **if** condição(s)<br>
> **then** diretiva(s)<br>
> **else** diretiva(s)<br>
> **end if**

b) Repeat-Until Loops

> **repeat** diretiva(s)<br>
> **until** condição(s)<br>
> **end repeat**

c) While-Do Loops

**while** condição(s)
**do** diretiva(s)
**end while**

d) For Loops

**for** condição(s)
**do** diretiva(s)
**end for**

e) For Loops Estendidos

**for** contador **from** expressão1 **to** expressão2 **step** expressão3
**do** diretiva(s)
**end for**

## 4. Conclusões

Os Sistemas Baseados em Conhecimento consistem em um importante avanço da tecnologia no que diz respeito à automatização, ainda que limitada, do raciocínio humano. As ferramentas que auxiliam a construção de SBCs permitem a otimização de esforços no desenvolvimento de tais sistemas, uma vez que a implementação do sistema é facilitada. Dessa forma pode-se destinar mais tempo as etapas de aquisição, estruturação e validação do conhecimento.

O *Flex* que é uma ferramenta para auxiliar o desenvolvimento de SBC vai além de uma shell normal, pois emprega uma arquitetura aberta e permite acessar, ampliar e modificar o seu comportamento através de uma camada de funções de acesso. A combinação de *Flex* e Prolog resulta num rico e versátil ambiente de desenvolvimento de SBC.

Além disso, a ferramenta *Flex* pode ser combinada com a ferramenta FLINT, que propicia suporte a inferência em lógica fuzzy incrementado assim o poder da ferramenta. A combinação de *Flex* com o LPA-Prolog proporciona os meios de acesso a base de dados ODBC submissas e facilidades de comunicação com outras linguagens de programação.

## 5. Referências Bibliográficas


[Arantes 95] Arantes, A. C. N.; Caulkins, C. W.; Oliveira, M. A. de; *Desenvolvimento de um Sistema Baseado em Conhecimento para Configuração de Tornos Mecânicos* , Relatório Técnico, Universidade de São Paulo - Campus São Carlos, 1995.

[Ávila 91] Ávila, B. C. *Representação de Conhecimento Utilizando Frames.* Tese de Mestrado, ICMSC-USP, 1991.

[Luger 89] Luger, G. F.; Stubblefield, W. A. *Artificial Intelligence and the Design of Expert Systems.* The Benjamin/Cummings Publishing Company Inc., 1989.

[Rich 93] Rich, E.; Knight, K. *Inteligência Artificial. (2da Ed.)* Mc Graw-Hill, São Paulo, 1993.

[Rodrigues 93] Rodrigues, S. R. *Criação e Aplicação de Módulos Automáticos para o Planejamento de Processo Assistido por Computador em Soluções Híbridas de Planejamento.* Tese de Doutorado, EESC-USP, 1993.

[Vasey 96] Vasey, P. *flex Expert System Toolkit.* Version 1.2, Edition 4, 1996.

[Westwood 96] Westwood, D. *A Guide to the Flex Expert System Toolkit-Examples.* 1996.

## Apêndice A - Sentenças em KSL

Aqui estão descritas as sentenças válidas na qual objetos e fórmulas podem ocorrer. Elas constituem o que pode e não pode ser estabelecido em uma programa KSL.

Uma sentença KSL começa com uma das palavras chaves do KSL que seguem, e teminam com um espaço e um ponto final.

- action

  **action** nome ;
  do diretiva(s) .

- constraint

  **constraint** nome
  **when** atributo **changes**
  **from** expressão1
  **to** expressão2
  **and** condição1(s)
  **then check that** condição2(s)
  **otherwise** diretiva(s) .

- data

  **data** nome
  **do** diretiva(s) .

- demon

  **demon** nome
  **when** atributo **changes**
  **from** expressão1
  **to** expressão2
  **and** condição(s)
  **then** diretiva(s) .

- do

  **do** diretiva(s) .

- frame

  **frame** Nome
  **default** Atributo1 **is** Valor ;
  **inherit** Atributo2 **from** Frame .

- function

  **function** nome = expressão .

  **function** nome = variável
  **where** condição(s) .

**function** nome =
    **if** condição(s)
    **then** expressão1
    **else** expressão2 .

- group

    **group** nome
        primeiro_item, segundo_item, ..., último_item .

- instance

    **instance** Instância **is a** Frame ;
        Atributo **is** Valor .

ou

    **instance** Instância **is an** Frame
    **instance** Instância **is a kind of** Frame
    **instance** Instância **is na instance of** Frame

- launch

    **launch** nome
        **when** instância **is a new** Frame
        **and** condição(s)
        **then** diretiva(s) .

ou

    **launch** nome
        **when** instância **is a new instance of** Frame
        **and** condição(s)
        **then** diretiva(s) .

- question

    **question** nome
        texto da pergunta ;
        **choose from** menu itens ;
        **because** explicação .

ou

    **question** nome
        texto da pergunta ;
        **choose some of** menu itens .

ou

    **question** nome
        texto da pergunta ;
        **choose one of** menu itens .

- relation

    **relation** nome ;
        **if** condição(s) .

- rule

    **rule** nome
        **if** condição(s)
        **then** diretiva(s) ;
        **because** explicação ;

**score** score .

- ruleset

  **ruleset** Nome

  **contains** regra(s) ;
  **initiate by doing** diretiva(s) ;
  **terminate when** condição(s) ;
  **select rule using** seleção_regra ;
  **update ruleset** atualização_agenda ;
  **when a rule misfire do** diretiva(s) .

- synonym

  **synonym** nome_a_ser_substituído expressão_substituta .

- template

  **template** substituição

  template_positiva ;
  template_negativa .

- watchdog

  **watchdog** nome

  **when** atributo **is requested**
  **and** condição1(s)
  **then check that** condição2(s)
  **otherwise** diretiva(s) .

## Apêndice B - Predicados do *Flex*

Aqui serão descritos os predicados Prolog fornecidos pela ferramenta *Flex*. Eles são fornecidos para acessar a funcionalidade do *Flex* através do Prolog ao invés de usar a KSL, ou pode-se escolher escrever alguma aplicação em Prolog usando esses predicados ao invés do seu equivalente em KSL. Para algumas operações, como processamento de lista, é algumas vezes mais fácil e eficiente usar os predicados Prolog do *Flex* num programa Prolog do que usar construções KSL. Pode-se, livremente, misturar códigos escritos usando predicados Prolog do *Flex* e códigos escritos usando KSL: frames, ações e relações podem ser acessadas do Prolog como programas Prolog podem ser acessados do KSL.

Os predicados neste apêndice estão organizados de acordo com sua categoria. Portanto, devido ao fato de que alguns predicados aparecem em mais de um grupo, eles serão repetidos.

**Constraints**

isa_constraint/8

isa_constraint(?Name, ?Attribute, ?Frame, -Old, -New, -Context, -Check, -Action): recupera as características de um *constraint*.

new_constraint/8

new_constraint(+Name, ?Attribute, ?Frame, ?Old, ?New, +Context, +Check, +Action): cria (ou atualiza se o *Name* já existir) um *constraint* que valida atualizações de slot.

remove_constraint/0

remove_constraint: limpa da área de trabalho todos os *constraints*.

remove_constraint/2

remove_constraint(+Attribute, +Frame): limpa da área de trabalho todos os *constraints* associados ao *Attribute* de *Frame*.

**Data**

isa_data/2

isa_data(?Name, -Action): recupera o *Name* e *Action* associados a uma diretiva *data*.

new_data/2

new_data(+Name, +Action): acrescenta (ou substitui caso o *Name* já exista) um novo dado *Action* na área de trabalho. A *Action* é automaticamente invocada sempre que houver uma chamada a *run_data/0*, *run_data/1* ou *restart/0*, e é geralmente usada para inicializar alguns valores do slot.

remove_data/0

remove_data: limpa da área de trabalho todas as diretivas de dados.

restart/0

restart: limpa da área de trabalho todas as instâncias, valores de slots, fatos e exceções, reabilita todas as regras e finalmente roda todas as diretivas de dados.

run_data/0

run_data: chama as ações associadas a todas diretivas de dados. Se alguma ação falhar, então a mensagem indicando isso é escrita na saída corrente.

run_data/1

run_data(+Name): chama as ações associadas ao dado *Name*. Se a ação falhar, então a mensagem indicando isso é escrita na saída corrente.

**Demons**

isa_demon/7

isa_demon(?Name, ?Attribute, ?Frame, -Old, -New, -Context, -Action): recupera as características de um *demon*.

new_demon/7

new_demon(+Name, ?Attribute, ?Frame, ?Old, ?New, +Context, +Action): cria (ou atualiza se o *Name* já existir) um *demon* que reage a atualizações de slot.

remove_demon/0

remove_demons: limpa da área de trabalho todos os *demons*.

remove_demon/2

remove_demons(+Attribute, +Frame): limpa da área de trabalho todos os *demons* associados a um particular *Attribute* de um *Frame*.

**Facts**

disprove/1

disprove(+Goal): dereferencia cada um dos argumento de *Goal* e depois tenta invalidá-lo.

forget_exception/1

forget_exception(+Term): remove o termo dado (*Term*), se existir, das exceções na área de trabalho. Se subsequentes backtracking acontecerem, então a exceção retirada será reafirmada.

forget_fact/1

forget_fact(+Term): remove o termo dado (*Term*), se existir, dos fatos na área de trabalho. Se subsequentes backtracking acontecerem, então o fato retirado será reafirmado.

isa_exception/1

isa_exception(?Term): recupera uma exceção conhecida da área de trabalho.

isa_fact/1

isa_fact(?Term): recupera um fato conhecido da área de trabalho.

nospy_fact/1

nospy_fact(?Name): remove qualquer ponto de parada que tenha sido estabelecido pelo fato do *Name* dado.

prove/1

prove(+Goal): dereferencia cada um dos argumentos de *Goal* e tenta prová-lo.

remember_exception/1

remember_exception(+Term): acrescenta o *Term* dado como um nova exceção na área de trabalho, a qual pode ser acessada através dos procedimentos *prove/1* e *disprove/1*.

remember_fact/1

remember_fact(+Term): acrescenta o *Term* dado como um novo fato na área de trabalho, a qual pode ser acessada através dos procedimentos *prove/1* e *disprove/1*.

remove_exceptions/0

remove_exceptions: limpa da área de trabalho todas as exceções conhecidas.

remove_facts/0

remove_facts: limpa da área de trabalho todos os fatos conhecidos.

spied_fact/1

spied_fact(?Name): testa se um ponto de parada está ou não correntemente estabelecido.

spy_fact/1

spy_fact(?Name): se um fato ou uma exceção, no qual o functor principal é *Name*, é adicionada ou removida da área de trabalho, essa informação é escrita na saída corrente.

## Forward Chaining

atn/3

atn(+Name, +Names, -Newnames): programa pré-definido para reorganizar a agenda de regras após cada ciclo do motor foward chaining, de acordo com a rede de transição de regras.

back/3

back(+Name, +Names, -Newnames): programa pré-definido para reorganizar a agenda de regras após cada ciclo do motor foward chaining. Ele coloca a regra *Name* no final da agenda de regras.

crss/3

crss(+Names, -Name, -Action): programa pré-definido para selecionar uma regra para disparar durante cada ciclo do motor forward chaining. Esse é o algoritmo de resolução de conflitos de de scores.

crss/4

crss(+Name, -Name, -Action, +Threshold): esse é uma variação do algoritmo de resolução de conflitos para incorporar valores threshold. A seleção termina assim que uma regra satisfeita atinge o valor threshold.

cycle/3

cycle(+Name, +Names, -Newname): programa pré-definido para reorganizar a agenda de regras após cada ciclo do motor foward chaining de acordo com o ciclo de rotação.

fcfs/3

fcfs(+Names, -Name, -Action): programa pré-definido para selecionar uma regra para disparar durante cada ciclo do motor forward chaining. Esse é o algoritmo de seleção *first come first served*.

fire_rule/1

fire_rule(+Name): executa, incondicionalmente, a parte ação associada com a regra *Name*. A regra é disparada independente se o *Name* está habilitado ou se suas condições estão satisfeitas.

fixed/3

fixed(+Name, +Names, -Newname): programa pré-definido para processar a agenda de regras após cada ciclo do motor foward chaining. Ele deixa a agenda de regras inalterada.

forward_chain/5

forward_chain (+Selection, +Misfire, +Termination, +Update, +Agenda): inicia uma sessão forward chaining com o algoritmo de seleção e a agenda de regras dados.

forward_chain/6

forward_chain (+Selection, +Misfire, +Termination, +Update, +Agenda, -Sequence): se comporta exatamente como *forward_chain/5*, com exceção de que um pedaço a mais de informação é retornado: *Sequence* é a lista de nomes de regras que são realmente disparadas durante a sessão de forward_chaining.

forward_chain/7

forward_chain (+Selection, +Misfire, +Termination, +Update, +Agenda, -Sequence, -Result): se comporta exatamente como *forward_chain/6*, com exceção de que um pedaço a mais de informação, o *Result*, é retornado.

front/3

front(+Name, +Names, -Newnames): programa pré-definido para reorganizar a agenda de regras após cada ciclo do motor foward chaining. Ele traz a regra disparada mais recentemente para o início da agenda.

misfire/1

misfire (+Name): programa pré-definido para manipular falhas durante a sessão de forward chaining. O argumento *Name* é o nome da regra que falhou.

nospy_chain/0

nospy_chain: inabilita o traçado do motor forward chaining removendo quaisquer pontos de parada.

once/3

once(+Name, +Names, -Newnames): programa pré-definido para reorganizar a agenda de regras após cada ciclo do motor foward chaining. A regra disparada mais recentemente é removida da agenda.

possibles/3

possibles(+Name, +Names, -Newnames): programa pré-definido para reorganizar a agenda de regras após cada ciclo do motor foward chaining. Ele remove regras não satisfeitas da agenda.

spied_chain/0

spied_chain: testa se o motor forward chaining possui ou não um ponto de parada estabelecido.

spy_chain/0

spy_chain: estabelece um ponto de parada no motor forward chaining.

**Frames**

ancestor/2

ancestor(+Frame, -Ancestor): é o próprio frame ou algum ancestral, como pai, avô, bisavô, etc.

descendant/2

descendant(+Frame, -Descendant) : é um descendente do *Frame* na hierarquia, ou seja, *Descendant* é o próprio frame, ou um frame filho, ou um frame neto, etc.

isa_frame/2

isa_frame(?Name, ?Parents) : testa a existência de um frame e seus pais.

new_default/3

new_constraint(+Attribute, +Frame, +Value) : cria (ou substitui) um *Value* default de um *Attribute* particular de um *Frame*.

new_frame/2

new_frame(+Name, +Parents) : é criado um novo frame chamado *Name*, cuja posição na hierarquia de frame é determinada pela lista *Parents*.

remove_defaults/0

remove_defaults: limpa da área de trabalho todos os valores default definidos correntemente.

remove_defaults/1

remove_defaults(+Frame): limpa da área de trabalho todos os valores default de todos os atributos de um particular *Frame*.

remove_frame/1

remove_frame(+Frame) : remove o frame nomeado e todos os seus valores associados (ambos, corrente e default) e qualquer link de herança especializado.

remove_frames/0

remove_frames: limpa todos os frames definidos correntemente da área de trabalho do *Flex*.

**Functions**

isa_function/3

isa_function(?Name, ?Arguments, ?Value) : recupera a definição de uma função.

new_function/3

new_function(+Name, +Arguments, +Value) : armazena a definição de uma função *Name*. Funções são avaliadas em tempo de execução através de chamdas a *dereference/2*.

remove_function/1

remove_function(+Name) : remove a função nomeada da área de trabalho.

remove_functions/0

remove_functions : limpa todas as funções da área de trabalho .

**General**

comparison/3

comparison(?Relation, +Term1, +Term2) : os dois são dereferenciados (para *Value1* e *Value2*) e os valores comparados.

comparison/4

comparison(?Relation, +Term1, +Term2, +Group): se comporta como *comparison/3*, exceto que o nome do *Group* fornecido é usado como ordem da relação. Os dois termos são dereferenciados e os valores resultantes (*Value1* e *Value2*) comparados de acordo com suas posições relativas no grupo de elementos.

dereference/2

dereference(+Term, -Value): esse é o kernel dos procedimentos em tempo de execução do *Flex*. Ele é usado por todo o sistema como um interpretador de objetos *Flex*.

equality/2

equality(+Term1, +Term2): os dois termos são dereferenciados e os termos resultantes igualados. Se a igualdade falhar (ou existir backtracking em equality/2) então valores dereferenciados alternativos serão igualados.

flatten_group/2

flatten_group(+Name, -Elements): é usado para achatar uma hierarquia ou rede de grupo em seus elementos constituintes.
Grupos, como frames e heranças, podem ser construídos em uma hierarquia ou rede de grupos. Isso acontece quando os elementos de um grupo são eles mesmos nomes de outros grupos.

flex_name/1

flex_name(?Name): recupera (ou testa) o nome de um predicado *Flex*.

## Groups

comparison/4

comparison(?Relation, +Term1, +Term2, +Group): se comporta como *comparison/3*, exceto que o nome do *Group* fornecido é usado como ordem da relação. Os dois termos são dereferenciados e os valores resultantes (*Value1* e *Value2*) comparados de acordo com suas posições relativas no grupo de elementos.

every_instance/2

every_instance(+Name, -Elements): recupera (ou checa) todos os *Elements* que são instâncias de *Name*. *Name* pode ser o nome de um grupo, o nome de um frame ou uma instância ou o nome de uma relação unária.

flatten_group/2

flatten_group(+Name, -Elements): é usado para achatar uma hierarquia ou rede de grupo em seus elementos constituintes.
Grupos, como frames e heranças, podem ser construídos em uma hierarquia ou rede de grupos. Isso acontece quando os elementos de um grupo são eles mesmos nomes de outros grupos.

inclusion/2

inclusion(+List, ?Term): o *Term* e todos os membros da *List* são a princípio dereferenciados. Se o *Term* for uma lista, então é feito uma checagem para verificar se todos os seus membros são também membros de *List*. Do contrário, o *Term* dereferenciado é um membro de *List*.

isa_group/2

isa_group(?Name, -Elements): recupera o *Name* e *Elements* de um grupo.

new_group/2

new_group(+Name, +Elements): cria (ou substitui caso o *Name* já exista) um novo grupo contendo os *Elements* dados.

remove_groups/0

remove_groups: limpa da área de trabalho todos os grupos conhecidos.

some_instance/2

some_instance(+Name, ?Element): recupera (ou checa) um *Element* que é uma instância de *Name*. *Name* pode ser o nome de um grupo, de um frame, de uma instância ou de uma relação unária.

## Inheritance

ancestor/2

ancestor(+Frame, -Ancestor): é o próprio frame ou algum ancestral, como pai, avô, bisavô, etc.

descendant/2

descendant(+Frame, -Descendant): é um descendente do *Frame* na hierarquia, ou seja, *Descendant* é o próprio frame, ou um frame filho, ou um frame neto, etc.

inherit/3

inherit(+Attribute, +Frame, -Value): será feita uma busca na hierarquia de frames a partir dos pais de *Frame*. A busca irá terminar quando for encontrado um ancestral de *Frame* que possua um valor corrente ou default (*Value*) para o *Attribute*.

inherit/4

inherit(+Attribute, +Frame, -Value, -Ancestor): comporta-se como o *inherit/3*, com exceção de que um pedaço de informação adicional, *Ancestor* (indica de onde exatamente, na hierarquia de frames, o *Value* se descendeu), é retornado.

inheritance/0

inheritance: retorna para a herança default estabelecida.

inheritance/4

inheritance(?Search, ?Root, ?Plurality, ?Effort): permite a inspeção e/ou alteração de certas características do algoritmo de herança.

isa_link/3

isa_link (?Attribute, ?Frame, -Parents): teste por um link de herança do atributo de uma frame.

new_link/3

new_link(+Attribute, +Frame, +Parents): estabece um link de herança entre um *Frame* (e intâncias desse frame) e uma lista de *Parents*, de um particular *Attribute*.

remove_links/0

remove_links: limpa da área de trabalho todos os links de herança especializados definidos correntemente.

remove_links/1

remove_links(+Frame): remove os links de herança especializados de um particular *Frame*. Isso significa que a herança será revertida para a herança default da hierarquia de frame existente.

**Initialisation**

initialise/0

initialise: limpa completamente a área de trabalho de todos os frames, regras, perguntas, etc.

restart/0

restart: limpa da área de trabalho todas as instâncias, valores de slots, fatos e exceções, reabilita todas as regras e finalmente roda todas as diretivas de dados.

run_data/0

run_data: chama as ações associadas a todas diretivas de dados. Se alguma ação falhar, então a mensagem indicando isso é escrita na saída corrente.

run_data/1

run_data(+Name): chama as ações associadas ao dado *Name*. Se a ação falhar, então a mensagem indicando isso é escrita na saída corrente.

**Instances**

every_instance/2

every_instance(+Name, -Elements): recupera (ou checa) todos os *Elements* que são instâncias de *Name*. *Name* pode ser o nome de um grupo, o nome de um frame ou uma instância ou o nome de uma relação unária.

isa_instance/2

isa_instance(?Instance, ?Frame): testa a existência de uma instância de um frame.

new_instance/2

new_instance(+Instance, +Frame): cria uma nova *Instance* de um *Frame* específico.

remove_instance/1

remove_instance(+Instance): limpa da área de trabalho essa *Instance* específica de algum frame, juntamente com seus valores correntes associados e qualquer link de herança especializado.

remove_instances/0

remove_instances: limpa da área de trabalho todas as instâncias definidas correntemente de todos os frames.

remove_instances/1

remove_instances(+Frame): limpa da área de trabalho todas as instâncias conhecidas de um particular *Frame*. Isso não somente irá remover todos os links entre as instâncias e o frame, como também irá remover qualquer slot existente daquelas instâncias.

some_instance/2

some_instance(+Name, ?Element): recupera (ou checa) um *Element* que é uma instância de *Name*. *Name* pode ser o nome de um grupo, de um frame, de uma instância ou de uma relação unária.

**Launches**

isa_launch/5
isa_launch(?Name, ?Instance, ?Frame, -Context, -Action): recupera as características de um *launch*.

new_launch/5
new_launch(+Name, ?Instance, ?Frame, +Context, +Action): cria (ou atualiza se o *Name* já existir) um *launch* que reage à criação de novas instâncias de frames.

remove_launches/0
remove_launches: limpa todos os procedimentos *launch* conhecidos da área de trabalho.

remove_launches/1
remove_launches(+Frame): limpa da área de trabalho todos os *launches* associados a um particular *Frame*.

**Logic**

isa_logic/1
isa_logic(?Logic): resupera ou checa se uma *Logic* está em vigor.

new_logic/1
new_logic(+Logic): muda a lógica estabelecida do sistema para *Logic*.

remove_logic/1
remove_logic(+Logic): remove *Logic* da área de trabalho.

remove_logics/0
remove_logics: limpa da área de trabalho todas as lógicas especializadas.

**Questions**

answer/2
answer(+Name, -Value): recupera a resposta dada à pergunta *Name*. Se a pergunta ainda não foi feita, então é feita uma chamada à *ask/1* e obtida a resposta.

ask/1
ask(+Name): Faz a pergunta *Name* e armazena a resposta na variável global *Name*. A resposta pode ser acessada tanto através do mecanismo de deferência ou diretamente através de *answer/2* ou *isa_value/2*.

isa_question/4
isa_question (?Name, -Question, -Answer, -Explanation): recupera uma pergunta da área de trabalho.

new_question/4
new_question(+Name, +Question, +Answer, +Explanation): acrecenta (ou substitui caso o *Name* já exista) uma nova pergunta na área de trabalho.

remove_questions/0
remove_questions: limpa da área de trabalho todas as perguntas conhecidas.

**Relations**

isa_relation/2
isa_relation(?Name, ?Arity): recupera ou checa o *Name* e a *Arity* de uma relação definida.

new_relation/2
new_relation(+Name, +Arity): armazena o *Name* e a *Arity* de uma relação do *Flex*. Essa informação é usada quando se inicializa a área de trabalho.

remove_relation/2
remove_relation(+Name, +Arity): retira a relação *Name/Arity* da área de trabalho.

remove_relations/0
remove_relations: retira todas as relações conhecidas da área de trabalho.

**Rules**

all_rules/1
all_rules(-Names): Encontra o nome de todas as regras na área de trabalho.

disable_rules/1
disabel_rules(+Names): todas as regras na lista *Names* estão inabilitadas, e não serão consideradas para disparo. É como se elas fossem temporariamente removidas da área de trabalho.

enable_rules/0
enable_rules: reabilita todas as regras que tenham sido anteriormente inabilitadas.

enable_rules/1
enable_rules(+Names): reabilita todas as regras contidas em *Names* que tenham sido anteriormente inabilitadas. Elas podem agora sere consideradas como regras que podem ser disparadas.

explain/1
explain(+Rules): inicia uma caixa de diálogos que permite cada regra da lista *Rules* ser explicada. A explicação depende de como os parâmetros de explicação da regra foram setados por *new_rule/5*.

fire_rule/1
fire_rule(+Name): executa, incondicionalmente, a parte ação associada com a regra *Name*. A regra é disparada independente se o *Name* está habilitado ou se suas condições estão satisfeitas.

isa_disabled_rule/1
isa_disable_rule(?Name): recupera (ou testa) o *Name* de uma regra que está correntemente inabilitada.

isa_rule/5
isa_rule(?Name, -Conditions, -Action, -Explanation, -Score): recupera as características de uma regra da área de trabalho.

new_rule/5

new_rule(+Name, +Conditions, +Action, +Explanation, +Score): acresta uma nova regra à área de trabalho.

nospy_rule/1

nospy_rule(?Name): remove qualquer ponto de parada que tenha sido estabelecido pela regra nomeada. Se o *Name* for uma variável, então os pontos de parada são removidos de todas as regras.

remove_rules/0

remove_rules: limpa da área de trabalho todas as regras conhecidas.

spied_rule/1

spied_rule(?Name): testa se a regra nomeada correntemente possui ou não um ponto de parada estabelecido.

spy_rule/1

spy_rule(?Name): estabelece um ponto de parada na regra nomeada.

trigger_rule/1

trigger_rule(+Name): se *Name* não for uma regra inabilitada e suas condições forem satisfeitas, então a parte de ação da regras são disparadas.

**Slots**

add_value/2

add_value(+Slot, +Term): pode ser usado tanto para aumentar listas quanto para incrementar números.

isa_default/3

isa_default(?Attribute, ?Frame, -Value): teste por um valor default de um atributo de um frame.

isa_slot/3

isa_slot(?Attribute, ?Frame, -Value): teste pelo valor corrente do atributo de um frame.

isa_value/2

isa_value(?Slot, -Value): recupera o valor corrente, ou na falta desse o valor default, de um *Slot*.

is_known/1

is_known(+Slot): a chamada é bem sucedida apenas se existir um valor corrente ou default para o *Slot*.

lookup/3

lookup(+Attribute, +Frame, -Value): recupera o valor corrente de um atributo de um frame, como em *isa_slot/3*. Caso não exista valor corrente, o valor default será retornado.

lookup/4

lookup(+Attribute, +Frame, -Value, -Ancestor): se comporta como o *lokup/3*, com exceção que um pedaço adicional de informação, o *Ancestor*, é retornado. Ele indica de onde exatamente na hierarquia de frame veio o *Value*.

new_default/3

new_constraint(+Attribute, +Frame, +Value): cria (ou substitui) um *Value* default de um *Attribute* particular de um *Frame*.

new_slot/3

new_slot(+Attribute, +Frame, +Value): cria (ou substitui) o *Value* corrente de um particular *Attribute* de um *Frame* (ou instância de um frame).

new_value/2

new_value(+Slot, +Term): o *Term* dado é primeiramente dereferenciado para algum *Value*.

nospy_slot/2

nospy_slot(?Attribute, ?Frame): remove qualquer ponto de parada que tenha sido estabelecido pelo *Attribute* e *Frame*.

remove_defaults/0

remove_defaults: limpa da área de trabalho todos os valores default definidos correntemente.

remove_defaults/1

remove_defaults(+Frame): limpa da área de trabalho todos os valores default de todos os atributos de um particular *Frame*.

remove_slots/0

remove_slots: limpa todos os valores correntes da área de trabalho.

remove_slots/1

remove_slots(+Frame): limpa da área de trabalho todos os slots conhecidos (isto é, valores correntes) de um particular *Frame* (e instâncias do frame).

spied_slot/2

spied_slot(?Attribute, ?Frame): checa se existe ou não um ponto de parada no *Attribute* e *Frame*.

spy_slot/2

spy_slot(?Attribute, ?Frame): estabelece um ponto de parada no *Attribute* de um *Frame*, de modo que se o slot for atulizado, informações relevantes são mostradas na saída corrente.

sub_value/2

sub_value(+Slot, +Term): pode ser tanto usado para remover itens de uma lista ou decrementar números.

**Synonyms**

isa_synonym/2

isa_synonym (?Name, -Term): recupera o *Name* usado como sinônimo do *Term*.

new_synonym/2

new_synonym (+Name, +Term): acrecenta (ou substitui caso o *Name* já exista) um novo sinônimo para *Term*.

remove_synonyms/0

remove_synonym: limpa da área de trabalho todos os sinônimos conhecidos.

**Templates**

isa_template/3

isa_template(?Name, -Positive, -Negative): recupera tanto a forma *Positive* quanto a *Negative* de um template. As formas do template serão retornadas como um lista de listas, com quebra nas posições do parâmetro.

new_template/3

new_template(+Name, +Positive, +Negative): acrescenta os templates *Positive* e *Negative* de *Name* na área de trabalho.

remove_templates/0

remove_templates: limpa da área de trabalho todos os templates conhecidos.

remove_templates/1

remove_templates(+Name): limpa da área de trabalho todos os templates, ambos positivo e negativo, de um dado *Name*.

**Watchdogs**

isa_watchdog/6

isa_watchdog(?Name, ?Attribute, ?Frame, -Context, -Check, -Action): recupera as características de um procedimento *watchdog*.

new_watchdog/6

new_watchdog(+Name, ?Attribute, ?Frame, +Context, +Check, +Action): cria (ou atualiza se o *Name* já existir) um procedimento *watchdog* que controla o acesso a um slot.

remove_watchdogs/0

remove_watchdog: limpa da área de trabalho todas as *watchdog*.

remove_watchdogs/2

remove_watchdog(+Attribute, +Frame): limpa da área de trabalho todos os procedimentos *watchdog* associados a um *Attribute* de um *Frame*.

# Apêndice C - Exemplo

## <u>Exemplo 1</u>: *species.ksl*

/* The Species Identification Example -
Ported from Mike to WIN-PROLOG by Clive Spenser and Dave Westwood

The objective of this example is to show two contrasting methods, forward and backward chaining, for identifying a species of animal given a set of attributes.

<u>Description</u>
The backward-chaining method looks at a given species and attempts to prove, by asking questions related to the attributes of the species, if this species is the correct one.

The forward-chaining method looks at the attributes provided prior to starting the forward-chaining process and attempts to build on these attributes until a full description of a species is given.

<u>Running The Example</u>
The species identification example reports a species appropriate to the answers to some discriminating questions. To run the forward-chaining method enter the following goal at the command line:

?- find_species_forward .

To run the backward-chaining method enter the following goal at the command line:

?- find_species_backward .

<u>Flex Technical Points</u>
The following flex technical points are demonstrated in this example:

1. Disjunctions ("or") used in the conditions of forward-chaining rules.
2. The constraining of backward-chaining relations to a single solution.

*/

% The attribute question is used in the forward-chaining method.
% It gives a list of attributes which the user can provide before forward-chaining starts.

```
question attributes
  Which attributes do you know to start with? ;
  choose some of attribute_types  .

group attribute_types
  body_covering, colour, eats, eyes, feeds_young_on, feet,
  legs_and_neck, marking, motion, reproduction, teeth .
```

```
% The following questions and groups are used in both the forward and
% backward chaining examples.

question body_covering
  What is the body_covering? ;
  choose one of body_covering_types
  because 'hair indicates a mammal, while feathers a bird' .

group body_covering_types
  hair, feathers, other.

question colour
  What is the colour? ;
  choose one of colour_types .

group colour_types
  tawny, black_and_white, other.

question eats
  What does it eat? ;
  choose one of eats_types .

group eats_types
  meat, grass, other.

question eyes
  How do its eyes point? ;
  choose one of eyes_types .

group eyes_types
  point_forward, point_sideways.

question feeds_young_on
  What does it feed its young on? ;
  choose one of feeds_young_on_types .

group feeds_young_on_types
  milk, other.

question feet
  What kind of feet does it have? ;
  choose one of feet_types .

group feet_types
  claws, hoofs, other.

question legs_and_neck
  Are its legs and neck long or short? ;
  choose one of legs_and_neck_types .

group legs_and_neck_types
  long, short, other.
```

```
question marking
  What kind of marking does it have? ;
  choose one of marking_types .

group marking_types
  dark_spots, black_stripes, other.

question motion
  How does it move? ;
  choose one of motion_types .

group motion_types
  flies, swims, walks, other.

question reproduction
  How does it reproduce? ;
  choose one of reproduction_types .

group reproduction_types
  eggs, other.

question teeth
  Are its teeth pointed or blunt? ;
  choose one of teeth_types .

group teeth_types
  pointed, blunt, other.

% The next section defines the ruleset and rules used in the forward-chaining method.

ruleset identify
  contains identity_rules;
  update ruleset by removing each selected rule .

group identity_rules
  mammal, bird, feeding_type_1, feeding_type_2,
  species_1, species_2, species_3, species_4,
  species_5, species_6, species_7 .

% The following rule has a disjunction in its conditions.
% It will be fired if either of the conditions hold.

rule mammal
   if    the body_covering is hair
   or    the feeds_young_on is milk
   then  the creature`s genus becomes mammal .

rule bird
   if    the creature`s genus is unknown
   and   [ the body_covering is feathers
   or    the motion is flies
   and   the reproduction is eggs ]
   then  the creature`s genus becomes bird.
```

```
% The "or" in the following rule says that either the first two conditions must be true or
% the next three conditions must be true before the rule can be fired.

rule feeding_type_1
   if    the creature's genus is mammal
   and   the eats is meat
   or    the teeth is pointed
   and   the feet is claws
   and   the eyes is point_forward
   then  the creature's feeding_type becomes carnivore.

% The "or" in the following rule is restricted to the last two conditions by the presence of
% the square brackets. This means that the first two conditions must hold along with
% either of the last two before the rule can be fired.

rule feeding_type_2
   if    the creature's genus is mammal
   and   the creature's feeding_type is unknown
   and   [ the eats is grass
   or    the feet is hoofs ]
   then  the creature's feeding_type becomes ungulate.

rule species_1
   if    the creature's feeding_type is carnivore
   and   the colour is tawny
   and   the marking is dark_spots
   then  the creature's species becomes cheetah.

rule species_2
   if    the creature's feeding_type is carnivore
   and   the colour is tawny
   and   the marking is black_stripes
   then  the creature's species becomes tiger.

rule species_3
   if    the creature's feeding_type is ungulate
   and   the colour is tawny
   and   the marking is dark_spots
   and   the legs_and_neck is long
   then  the creature's species becomes giraffe.

rule species_4
   if    the creature's feeding_type is ungulate
   and   the colour is black_and_white
   then  the creature's species becomes zebra.

rule species_5
   if    the creature's genus is bird
   and   the motion is walks
   and   the colour is black_and_white
   and   the legs_and_neck is long
   then  the creature's species becomes ostrich.
```

```
rule species_6
   if     the creature`s genus is bird
   and    the motion is swims
   and    the colour is black_and_white
   then   the creature`s species becomes penguin.

rule species_7
   if     the creature`s genus is bird
   and    the motion is flies
   then   the creature`s species becomes albatross.

% This section defines the relations for the backward-chaining method.

% The following relation uses one/1 to constrain the suggest_family/1 relation to a single
% solution. So that un-necessary questions are not asked. That is, once we have
% established that the family is mammal we don't want to go on to check that the family
% is not bird

relation check_family( Family )
  if one( suggest_family( SuggestedFamily ) )
  and Family = SuggestedFamily .

relation suggest_family( mammal )
  if the body_covering is hair
  or  the body_covering is other
  and the feeds_young_on is milk.

relation suggest_family( bird )
  if the body_covering is feathers
  or  the motion is flies
  and the reproduction is eggs.

relation check_feeding_type( FeedingType )
  if check_family( mammal )
  and one( suggest_feeding_type( SuggestedFeedingType ) )
  and FeedingType = SuggestedFeedingType .

relation suggest_feeding_type( carnivore )
  if eats is meat
  or  teeth is pointed
  and feet is claws
  and eyes is point_forward .

relation suggest_feeding_type( ungulate )
  if eats is grass
  or  feet is hoofs.

relation check_species( Species )
  if one( suggest_species( SuggestedSpecies ) )
  and Species = SuggestedSpecies .
```

```
relation suggest_species( cheetah )
  if check_feeding_type( carnivore )
  and colour is tawny
  and marking is dark_spots
  and legs_and_neck is short.

relation suggest_species( tiger )
  if check_feeding_type( carnivore )
  and colour is tawny
  and marking is black_stripes.

relation suggest_species( giraffe )
  if check_feeding_type( ungulate )
  and colour is tawny
  and marking is dark_spots
  and legs_and_neck is long.

relation suggest_species( zebra )
  if check_feeding_type( ungulate )
  and colour is black_and_white.

relation suggest_species( ostrich )
  if check_family( bird )
  and motion is walks
  and ask colour and colour is black_and_white
  and legs_and_neck is long.

relation suggest_species( penguin )
  if check_family( bird )
  and motion is swims
  and ask colour and colour is black_and_white.

relation suggest_species( albatross )
  if check_family( bird ).

% This action is the top-level goal for the backward-chaining solution

action find_species_backward
 do restart
 and check_species( Species )
 and write( Species )
 and nl .

% This action is the top-level goal for the forward-chaining solution

action find_species_forward ;
  do restart
  and ask attributes
  and invoke ruleset identify
  and write( creature`s species )
  and nl .
```

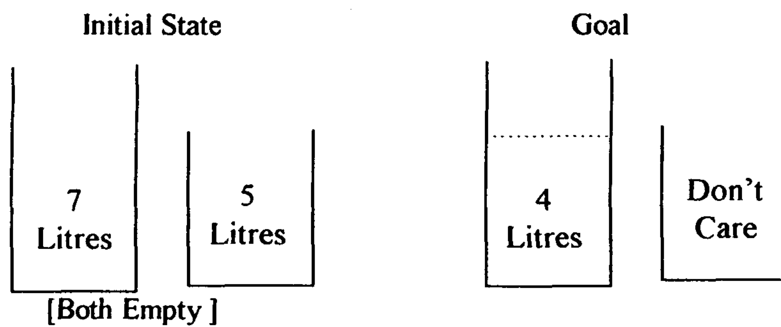
## ***Exemplo 2: water.ksl***

/* Water Containers Example - Adapted by - Phil Vasey

This example implements the water containers problem, as defined in
'Logic for Problem Solving' by Robert Kowalski.

Description
Given both a seven and a five litre container, initially empty, the goal is to find a sequence of actions which leaves four litres of liquid in the seven litre container. There are three kinds of actions which can alter the state of the containers:

i. A container can be filled.
ii. A container can be emptied.
iii. Liquid can be poured from one contained into the other, until the first is empty or the second is full.




Running The Example
The water containers example lets you choose between having or not having a presentation of the explanations for the rules that were fired during the solution of the problem. The state of the containers is reported at each stage of the solution. To run the example enter the following goal at the command line:

?- find_container_solution .

Flex Technical Points
The following flex technical points are demonstrated in this example:

1. The use of dynamically updated scores in forward chaining rules.
2. The linking of forward chaining rules into a network using groups.
3. The use of rule explanations.
*/

```
% the following template allows the words spare capacity to be used in place of the
% word spare

template spare
  spare capacity .
```

```
% the initial contents and capacity of the two containers is set here

frame container ;
  default contents are 0 and
  default capacity is 7 and
  default spare capacity is its capacity minus its contents .

instance master is a container .

instance slave is a container ;
  capacity is 5 .

template not_full
  ^ is not full .

template not_empty
  ^ is not empty .

template enough
  ^ contains more than ^ ;
  ^ does not contain more than ^ .

relation X is not full
  if X`s contents are below X`s capacity .

relation X is not empty
  if X`s contents are above 0 .

relation X contains more than Z
  if X`s contents are above Z .

% The transfer operations ...

template fill_up
  fill up ^ .

template empty_out
  empty out ^ .

template fill_from
  fill ^ from ^ .

template empty_into
  empty ^ into ^ .

action fill up X ;
  do X`s contents become X`s capacity .

action empty out X ;
  do X`s contents become 0 .
```

```
action fill X from Y
  do  subtract X`s spare capacity from Y`s contents
  and fill up X .

action empty X into Y ;
  do  add X`s contents to Y`s contents
  and empty out X .

% The rules for master ...

rule fill_master
  if    the master is not full
  then  fill up the master
  and   write_action( 'fill master' )
  because the emptier the master the better it is to fill it ;
  score master`s spare capacity .

rule empty_master
  if    the master is not empty
  then  empty out the master
  and   write_action( 'empty master' )
  because the fuller the master the better it is to empty it ;
  score master`s contents .

rule fill_master_from_slave
  if    the master is not full
  and   the slave contains more than the master`s spare capacity
  then  fill the master from the slave
  and   write_action( 'fill master from slave' )
  because the emptier the master and the fuller the slave the better it
        is to fill the master from the slave ;
  score master`s spare capacity + slave`s contents .

rule empty_slave_into_master
  if      the slave is not empty
  and   the slave does not contain more than the master`s spare capacity
  then    empty the slave into the master
  and   write_action( 'empty slave into master' )
  because the emptier the master and the fuller the slave the better it
        is to empty the slave contents into the master ;
  score master`s spare capacity + slave`s contents .

% The rules for slave ...

rule fill_slave
  if    the slave is not full
  then  fill up the slave
  and   write_action( 'fill slave' )
  because the emptier the slave the better it is to fill it ;
  score slave`s spare capacity .
```

```
rule empty_slave
  if   the slave is not empty
  then  empty out the slave
  and   write_action( 'empty slave' )
  because the fuller the slave the better it is to empty it ;
  score slave`s contents .

rule fill_slave_from_master
  if   the slave is not full
  and   the master contains more than the slave`s spare capacity
  then  fill the slave from the master
  and   write_action( 'fill slave from master' )
  because the emptier the slave and the fuller the master the better
        it is to fill the slave from the master ;
  score slave`s spare capacity + master`s contents .

rule empty_master_into_slave
  if   the master is not empty
  and   the master does not contain more than the slave`s spare capacity
  then  empty the master into the slave
  and   write_action( 'empty master into slave' )
  because the emptier the slave and the fuller the master the better
        it is to empty the master contents into the slave ;
  score slave`s spare capacity + master`s contents .

ruleset container_rules
  contains start_rules ;
  select rule using conflict resolution with threshold 6 ;
  update ruleset using rule transition network ;
  initiate by doing restart ;
  terminate when the contents of the master is 4 .

% The rules are now linked into a network by declaring groups ...

group start_rules
  fill_master, fill_slave .

group fill_master
  empty_master_into_slave, fill_slave_from_master,
  fill_slave, empty_slave .

group empty_master
  fill_master_from_slave, empty_slave_into_master,
  fill_slave, empty_slave .

group fill_master_from_slave
  empty_master, fill_slave, empty_slave .

group empty_master_into_slave
  fill_master, fill_slave, empty_slave .
```

```
group fill_slave
  empty_slave_into_master, fill_master_from_slave,
  fill_master, empty_master .

group empty_slave
  fill_slave_from_master, empty_master_into_slave,
  fill_master, empty_master .

group fill_slave_from_master
  empty_slave, fill_master, empty_master .

group empty_slave_into_master
  fill_slave, fill_master, empty_master .

% The following action uses the forward_chain/6 flex support predicate to return the list
% of rules that were used during the solution to the problem. This list of rules is then
% passed through to the explain facility, which allows you to look at the explanations for
% the rules that were fired.

action find_container_solution ;
 do  if the answer to solution_presentation is explanation
     then forward_chain(crss(6),misfire,
              termination_criteria,
              atn, start_rules,
                   RulesUsed )
        and explain( RulesUsed )
     else invoke ruleset container_rules
     end if
 and write( 'the goal state has now been reached' )
 and nl .

question solution_presentation
 How would you like the solution presented? ;
 choose one of explanation, 'no explanation' .

relation termination_criteria
 if the contents of the master is 4 .

relation write_container( 0, Capacity, _ )
 if write( '|_____|') .

relation write_container( Line, Capacity, _ )
 if  Line > Capacity .

relation write_container( Line, _, Contents )
 if  Contents < Line
 and write( '|     |') .

relation write_container( _, _, _ )
 if  write( '|ÛÛÛÛÛ|') .
```

```
action write_containers
  do nl
  and check MaxLine is equal to the capacity of master
  and for Line from MaxLine to 0 step -1
     do write_container( Line, the capacity of master, the contents of master )
     and tab(8)
     and write_container( Line, the capacity of slave, the contents of slave )
     and nl
  end for
  and write( master ) and tab(10) and write( slave) and nl and nl .

action write_action( Action )
 do nl
 and write(Action)
 and nl
 and write_containers .
```