UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

CONTRIBUIÇÃO AO ESTUDO E DESENVOLVIMENTO DE UM SISTEMA DE REGRAS DE PRODUÇÃO

Dissertação submetida à Universidade Federal de Santa Catarina como requisito parcial à obtenção do grau de Mestre em Engenharia Elétrica

CELSO ANTÔNIO ALVES KAESTNER

Florianópolis, abril de 1989

CONTRIBUIÇÃO AO ESTUDO E DESENVOLVIMENTO
DE UM SISTEMA DE REGRAS DE PRODUÇÃO

CELSO ANTÔNIO ALVES KAESTNER

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA ELÉTRICA

ESPECIALIDADE ENGENHARIA ELÉTRICA E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO

Prof. Hervé E. Garnousset, Dr.Ing.
ORIENTADOR

Prof. José Carlos M.Bermudez, Ph.D.
COORDENADOR DO CURSO

BANCA EXAMINADORA

Prof. Hervé E. Garnousset, Dr.Ing.
PRESIDENTE

Prof. Daniel Schwabe, Ph.D.

Prof. Flávio R. Dias Velasco, Ph.D.

Prof. Ricardo Barcia, Ph.D.

A Melissa, Camila e Tatiana

Agradecimentos

Agradeço à CAPES pelo auxílio financeiro recebido em forma de bolsa.

Ao Centro Federal de Educação Tecnológica do Paraná (CEFET-PR), à Pontifícia Universidade Católica do Paraná (PUC-PR) pelo integral apoio.

Ao Professor Doutor Hervé Eric Garnousset pela orientação, interesse e dedicação ao longo do desenvolvimento deste trabalho.

Ao corpo docente do Curso de Pós-graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina pelos conhecimentos recebidos. Em particular aos professores Dr. Augusto Humberto Bruciapaglia, Jean-Marie Farines e Joni da Silva Fraga e demais integrantes do Laboratório de Controle e Microinformática (LCMI) pela atenção.

A minha esposa Lílian, a meus pais Dalila e Egon e a todos que me auxiliaram durante esta caminhada.

Aos amigos que tornaram este período de minha vida uma agradável fase de descoberta de novos valores.

Sumário

RESUMO

A presente dissertação apresenta o SP1, um Sistema de Produção que manipula regras com variáveis quantificadas. O SP1 incorpora uma linguagem para representação do conhecimento, um interpretador e um ambiente para execução.

A linguagem SP1 utiliza uma notação predicativa e se caracteriza por sua legibilidade e extensibilidade.

Para remediar a ineficácia da operação de filtragem, crítica em um sistema de produção, o interpretador do SP1 incorpora um novo algoritmo baseado na compilação das regras, que procura reduzir ao máximo os testes a efetuar para a instanciação das regras pela construção de uma rede otimizada. Vários tipos de redundâncias estruturais são assim evitadas. Esta rede é utilizada para propagar as modificações produzidas nos dados e atualizar as instâncias sobre as regras.

O ambiente de execução inclui as funções necessárias para a execução, monitoração e depuração.

ABSTRACT

This dissertation presents SP1, an expert system tool which uses a first order production rule model. The SP1 system includes a language for knowledge representation, an interpreter and an environmemt for execution and debugging.

The SP1 language uses a predicative notation and gives great legibility and extensibility.

In order to minimize the matching cost, a rule compiler was implemented. It reduces in large scale the number of tests for rules instantiation, by constructing a pattern equivalent network. This network is used in order to propagate every data modifications produced in the working memory by rule execution and to update the conflict set. The incremental feature of the network construction allows that new rules can be dynamically added to the rule memory.

The execution environment includes all necessary functions for execution, monitoration and debugging.

LISTA DE ABREVIATURAS

BC : Base de Conhecimentos

CC : Conjunto de Conflitos

IA : Inteligência Artificial

LDR : Lado Direito de uma Regra

LER : Lado Esquerdo de uma Regra

MT : Memória de Trabalho

MR : Memória de Regras

MI : Motor (ou Máquina) de Inferência

SE : Sistema Especialista

SP : Sistema (de regras) de Produção

I. INTRODUÇÃO

A partir dos anos 70 o uso de técnicas de Inteligência Artificial em Informática tem novo impulso, com o desenvolvimento dos primeiros Sistemas Especialistas (SE). Os SE são programas que utilizam o conhecimento de especialistas humanos para a realização, sobre um domínio restrito e bem definido, de certas tarefas cuja representação por métodos tradicionais, através de algoritmos, se torna extremamente difícil ou até impossível.

O sucesso dos SE tem sido amplo e suas áreas de atuação são cada vez mais diversificadas. Para uma descrição mais detalhada sobre os SE e suas aplicações, ver por exemplo [HAYES-ROTH 83], [HARMONN 85], [WATERMAN 86], [NILSSON 80], [FARRENY 87].

O desenvolvimento de SE é uma tarefa incremental e requer um ambiente de desenvolvimento adequado. Aspectos requeridos por um tal ambiente são flexibilidade, processamento simbólico, e casamento de padrões (matching) [VELASCO 88b]. Tais características levaram à criação de linguagens orientadas para tal desenvolvimento, dentre as quais os Sistemas de Produção (SP) tem se revelado particularmente adequados.

Os SP constituem um paradigma de representação centrado num conjunto não-ordenado de pares (condição, ação) denominados regras de produção. Sua popularidade provêm: da grande expressividade das regras de produção, que são ao mesmo tempo um modelo natural e de fácil compreensão; do aspecto declarativo das regras, pois cada uma delas representa uma porção especifíca do

saber, o que proporciona um desenvolvimento fácil e rápido; e da forte separação entre o conhecimento e a sua forma de utilização (controle) que este formalismo oferece.

No caso dos SE as regras de produção representam o savoir-faire do especialista. Cada regra corresponde a uma parcela de conhecimento que é aplicável a situação do problema decrita por seu antecedente.

Esta dissertação tem como objeto o SP1, um sistema de regras de produção que manipula regras com variáveis quantificadas.

As principais etapas cumpridas foram :

(a) especificação da linguagem de representação do conhecimento para o sistema;

(b) desenvolvimento de um algoritmo para a operação de filtragem;

(c) desenvolvimento do motor de inferência do sistema e mecanismos de controle;

(d) implementação de uma versão protótipo do sistema em LISP; e

(e) especificação e implementação de um ambiente de execução para o sistema.

Estes assuntos são desenvolvidos neste trabalho da seguinte forma:

O capítulo II faz uma descrição dos Sistemas de Produção, com fixação da terminologia empregada, descrição de critérios para classificação dos SP e caracterização do SP1 dentro dos conceitos apresentados.

O capítulo III apresenta a linguagem SP1, com descrição dos principais comandos e apresentação de exemplos de utilização.

O capítulo IV descreve o algoritmo de filtragem do sistema, que se baseia na construção de um compilador de regras e na geração incremental do conjunto de conflitos.

O capítulo V apresenta o motor de inferência do sistema, descrevendo os algoritmos utilizados para o controle dentro das diversas opções disponíveis.

O capítulo VI apresenta o ambiente de execução do sistema, a partir do qual o usuário pode configurar da maneira desejada o ambiente de trabalho e controlar o processo de inferência.

Finalmente, o capítulo VII apresenta as conclusões e perspectivas finais sobre o trabalho desenvolvido.

II. SISTEMAS DE PRODUÇÃO

O modelo "Sistema de Produção" (SP) foi introduzido por Post (1943), na definição de linguagens e gramáticas formais. Quando surgem os primeiros sistemas baseados no conhecimento (Knowledge-Based Systems), no início da década de 70, o modelo de Post se revela um poderoso formalismo de programação devido ao carácter declarativo de expressão do conhecimento e ao aspecto associativo de sua manipulação. A fim de aumentar sua expressividade o modelo é enriquecido de sofisticados mecanismos de filtragem.

Este primeiro capítulo objetiva fixar a terminologia que será utilizada ao longo do texto, e caracterizar o sistema de produção SP1 dentro dos conceitos apresentados.

Já que não há uma análise formal única para os SP, e que as implementações exploram variações em praticamente todos os aspectos, a abordagem seguida neste capítulo se limitará a uma apresentação informal, considerando o que se encontra classicamente.

2.1 SISTEMAS DE PRODUÇÃO EM IA:

2.1.1 ARQUITETURA DE UM SISTEMA DE PRODUÇÃO:

Um SP é composto basicamente pelos três elementos seguintes:

(1) Uma base de dados global, denominada Memória de Trabalho (MT), cujos elementos descrevem situações estabelecidas ou a estabelecer durante o processo de solução.

(2) Um conjunto de regras de produção, denominado Memória de Regras (MR), que representa o conhecimento operacional sobre o problema.

(3) Um interpretador, denominado Motor (ou Máquina) de Inferência (MI), responsável pelo controle do sistema.

A união dos conjuntos MT e MR é normalmente designada como base de conhecimentos (BC).

Intrinsecamente relacionada a estas estruturas está a linguagem utilizada pelo SP. Na elaboração desta linguagem devem ser considerados fatores tais como: legibilidade, expressividade, eficiência computacional e extensibilidade.

Os SP apresentam duas importantes características:

(a) "acesso associativo": o acesso às regras não é feito explicitamente por apontamento, mas depende do estado do problema; e

(b) "não-determinismo": durante a execução do SP existem etapas de escolha abertas a diversas alternativas, o que corresponde a uma separação entre o conhecimento e seu uso.

Estas características conferem aos SP poder e generalidade, e possibilitam que a execução dos SP seja baseada na freqüente reavaliação do estado do sistema (controle dirigido pelos dados),

em contraposição a seqüência fixa dos algoritmos (controle dirigido pelas instruções).

2.1.2 MEMÓRIA DE TRABALHO:

A memória de trabalho (MT) contém a qualquer momento o conhecimento que o sistema adquiriu sobre o problema que está resolvendo. Representa portanto o estado corrente do problema. Dependendo da aplicação a MT pode ser uma simples estrutura vetorial (Expert-Easy [HARMON 85]) ou até um banco de dados relacional (Hearsay-III [HAYES-ROTH 83]).

Os elementos da MT são denominados "objetos", e podem representar tanto instâncias de objetos físicos ligados ao problema a resolver, como elementos conceituais tais como metas ou objetivos que devem ser atingidos em etapas intermediárias do processo de resolução. Alguns sistemas podem distinguir diferentes categorias de objetos, como "fatos" e "hipóteses" no sistema Expert [HAYES-ROTH 83] ou "fatos", "problemas" e "planos" no sistema Argos-II [FARRENY 87].

Os objetos condicionam a aplicação das regras e podem ser criados, modificados ou destruídos pela execução das mesmas. São usualmente considerados como instâncias de classes arbitrárias de objetos, caracterizadas por um conjunto de atributos. Cada atributo reflete uma característica do objeto, e é definido por um identificador e pelo domínio de seus valores.

Além dos atributos, muitos SP associam aos objetos outras características com finalidade especial durante o processo de

inferência, tais como: marcas de tempo, representando a ordem de criação do objeto na MT, como no sistema OPS5 [FORGY 81]; fatores de certeza que designam o grau de crença que se tem sobre a asserção representada pelo objeto, como nos sistemas Mycin [BUCHANAN 85], Expert [HAYES-ROTH 83] e M.1 [HARMONN 85]; etc.

2.1.3 MEMÓRIA DE REGRAS:

A MR é constituída por um conjunto não-ordenado de regras de produção.

A parte condição de uma regra (geralmente seu lado esquerdo -LER ou antecedente) deve ser satisfeita pelo estado corrente (MT) para que a regra possa ser aplicada (ou disparada). A parte ação (geralmente o lado direito -LDR ou conseqüente) especifica a alteração sobre este estado a ser efetuada quando do disparo da regra.

A determinação da regra a ser disparada depende da estratégia de controle do SP e está a cargo do MI, como será visto a seguir.

Uma regra deve conter todas as informações necessárias a realização de suas ações, de modo que as regras podem ser acrescentadas, retiradas ou modificadas sem que isto cause outros efeitos sobre o sistema.

PARTE CONDIÇÃO:

É normalmente composta por um conjunto de condições interligados por operadores booleanos, usualmente conjunções e negações.

O conjunto das condições a serem testadas sobre um mesmo objeto da MT define um "padrão" no qual este objeto deve se encaixar para instanciá-lo [GHALLAB 88].

A presença de variáveis permite que uma mesma condição seja testada entre diferentes objetos, acrescentando flexibilidade e expressividade à linguagem de representação. O escopo de uma variável limita-se ao corpo da regra, e na maioria dos sistemas são consideradas como quantificadas universalmente. Sistemas sem variáveis (proposicionais), são denominados de ordem 0, ao passo que sistemas com variáveis são ditos de ordem 1. Alguns poucos sistemas admitem o uso de variáveis também para os predicados. Neste caso se denominam sistemas de ordem 2, como por exemplo sistema Snark-2 [VIALATTE 84].

PARTE AÇÃO:

As ações presentes no LDR de uma regra podem ser, numa primeira análise, de quatro tipos:

(1) Ações sobre a MT: permitem a criação, modificação e deleção de objetos, provocando portanto alteração no estado do sistema;

(2) Ações sobre a MR: alguns sistemas admitem ações especiais que permitem a criação, deleção ou alteração de regras. Esta possibilidade permite dotar o sistema de mecanismos de aprendizagem (Learning) [CHARNIAK 86], [WINSTON 84a];

(3) Entrada/saída: proporcionam comunicação com ambientes externos, tais como os usuários do sistema, processos sob controle, etc.; e

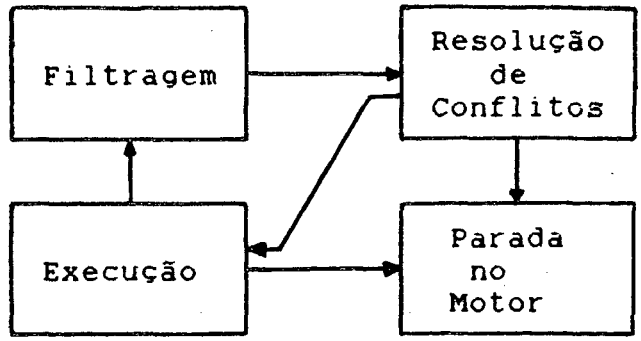
(4) Ações de controle: têm efeito direto sobre o MI, alterando seu comportamento.

De modo análogo ao caso dos objetos também aqui outros elementos podem estar associados às regras. Assim a cada regra pode estar associada: uma prioridade, fixando sua importância durante a execução, como no sistema Snark-2 [VIALATTE 84]; um fator de certeza, que caracteriza o grau de crença acerca da informação que representa, como nos sistemas Mycin [BUCHANAN 85], Expert [HAYES-ROTH 83] ou M.1 [HARMONN 85]; etc. Tais elementos usualmente têm papel importante na estratégia de controle empregada.

2.1.4 MOTOR DE INFERÊNCIA:

O MI atua de modo análogo aos interpretadores dos sistemas computacionais, e tem como principal tarefa decidir qual a próxima regra (ou regras) deve ser disparada.

O MI pode ser encarado como uma máquina de estado finito com três estados formando um laço e um estado final [BROWNSTON 86], conforme representado na figura abaixo:



O MI como máquina de estado finito

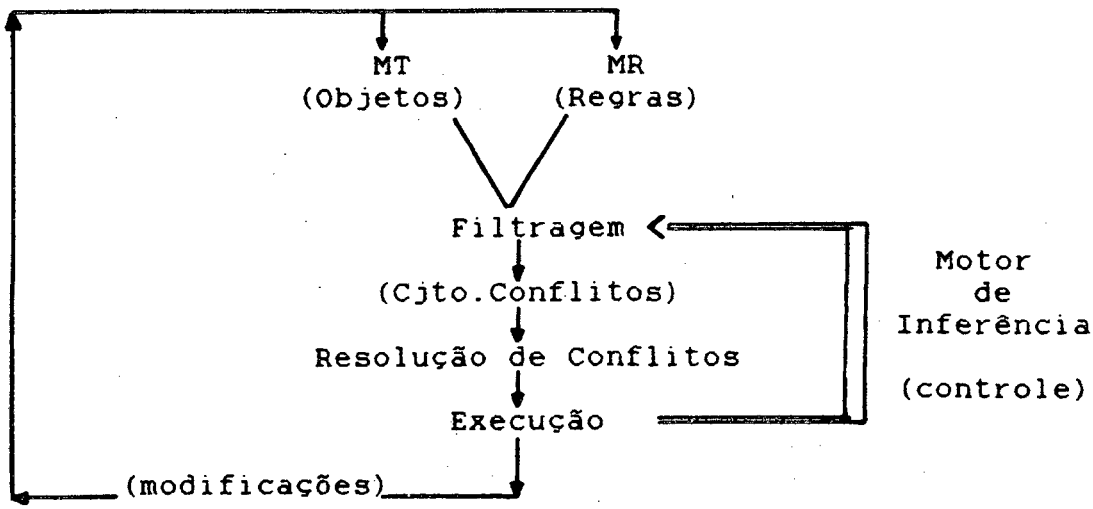
O funcionamento do MI segue o Ciclo de Base:

(1) Filtragem (Matching): através do casamento dos objetos com as regras são determinadas as regras satisfeitas pelo estado atual do sistema. O conjunto destas regras e dos objetos que as instanciam (instâncias de regras) constituem o conjunto de conflitos (CC). Uma mesma regra pode aparecer diversas vezes no CC, já que pode ser instanciada por várias n-uplas de objetos.

(2) Resolução de conflitos: dentre as instâncias selecionadas na etapa anterior, algumas são escolhidas para o disparo. Normalmente apenas uma é escolhida. Caso não haja nenhuma instância, o MI pode voltar a um estado anterior, de forma a efetuar a busca da solução por outro caminho. Esta etapa é não-determinista, pois a seleção da regra a ser aplicada é um problema aberto.

(3) Execução: as ações presentes no LDR da (s) regra (s) escolhida (s) são executadas, provocando alterações na MT e eventualmente modificações sobre MR, ações de entrada / saída ou ações de controle.

(4) Volta ao passo (1), a menos que uma ação (de controle) determine a parada do SP.



O Ciclo de Base do Motor de Inferência

FILTRAGEM:

A etapa de filtragem consiste na determinação de todas as instâncias de todas as regras da MR que são satisfeitas pelo estado da MT.

Devido ao acesso associativo às regras, esta etapa é crítica em termos de tempo, especialmente em sistemas com variáveis. Na maioria destes sistemas o casamento entre objetos e regras é feito por semi-unificação, uma forma particular da unificação utilizada em lógica predicativa (ver e.g. [MENDELSON 79] ou

[CASANOVA 87]), que permite variáveis nas regras mas não nos fatos. Poucas são as excessões, como por exemplo certas versões de PROLOG, que admitem variáveis tanto nas regras quanto nos objetos. Por abuso de linguagem o termo unificação será utilizado algumas vezes em lugar de casamento.

Diversos sistemas prevêem mecanismos para melhorar a eficiência desta etapa, com base na reestruturação da BC. A forma mais simples e natural consiste em particionar a BC em subconjuntos, de forma a reduzir o número de comparações necessárias (e.g. a árvore contextual do sistema Mycin [BUCHANAN 85]).

Mecanismos mais elaborados incluem a compilação das regras. Por compilação entenda-se a transformação de uma representação de entrada (as regras) em uma representação de saída (estruturas internas de apoio ao MI). Esta transformação é aqui acompanhada de uma reorganização das informações em estruturas de dados mais adaptadas ao tratamento pelo MI, reduzindo deste modo a complexidade da operação de unificação [GHALLAB 88].

ESTRUTURA DE CONTROLE DO SP:

A estrutura de controle do SP está baseada no não-determinismo da etapa de resolução de conflitos. Dois pontos principais deste controle são ressaltados:

(a) O regime de funcionamento:

O regime de funcionamento define a atitude a tomar quando o CC for vazio.

Em regime irrevogável, o MI simplesmente pára. Nesta situação o disparo de uma regra é feito sem qualquer previsão para uma eventual reconsideração de seus efeitos, o que torna crítica a escolha da regra a disparar.

Em regime por tentativas (com retrocesso ou backtrack) , o MI volta a um estado anterior do sistema e reconsidera a sua escolha neste estado disparando uma outra instância de regra. É de extrema importância a escolha do ponto de retorno (backtrack point), um estado ativável a partir do qual se dará a inferência [LAURENT 84]. No retrocesso (backtrack) sistemático (ou cronológico), o estado imediatamente anterior é restaurado. Este tipo de restituição é ineficiente já que se desconsideram os motivos que levaram ao insucesso [HAYES-ROTH 83]. Também é possível o estabelecimento de pontos de retorno pré-determinados ou mesmo dirigido pelos dados (backtrack seletivo).

(b) Critérios para a resolução de conflitos:

Durante o processo de inferência freqüentemente várias instâncias de regras estão em conflito. As opções de controle possíveis são:

- o disparo de todas as regras (paralelamente ou através de disparos sucessivos até a resolução do problema), como no sistema Mycin [BUCHANAN 85];
- escolha de uma instância para o disparo, com base em critérios de avaliação; e
- escolha de uma instância utilizando metaregras [DAVIS 80].

A escolha por avaliação das instâncias é o caso mais comum. Vários critérios são utilizados [BARR 86], como a escolha:

(1) da primeira regra disponível;

(2) da regra de mais alta prioridade;

(3) da regra mais específica, i.e., aquela composta por maior número de padrões;

(4) da instância que utiliza o elemento mais recentemente adicionado a MT;

(5) de uma regra que ainda não havia sido instanciada.

As metaregras são conhecimentos operatórios que especificam a forma de utilização das regras. Como as regras, as metaregras são selecionadas por filtragem, e proporcionam uma forma de controle mais flexível.

A estratégia de resolução de conflitos corresponde a principal ação de controle, sendo portanto fundamental para que se encontre a solução de uma forma rápida e eficiente. Ela afeta duas importantes características dos SP: (a) sensibilidade, que é a habilidade de responder rapidamente a mudanças do sistema; e (b) estabilidade, que é a capacidade do mesmo de concluir longas seqüências de ações [BARR 86].

RACIOCÍNIO MONOTÔNICO OU NÃO-MONOTÔNICO:

Este conceito discute se um fato estabelecido pode ou não ser contestado (*monotonia em lógica*, ver por exemplo [TURNER 86]).

Em raciocínio monotônico nenhum fato estabelecido pode ser retirado da MT. Geralmente os SP com esta característica utilizam raciocínio inexato e eventuais problemas de supressão ou contradição são contornados pelo valor dos fatores de certeza correspondentes. Sistemas onde há comutatividade ou reversibilidade das regras são naturalmente monotônicos [LAURENT 84], [NILSSON 80].

Nos sistemas monotônicos quando uma regra se torna válida, permanece válida durante todo o restante do processo de resolução, o que simplifica o controle.

Já os sistemas que utilizam raciocínio não-monotônico se caracterizam pela possibilidade de supressão de fatos pré-estabelecidos. Tais supressões podem ocorrer devido: ao surgimento de contradições; utilização de raciocínio por **default**, que posteriormente sofre revisões; na modelagem de ações que implicam em modificações físicas nos objetos que manipulam; etc.

Além disto, certos sistemas podem ser projetados de modo a suportar determinadas contradições: o uso de sistemas construídos com base em lógica paraconsistente constituem um exemplo de tais sistemas [BLAIR 88].

ENCADEAMENTO DAS REGRAS:

A aplicação das regras num SP pode ser vista como uma seqüência de mudanças de estado obtida por modus ponens (e no caso de variáveis nas regras, por especialização universal). As regras são aplicadas segundo dois métodos:

(a) No encadeamento para frente (forward chaining) as regras são aplicadas a partir do estado corrente do problema, descrito pela MT, a fim de produzir um novo estado, até que a condição de parada seja satisfeita.

(2) No encadeamento para trás (backward chaining) as regras são aplicadas não com base nos fatos presentes na MT, mas sim segundo o(s) problema(s) a resolver. O problema é então decomposto em um ou mais subproblemas cuja resolução, supõe-se, seja mais simples que o problema original.

A direção de encadeamento está fortemente ligada a estratégia de solução escolhida para o problema. Quando do encadeamento para frente a resolução é efetuada por busca de uma solução no espaço de estados, ao passo que em encadeamento para trás a resolução consiste em achar um sub-grafo de redução de problemas cujas folhas são fatos na MT [NILSSON 71], [FARRENY 87].

Sistemas com encadeamento para frente são mais adequados em situações em que há vários estados finais aceitáveis e poucos estados iniciais, onde portanto a explosão combinatória não é muito grande. Já o encadeamento para trás é mais conveniente em

tarefas tais como diagnose, onde um único objetivo deve ser atingido, e vários fatos iniciais estão presentes na MT.

PROCEDIMENTOS:

As vezes é fundamental a utilização, junto às regras, de procedimentos correspondentes a partes bem definidas do problema que se prestam a uma solução algorítmica. A maioria dos SP permite a chamada de procedimentos externos escritos numa linguagem de programação procedural a partir das regras (procedural atachíng) [FARRENY 87], [NILSSON 80].

DEMONS:

Os "demons" são conhecimentos operatórios análogos às regras, porém não sujeitos ao mesmo ciclo de controle. Quando um demon é instanciado, seu disparo ocorre automaticamente e imediatamente. Os demons aumentam a capacidade de reação dos SP, equivalendo às interrupções [FARRENY 87].

2.1.5 AMBIENTE DE PROGRAMAÇÃO E EXECUÇÃO:

O objetivo do ambiente de programação e execução é de auxiliar o usuário na edição, execução, monitoração, depuração e documentação de um programa. Este ambiente constitui o nível superior (top level) do sistema, e é centrado numa idéia básica: a interação. Esta interação é uma necessidade, pois o

desenvolvimento de programas no modelo SP é incremental, e não utiliza as metodologias tradicionais de programação [BARR 86].

Tal ambiente comportaria como módulos mais importantes:

Editor: orientado à linguagem de representação do conhecimento utilizada pelo SP, com verificação automática de sintaxe, dos tipos de dados e análise de consistência das informações.

Ambiente de execução: responsável pela interface com o usuário durante a execução do SP. Deve prover comandos que permitam a carga de BC's, o exame e alteração da MT e MR, a execução de inferências e modificações nos parâmetros do MI.

Mecanismos de monitoração e depuração: que devem ser providos de modo interativo.

Módulo de explanação: prove explanação e justificativa das conclusões obtidas pelo processo de inferência.

Módulo de aprendizagem: preocupa-se com a geração automática de novos conhecimentos a partir das situações descritas pelos dados.

2.1.6 SISTEMAS DE PRODUÇÃO E A PROGRAMAÇÃO CONVENCIONAL:

O modelo SP contrasta em vários aspectos com a programação convencional (algorítmica). Na tabela abaixo são resumidas suas principais diferenças [BARR 86], [WATERMAN 86].

Programação Convencional	Programação por SP
Informações acessíveis numericamente por endereços	Acesso as informações em modo associativo
Algoritmos	Busca heurística
Controle e conhecimento misturados	Separação relativa entre controle e conhecimento
Processamento numérico	Processamento simbólico
Modificações difíceis	Modificação, ampliação, atualização fácil
Resposta correta	Resposta satisfatória

2.1.7 VANTAGENS E DESVANTAGENS DOS SP:

Apesar da grande diversidade apresentada pelos SP quanto a expressividade de sua linguagem e opções de controle, algumas características lhes são bastante peculiares [BARR 86], [BROWNSTON 86]:

VANTAGENS:

Modularidade: é uma vantagem clara dos SP; como cada regra representa uma porção de conhecimento independente, pode ser adicionada, retirada ou modificada sem que isto produza outros efeitos sobre o sistema.

Uniformidade: como idealmente todo conhecimento operacional dos SP aparece no formato restrito das regras, há uma maior facilidade na compreensão do mesmo.

Aprendizagem: as duas características anteriores permitem a geração automática de regras, dotando o sistema de capacidade de aprendizagem.

Naturalidade: conhecimentos que determinam quais ações a tomar sob certas circunstâncias são os mais freqüentes nas tarefas de resolução de problemas: sua conversão à forma de regras de produção é imediata.

Paralelismo: a estrutura independente das regras e os variados caminhos emergentes durante o processo de inferência sugerem que os SP são adequados para a execução em sistemas de processamento paralelo.

DESVANTAGENS:

Ineficiência: os SP revelam-se ineficientes quando comparados aos sistemas tradicionais. A modularidade e uniformidade das regras

ocasiona o surgimento de grande número de alternativas a explorar, proporcionando grande contrapartida em termos de tempo de processamento. Em especial o processo de filtragem se revela extremamente custoso.

Opacidade: o funcionamento dos SP em modo associativo torna obscuro o fluxo de execução. Como o controle não é explícito, a prevenção de interações e loops indesejáveis é extremamente difícil.

2.1.8 ÁREAS DE APLICAÇÃO PRÓPRIAS AO USO DOS SP:

Os SP provêem um formalismo adequado à resolução de problemas mal-definidos ou de difícil expressão. São indicados os seguintes domínios [BARR 86], [BUCHANAN 85]: (a) áreas onde o conhecimento é esparso, em contraposição àquelas onde há uma teoria unificadora; (b) áreas onde o processo é representado por um conjunto de ações independentes ou pouco relacionadas; e (c) áreas onde o conhecimento pode ser facilmente separado da forma com que é utilizado.

A principal aplicação dos SP corresponde a geração dos SE: neste caso o engenheiro de conhecimento, responsável pela modelagem e construção do SE pode se concentrar na busca da estratégia de solução adequada ao problema e não precisa se preocupar com a criação de estruturas de dados e mecanismos de controle, que são providos diretamente pelo SP.

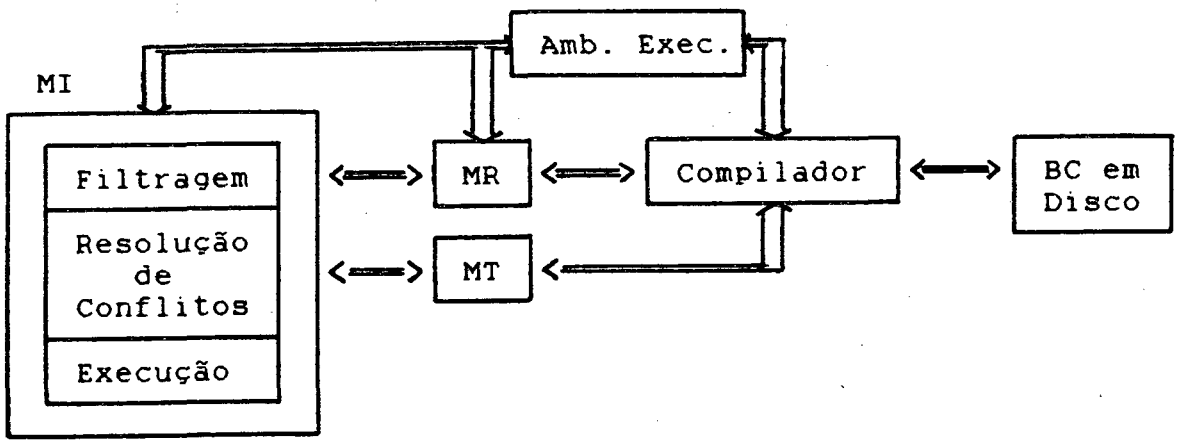
2.2 SP1 : UM SISTEMA DE PRODUÇÃO DE ORDEM 1

O SP1 (Sistema de Produção de Ordem 1), o objeto deste trabalho, é um SP de ordem 1, i.e., admite regras com variáveis quantificadas.

Sua linguagem para expressão do conhecimento foi especificada com constante preocupação quanto a sua legibilidade, sem que isto prejudique sua expressividade.

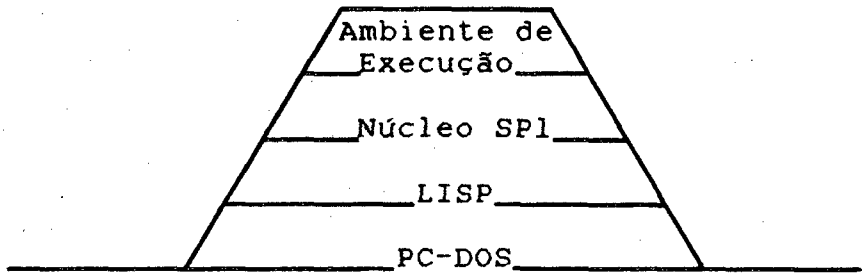
Por outro lado um eficiente algoritmo de filtragem foi especificado e implementado. Através da compilação das regras, o algoritmo constrói um "grafo de unificação" onde os objetos criados na MT são propagados e um "grafo de junções" que permite a avaliação de condições entre objetos diferentes. O aspecto incremental do compilador permite o acréscimo e deleção de regras.

O motor de inferência do sistema se caracteriza por sua versatilidade, provendo o usuário de diversas opções de controle. O usuário tem acesso ao sistema de modo interativo, através de uma ambiente de execução.



Componentes do sistema SP1

Uma versão protótipo do SP1 foi implementada sobre interpretador LISP, num ambiente compatível com PC. Foram então desenvolvidos o núcleo do sistema, composto dos procedimentos básicos para as três etapas do interpretador (filtragem, resolução de conflitos e execução), o compilador de regras e o ambiente de execução do sistema.



Camadas do Ambiente SP1

2.2.1 A LINGUAGEM SP1:

A linguagem para representação do conhecimento possui como características:

(a) definição dos objetos através de representação multi-atributo;

(b) associação de variáveis a objetos (funcionamento em ordem 1);

(c) ações básicas para criação, modificação e remoção de objetos;

(d) ações especiais para E/S e controle;

(e) capacidade de criação dinâmica de regras;

(f) possibilidade da incorporação de conhecimentos procedurais através do uso de funções;
(g) grande extensibilidade: funções e predicados podem ser definidos pelo usuário.

2.2.2 O ALGORITMO DE FILTRAGEM:

Como a operação de filtragem representa a grande fonte de ineficiência dos SP, o desenvolvimento do SP1 levou em conta grande preocupação com esta etapa.

Um novo algoritmo de filtragem foi desenvolvido [GARNOUSSET 88], [GARNOUSSET 89], baseado num compilador de regras, que permite reduzir de maneira importante o tempo de interpretação. O compilador transforma os antecedentes das regras numa rede que descreve as condições necessárias e suficientes para a sensibilização das mesmas, e é utilizada pelo interpretador (MI) para minimizar o tempo de geração de conflitos.

2.2.3 O MOTOR DE INFERÊNCIA:

O motor de inferência foi concebido de forma a proporcionar o máximo de flexibilidade, dentro de diversas técnicas; apresenta como características:
(a) operação em modo não-monotônico: estão presentes ações para modificação e deleção dos objetos;

(b) encadeamento para frente, com sensibilização das regras pelo seu LER a partir do estado corrente do sistema; execução das ações presentes no LDR de uma única regra escolhida;

(c) diversas estratégias para resolução de conflitos por avaliação, com utilização de prioridades e privilegiando fatos mais recentes;

(d) funcionamento tanto em regime revogável como por tentativas: foram implementados para execução de retrocesso (backtrack) sistemático e dirigido pelos dados;

(e) outras estruturas de controle: demons e funções.

2.2.4 O AMBIENTE DE EXECUÇÃO:

O ambiente de execução do SP1, através do qual o usuário tem acesso ao sistema, possui uma série de comandos interativos necessários às diversas etapas de resolução de um problema. Para tanto estão implementados mecanismos para execução, monitoração, depuração e armazenamento dos resultados para posterior análise.

2.3 CONCLUSÃO:

Neste capítulo é apresentado o modelo SP, um poderoso formalismo para representação e utilização do conhecimento. Os diversos componentes do modelo foram descritos de acordo com o que é classicamente encontrado.

Em seguida o SPl (objeto deste trabalho), foi sumariamente descrito, de forma a caracterizá-lo dentro dos conceitos apresentados.

III. A LINGUAGEM SP1

Este capítulo descreve a linguagem de programação SP1, especificada e implementada neste trabalho. O SP1 segue o paradigma Sistema de Produção, descrito no capítulo anterior. Suas diversas características são apresentadas a seguir, através de uma série de exemplos.

3.1 ARQUITETURA DO SP1:

Um programa SP1 é formado por um conjunto de regras de produção, constituindo a MR. As regras operam sobre uma base de dados global, a MT, que representa o estado do sistema. O controle está a cargo do MI, cuja operação obedece ao ciclo de base: filtragem, resolução de conflitos e execução.

O MI do SP1 funciona de forma não-monotônica, em encadeamento para frente, com sensibilização das regras pelo lado esquerdo. A cada ciclo apenas uma instância de regra é disparada. A escolha esta instância dentre as aplicáveis está baseada na prioridade e especificidade (número de condições) da regra e na anterioridade dos fatos que compõem a instância. O disparo da regra consiste na execução seqüencial das ações presentes no seu lado direito. São previstos mecanismos para restituição de estados, o que permite o funcionamento do SP1 em regime por tentativas. O funcionamento do MI é analisado em maiores detalhes no capítulo V.

3.2 ANÁLISE LÉXICA NO SP1:

O sistema léxico utilizado no SP1 é herdado da linguagem LISP utilizada para seu desenvolvimento. A entrada é feita segundo formato completamente livre: espaços, tabulações e novas linhas podem ser utilizadas para melhorar a legibilidade das regras.

3.3 CARACTERES MAIÚSCULOS E MINÚSCULOS:

A fim de aumentar a clareza e facilitar a compreensão e escrita das regras, o SP1 faz distinção entre caracteres maiúsculos e minúsculos. Palavras reservadas da linguagem devem ser escritas em maiúsculas. Os símbolos estão divididos em duas categorias, de acôrdo com sua primeira letra. Aqueles que iniciam por letras maiúsculas serão denominados por abuso de linguagem símbolos maiúsculos, enquanto os demais serão denominados símbolos minúsculos. Esta divisão foi feita tendo em vista a função sintática dos mesmos, e será explicada no decorrer do texto.

3.4 EDIÇÃO, TIPAGEM E VERIFICAÇÃO DE ERROS:

A edição de um programa no SP1 é feita usando um editor de textos padrão. O arquivo assim editado pode conter além das

regras de produção que constituem o programa em si, a definição de objetos.

Na versão atual o SP1 não faz qualquer verificação de tipos ou deteção de erros de consistência. Eventuais problemas podem ocorrer quando da execução do sistema, provocando erros detetados pelo interpretador LISP. Para arcar com estes problemas, está prevista a especificação e implementação de um editor orientado à linguagem, encarregado de verificações à medida da edição.

3.5 MEMÓRIA DE TRABALHO:

A MT no SP1 é constituída por um conjunto de objetos. Cada objeto representa uma instância de uma classe, munida de um conjunto de propriedades ou atributos. Um atributo representa uma característica do objeto e é definido por um identificador e seu domínio de valores.

O formalismo de representação dos objetos segue a forma multi-atributo utilizada no sistema OPS5 [FORGY 81]. Este formalismo foi escolhido por sua grande concisão e legibilidade.

Assim o objeto:

```
(Maquina  ^nome   m1
          ^tipo   furadeira
          ^eixo   vertical
          ^x      10
          ^y      20
          ^celula c1
          ^estado livre)
```

representa um objeto da classe das máquinas, indicada pelo primeiro elemento da lista, e cujas propriedades são indicadas por um conjunto de atributos, prefixados por ^ e seguidos pelos seus valores. Esta sintaxe é apresentada em maiores detalhes a seguir.

Seja C = { C1, C2, ...,Cn } um conjunto de classes de objetos, { aij } o conjunto dos atributos definidos para a classe Cj. Um objeto no SP1 tem a forma

(Cj ^a1j v1j ^a2j v2j ... ^apj vpj).

O identificador da classe (Cj) é um símbolo maiúsculo. O atributo (aij) vem prefixado pelo símbolo especial "^" é um símbolo minúsculo. O valor de qualquer atributo (vij) pode ser qualquer objeto LISP atômico (número ou símbolo) ou composto (lista ou vetor). As listas LISP são denominadas conjuntos no SP1, e são delimitadas pelos caracteres { e }. A fim de aumentar a expressividade da linguagem um atributo pode também ter como valor outro objeto SP1.

A cada objeto está associada uma marca de tempo, que indica a ordem de sua criação na MT. A relação bi-unívoca entre marcas de tempo e objetos define uma relação de ordem sobre MT, que é utilizada para fins de controle.

Não há a priori qualquer limitação sobre o tamanho da MT, ou sobre o número de atributos definidos para cada objeto.

São exemplos de objetos:

```
$3 : (Homem ^nome joao  ^pai jose ^cor_olhos azul ^peso 72.4)
$4 : (Homem ^nome pedro ^pai $3   ^altura 1.78)
$10: (Robo  ^nome r2d2  ^construtor acme_sa  ^serie {x 34})
```

$12: (Maquina ^tipo fresa ^serie al2 ^numero 32)

Os objetos $3 e $4 acima representam um caso típico de relacionamento hierárquico.

3.6 A MEMÓRIA DE REGRAS:

A memória de regras no SP1 é composta por um conjunto não-ordenado de elementos de dois tipos: regras de produção e demons.

3.6.1 AS REGRAS:

Uma regra no SP1 tem a forma:

(RULE <nome> { <prioridade> }
IF <LER>

THEN <LDR>)

onde:

<nome> é um símbolo minúsculo que identifica a regra;

<prioridade> é um inteiro opcional utilizado para fins de controle;

<LER> é uma conjunção de condições sobre objetos da MT;

<LDR> é formado por um conjunto de ações que serão executadas seqüencialmente quando do disparo da regra.

A cada regra está associado um conjunto de variáveis (*), que serão instanciadas por objetos durante os processos de filtragem e execução. O algoritmo utilizado para a instanciação é descrito no próximo capítulo. As variáveis são implicitamente quantificadas universalmente e tem seu escopo limitado ao corpo da regra.

Apresentamos a seguir uma regra do SP1, que referencia duas variáveis X e Y:

```
(RULE robo_pega_peca 3                  ;prioridade 3
      IF
          Robo(X)                       ;existe (na MT) um Robô
          Peca(Y)                       ;existe uma Peça
          garra(X) = vazia              ;o Robô nao segura nada
          acessivel(Y) = ok             ;a Peça está acessível
          capacidade(X) >= peso(Y)      ;a capacidade do Robô é
      THEN                              ;superior ao peso da Peça
          garra(X) = nome(Y)            ;o Robô pega a Peça
          acessivel(Y) = nao)           ;a Peça não está mais
                                        ;acessível
```

(*) Nota: A noção de variável difere da utilizada na maioria dos SP. Uma variável no SP1 representa uma instância de um objeto, enquanto codifica no OPS5, por exemplo, uma instância de um atributo deste objeto. A noção de variável no SP1 é equivalente a de variável-elemento no OPS5 [FORGY 81] ou ao "quark" do sistema SNARK [VIALATTE 84].

3.6.2 OS DEMONS:

Os demons são tipos especiais de regras, de sintaxe similar, porém sujeitos a um controle especial. Serão analisados posteriormente no capítulo V.

3.6.3 O Lado Esquerdo da Regra (LER):

O LER de uma regra é formado por uma conjunção de condições necessárias para a instanciação da regra.

Estas condições são de dois tipos:

(a) um conjunto de declarações que determinam a pertinência dos objetos que instanciam as variáveis a classes em C. Uma declaração da forma

Cj(Xj)

indica que a variável Xj só pode ser instanciada por objetos da classe Cj. Estas declarações devem ser as primeiras condições presentes no LER.

Por exemplo na regra acima as variáveis X e Y só poderão ser instanciadas por objetos das classes "Robo" e "Peca", respectivamente.

(b) um conjunto eventualmente vazio de condições, que estabelecem restrições adicionais a serem satisfeitas pelos objetos. As condições são da forma

aij(Xj) <p> <t>

onde:

aij(Xj) é um atributo da variável Xj;

<p> é um predicado do SP1 ou definido pelo usuário; e

<t> é um termo que serve como parâmetro para avaliação dos predicados.

As condições são convertidos quando da avaliação para a fórmula equivalente <p>(aij(Xj), <t>), onde o atributo indicado aparece como primeiro argumento do predicado.

ATRIBUTOS:

O valor de um atributo é obtido a partir do objeto que instancia a variável indicada (*). Por exemplo se o objeto

$8 : (Robo ^garra vazia ^capacidade 4.0)

instancia a variável X no contexto de uma regra, os atributos garra(X) e capacidade(X) tem os valores "vazia" e "4.0" respectivamente.

PREDICADOS:

Os predicados predefinidos pelo SP1 são:

(a) predicados sobre termos quaisquer: = <>

(b) predicados numéricos: > < >= <=

(*) Nota: Em certos casos o primeiro componente de uma condição pode ser um "atributo composto", ou seja, uma expressão do tipo akl(aij(Xj)). Esta composição é possível quando o valor de um atributo é um objeto. Por exemplo, considerando os objetos

$19 : (Homem ^nome joao ^pai jose)

$21 : (Homem ^nome pedro ^pai $19)

e supondo que o objeto $21 instancie a variável X, nome(X), nome(pai(X)) e pai(pai(X)) têm respectivamente os valores pedro, joao e jose.

(c) o predicado de pertinência: Member

(d) e os operadores lógicos: Not And Or.

Os operadores lógicos Not, And e Or escapam à forma geral apresentada acima, pois são n-ários a argumentos booleanos. Assim qualquer expressão construída a partir destes operadores será da forma

aij(Xj) <p'> (<p1> <t1>) (<p2> <t2>) ... (<pn> <tn>)

onde:

<p'> é um operador lógico Not, And ou Or;

<pi> são predicados quaisquer; e

<ti> são os termos correspondentes.

Esta forma será convertida para avaliação em

<p'>(<p1>(aij(Xj) <t1>) ... (<pn>(aij(Xj) <tn>)).

São exemplos de condições que incluem operadores lógicos:

cor(X) Not (= azul)

a1(Y) Or (= a2(Y)) (> 4)

a1(X) And (Or (> 1) (< 5)) (Member {2 3})

Embora esta notação não seja totalmente uniforme, foi adotada como a mais conveniente tanto em termos de clareza quanto devido a problemas de implementação.

A definição de predicados pelo usuário é feita diretamente em LISP, observando as restrições sintáticas: (a) o predicado deve ser binário; e (b) seu nome deve ser um símbolo maiúsculo.

TERMOS:

Um termo é definido no SP1 como sendo:

(a) uma constante (átomo numérico ou simbólico, conjunto, ou vetor) (*);

(b) um atributo (simples ou composto);

(c) uma função escalar n-ária e aplicada a n termos. Esta função será avaliada no sentido LISP usual sobre seus argumentos;

(d) uma função vetorial formada por um conjunto de funções coordenadas escalares;

(e) uma variável, que neste caso será substituída quando da avaliação pela marca de tempo do objeto que a instancia.

Qualquer função residente no ambiente pode ser diretamente utilizada. As mais usadas são: (a) as funções aritméticas +, -, *, /, **, abs, modulo; (b) as funções reais log, exp, log10, exp, power, sqrt; (c) as funções trigonométricas sin, cos, asin, acos, atan; e (d) as funções sobre listas car, cdr, cons, append, length, nth, union. São exemplos termos:

```
(+ 1 2 3)

(- 1.1  @pi)
(union {1 2 3} {4 5 6})
```

(*) Nota: Por convenção símbolos e expressões no SP1 não são avaliados (ao contrário do que ocorre em LISP). A avaliação é possível pela utilização do macro-caracter @ (operador de avaliação), e é feita durante a carga da BC. A função SET atribui a um símbolo seu valor. Por exemplo o comando (SET pi 3.141592) permite a utilização do símbolo @pi como termo, simplificando a redação das regras. A condição aij(Xj) = @pi será interpretado como aij(Xj) = 3.141592.

As funções vetoriais são obtidas a partir de conjuntos e funções escalares (eventualmente constantes). Por exemplo:

{ 1 (+ 1 2 3) (- 1.1 @pi) }

representa uma função vetorial composta pelas funções coordenadas f1 = 1 (função constante), f2 = (+ 1 2 3) e f3 = (+ 1.1 @pi).

A definição de funções pelo usuário deve ser feita segundo a sintaxe LISP, com as restrições: (a) seu nome deve ser um símbolo minúsculo; e (b) a função deve ser do tipo EXPR.

Se o termo é uma variável, o predicado é aplicado sobre a marca de tempo do objeto que instancia a variável no contexto da regra. Por exemplo a condição pai(Y) = X será verdadeiro para os objetos

$19 : (Homem ^nome joao ^pai jose)

$21 : (Homem ^nome pedro ^pai $19)

considerando que $19 e $20 instanciam respectivamente as variáveis X e Y.

3.6.4 NEGAÇÕES:

A designação da classe de uma variável pode estar precedida pelo símbolo de negação "-". Como as variáveis são consideradas universalmente quantificadas, a expressão $\forall$ X $\neg$C(X) equivale a $\neg \exists$ X C(X), ou seja, para a instanciação da regra nenhum objeto com as características indicadas pela variável pode pertencer a MT. Por exemplo a regra

```
(RULE ativa_se_nao_ha_problema
      IF
          - Problema(X)
      THEN     ...)
```

estará ativa a menos que um objeto de classe Problema esteja presente na MT.

Do mesmo modo a regra

```
(RULE robo_pega_peca_selecionada      ;prioridade 1
      IF
          Robo(X)                     ;existe um Robô
          Peca(Y)                     ;existe uma Peça
        - Peca(Z)                     ; de tipo beta
          tipo(Y) = beta              ;não há Peça
          tipo(Z) = alfa              ; de tipo alfa
          garra(X) = vazia            ;Robô não segura nada
          acessivel(Y) = ok           ;a Peça está acessível
          capacidade(X) >= peso(Y)    ;a capacidade do Robô é
      THEN                            ;maior que peso da Peça
          garra(X) = nome(Y)          ;Robô pega a Peça
          acessivel(Y) = nao)         ;Peça não está mais
                                      ;acessível
```

só estará ativa se, além de estarem presentes na MT objetos das classes "Robô" e "Peça" com os atributos desejados (e.g., "Peça" de tipo "beta") não houver na MT nenhum objeto da classe "Peça" cujo atributo tipo seja igual a "alfa".

3.6.5 O Lado Direito da Regra (LDR):

O LDR da regra é composto por um conjunto de ações que vão ser avaliadas incondicionalmente e seqüencialmente quando do disparo da regra. Estas ações podem ser: (a) ações de modificação da MT; (b) ações de controle, com efeito sobre o comportamento do MI; (c) entrada / saída; e (d) ações auxiliares.

AÇÕES SOBRE A MEMÓRIA DE TRABALHO:

As ações que têm efeito direto sobre a MT permitem o acréscimo de novos objetos e a modificação ou deleção de um objeto filtrado pelo LER da regra.

(a) Acréscimo de objeto:

Um novo objeto é criado na MT implicitamente pela colocação de uma nova variável no LDR, através de sua declaração de classe Ck(Xk). Após esta declaração os atributos do novo objeto criado são definidos através de uma seqüência de instruções de afetação, da forma aik(Xk) = <t> (onde Xk é a nova variável, i.e., não aparece no LER), que determinam o valor de seus atributos. Por exemplo a regra

```
(RULE robo_solda_pecas
      IF
          Robo(X)                    ; existe um Robô
          Peca(Y)                    ; e duas Peças
          Peca(Z)
          garra1(X) = Y              ; o Robô segura as
          garra2(X) = Z              ; Peças
      THEN
        - Y                          ; as Peças agora unidas
        - Z                          ; formam uma nova Peça
          Peca(W)
          tipo(W) = a20              ; atributos definidos
          acessivel(W) = ok          ; sobre esta Peça
          cor(W) = azul
          garra1(X) = vazia          ; esvaziar as garras
          garra2(X) = vazia)
```

cria um novo objeto de classe "Peca" e com os atributos "tipo", "acessivel" e "cor" tendo respectivamente os valores "a20", "ok" e "azul".

Outra possibilidade é o acréscimo à MT de objeto análogo ao que se encontra armazenado como atributo de outro objeto. Para tanto utilizou-se a sintaxe + Xk = aij(Xj). Por exemplo, sejam os objetos:

```
$10   : (Robo ^nome r2d2 ^capacidade 4.0 ^garra $7)
$7    : (Peca ^tipo alfa ^peso       2.2 ^numero 123)
```

O disparo da regra

```
(RULE robo_devolve_peca
      IF
          Robo(X)                    ; existe um Robô
          garra(X) <> vazia          ; que segura uma Peça
      THEN
          cor(garra(X)) = azul       ; O Robô pinta e
        + Y = garra(X)               ; devolve a Peça
          garra(X) = vazia)
```

causará a restituição (cópia) ,do objeto da classe "Peca" à MT, embora com uma nova marca_de_tempo associada. Esta possibilidade acrescenta potencialidade ao sistema, já que todas as características do objeto são preservadas (não é necessário declarar todos os atributos do novo objeto).

(b) Modificação de atributos:

Um objeto que instancia uma variável no LER de uma regra pode ser modificado a partir de declarações presentes no LDR desta regra. Estas ações são efetuadas de acordo com a sintaxe aij(Xj) = <t>, onde Xj é uma variável aparecendo no LER.

Na regra abaixo, o atributo garra do objeto de classe Robô tem seu valor modificado de "vazia" para a o objeto que instancia a variável Y.

```
(RULE robo_pega_peca
      IF
          Robo(X)                    ; existe um Robô
          Peca(Y)                    ; e uma Peça
          garra(X) = vazia           ; o Robô não segura nada
          acessivel(Y) = ok          ; a Peça está acessível
      THEN
          garra(X) = Y)              ; o Robô pega a Peça
```

Modificações compostas podem ocorrer. Por exemplo, o objeto armazenado no atributo garra da variável X pode ter seu atributo "côr" modificado para "azul" conforme indicado na regra abaixo:

```
(RULE robo_pinta_peca
      IF
          Robo(X)                    ; o Robô segura uma Peça
          garra(X) <> vazia
      THEN
          cor(garra(X)) = azul)      ; e a pinta de "azul"
```

(c) Deleção de um objeto:

Um objeto que instancia uma variável no LER de uma regra pode ser retirado da MT. Esta ação é declarada prefixando a variável correspondente pelo símbolo "-". Na regra robo_pega_peca_2 abaixo, o objeto de classe "Peca" que instancia a variável Y é retirado da MT pelo disparo da regra.

```
(RULE robo_pega_peca_2
      IF
          Robo(X)                    ; existe um Robô
          Peca(Y)                    ; e uma Peça
          garra(X) = vazia           ; a garra está livre
          acessivel(Y) = ok          ; e a Peça acessível
      THEN
          garra(X) = Y               ; o Robô pega a Peça
        - Y)                         ; (objeto Peça
                                     ; removido da MT)
```

AÇÕES DE CONTROLE:

As ações de controle afetam diretamente o comportamento do MI cujo funcionamento é descrito em detalhes no capítulo V.

(a) STOP:

A ação STOP ocasiona a parada definitiva do MI. São apresentadas informações relativas às regras disparadas, e número de ciclos e retrocessos (backtracks) efetuados.

(b) HALT:

A ação HALT provoca a parada temporária do MI. Difere do STOP no sentido que o MI pode ser relançado a partir deste ponto por comando do usuário. Esta ação é particularmente interessante na fase de depuração.

(c) PRIORITY:

A ação PRIORITY permite alterar o campo prioridade da regra cujo nome é fornecido como argumento.

(d) EXCISE:

Permite desativar uma regra.

(e) REVIVE:

Permite reativar uma regra com uma prioridade default.

(f) HARAKIRI:

Torna a regra inativa logo após seu disparo.

(g) RETURN-OVER:

A ação RETURN-OVER permite realizar um retrocesso dirigido pelos dados. Corresponde a ação "revenir-sur" do sistema SNARK [VIALATTE 84]. O argumento fornecido é uma variável presente no LER da regra. Sua avaliação provoca a restituição do estado do sistema quando da criação do objeto que instancia esta variável. Por exemplo o disparo da regra abaixo fará com que o sistema retorne ao estado correspondente ao aparecimento do objeto que instancia a variável X:

```
(RULE retorno_a_ponto_chave
      IF
          Decisao(X)
          Problema(Y)
          estado(Y) = insoluvel
      THEN
          (RETURN-OVER X))
```

ENTRADA / SAÍDA:

As entradas/saídas fornecem meios de comunicação com o usuário. No SP1 duas funções básicas foram implementadas: accept e WRITE.

(a) accept:

A entrada de dados é realizada através da função accept. É especialmente utilizada para a atribuição de parâmetros fornecidos pelo usuário. Por exemplo, a regra abaixo solicita ao usuário a entrada de dois valores que são utilizados posteriormente pelo sistema.

```
(RULE inicio
      IF
          Comeco(X)
      THEN
          Numero(Y)
          Numero(Z)
          valor(Y) = (accept "Entre um numero nao negativo")
          valor(Z) = (accept "Entre um segundo numero"))
```

(b) WRITE:

A ação WRITE permite a apresentação de resultados ao usuário. A regra abaixo exibe como saída o atributo valor da variável X.

```
(DEMON fim
        IF
            Numero(X)
            Numero(Y)
            valor(Y) = 0
        THEN
            (WRITE "mdc = " valor(X))
            (STOP))
```

AÇÕES AUXILIARES:

(a) BIND:

A ação BIND (*) permite atribuir um valor a um símbolo. Com isto é possível o armazenamento de valores intermediários para posterior uso dentro da regra. Por exemplo a regra abaixo atribui o mesmo valor aos atributos nome(Y) e prox(X):

```
(RULE coloca_na_lista
      IF  No(X)
          prox(X) = nil
      THEN
          (BIND Z (gensym))
          No(Y)
          prox(Y) = nil
          nome(Y) = Z
          prox(X) = nome(Y) )
```

(*) Nota: Na implementação atual o uso desta ação é necessário para se obter a seqüencialidade quando do uso de funções LISP físicas (ver exemplo 9, apêndice C).

(b) BUILD:

A ação BUILD permite a criação dinâmica de uma nova regra. Inicialmente os elementos já definidos presentes no corpo do BUILD são substituídos por seus valores. A regra assim formada é então compilada e incorporada às estruturas já existentes. Esta facilidade permite dotar o sistema de capacidade de aprendizagem (learning) [WINSTON 84a], [CHARNIAK 86]. Por exemplo, seja a regra:

```
(RULE cria_outra
      IF
          C1(X)
          a1(X) = a2(X)
      THEN
          (BUILD (RULE nova_regra
                       IF
                           C2(Y)
                           a1(Y) = a3(X)
                       THEN
                           (STOP)  )))
```

e o objeto

$3 : (C1 ^a1 1 ^a2 1 ^a3 3)

o disparo desta regra instanciada pelo objeto $3 acresce a seguinte regra à MT:

```
(RULE nova_regra
      IF
          C2(Y)
          a1(Y) = 3
      THEN
          (STOP))
```

3.7 CONCLUSÃO:

A linguagem SP1 foi concebida tendo como principal objetivo a facilidade de redação, e sua legibilidade. Para os objetos a notação multi-atributo foi utilizada pela sua concisão e potencialidade. Para as regras optou-se pela associação de variáveis diretamente aos objetos, de forma que a representação de um atributo fosse ao mesmo tempo simples e natural, além de proporcionar maior clareza às condições.

A expressividade da linguagem é aumentada pela possibilidade de utilização de atributos compostos, qualidade muito útil na modelagem de diversos problemas.

Além destas características destacam-se a possibilidade de criação dinâmica de regras e a grande extensibilidade da linguagem, pois conhecimentos procedurais, funções e predicados podem ser implementados diretamente pelo usuário através de funções LISP.

IV. O ALGORITMO DE FILTRAGEM

A determinação das regras aplicáveis ao estado corrente do sistema (Pattern-Matching) é a fase mais penalizante do funcionamento de um SP, e pode ocupar até 90 % tempo de execução do sistema [GHALLAB 84], [SOBEK 87]. Dela resulta o desempenho medíocre dos SP, que limita seu uso quando restrições temporais condicionam o problema.

A complexidade da operação de filtragem para o algoritmo trivial, i.e., tentativa exaustiva de unificação entre objetos e regras, é O (no.nr.c), onde no é o número de objetos presentes na MT, nr é o número de regras na MR e c é o número médio de condições por regra.

Neste capítulo serão inicialmente apresentados os diversos mecanismos utilizados para reduzir a complexidade da unificação. Em seguida apresentar-se-á a abordagem utilizada no SP1, que aparenta-se aos métodos de redes, e consiste na compilação das condições das regras em estruturas equivalentes (grafo de unificação e grafo de junções) nas quais várias redundâncias são tornadas explícitas a fim de otimizar os testes a efetuar.

4.1 ABORDAGEM DO PROBLEMA:

A ineficiência do processo de unificação foi notada logo que desenvolvidos os primeiros sistemas e vários pesquisadores propuseram soluções para reduzí-la.

A abordagem mais natural consiste em melhorar a eficiência do processo considerando um subconjunto de regras e/ou objetos candidatos a unificação (partição da BC). A obtenção de um subconjunto de objetos é efetuada enfatizando elementos peculiares em função de critérios heurísticos. Por outro lado a MR é limitada às regras vinculadas ao contexto corrente. Esta abordagem leva então a partilhar a BC do sistema em sub-conjuntos específicos. A árvore contextual do sistema MYCIN, por exemplo, define uma partição da MR em grupos de regras específicas aplicáveis dentro de um contexto dado [BUCHANAN 85].

Ainda que melhorem de forma sensível o desempenho temporal do sistema, esta abordagem não diminui a complexidade da unificação. Por outro lado impõe a introdução de informações de controle, contra a filosofia básica de um SP.

Uma maior eficiência pode ser obtida considerando a hipótese que o conhecimento (MT e MR) sofre alterações mínimas entre duas inferências [GHALLAB 84]. Esta hipótese repousa sobre duas constatações válidas na maioria dos problemas: (a) a cada ciclo do sistema as modificações ocorridas na MT são bem inferiores a seu tamanho; e (b) a MR quase nunca é alterada.

Isto permite concluir que a grande maioria das regras aplicáveis no ciclo t é ainda aplicável no ciclo $t + 1$. Como conseqüencia não há necessidade de se redeterminar o CC todo a cada ciclo. É suficiente que seja atualizado convenientemente em função das modificações produzidas em MT e MR.

Duas grandes famílias de métodos baseiam-se neste princípio: (a) "métodos de filtragem e interpretação"; e (b) "métodos de rede".

Nos métodos de filtragem e interpretação, um conjunto de condições necessárias para a validade das regras é definido. Estas condições não são, no entanto, suficientes. As regras filtradas são então comparadas diretamente com os dados presentes na MT para a definição final do conjunto de conflitos. O MI do sistema HSP utilizado para uma implementação do sistema Hearsay-II é um exemplo deste tipo de abordagem [McCRACKEN 78]. Estes métodos, embora tragam melhoria em relação à interpretação direta, ficam ainda muito penalizantes por causa da etapa de interpretação final.

Nos métodos de rede, a filtragem não conduz apenas a uma condição necessária de validade, mas diretamente a uma condição necessária e suficiente. Utilizam para tanto a compilação das regras: procuram a diminuição da complexidade por eliminação de redundâncias e reordenação de testes, além de prover estruturas de apoio que tornam o processo de unificação mais eficaz para o interpretador.

O sistema de produção OPS5 possui um dos mais eficientes algoritmos de unificação. Seu compilador de regras RETE [FORGY 82] constrói uma rede em dois tempos: inicialmente cada padrão é transformado numa seqüência linear de testes sobre os atributos (árvore de unificação). As ligações entre diferentes padrões, que correspondem no OPS5 a existência de uma variável entre dois padrões são então compiladas sob a forma de um conjunto de testes

(junções) entre as características compartilhadas. O compilador do OPS5 busca a eliminação de redundâncias quando da construção da rede. Entretanto, numerosas duplicações subsistem, devido à seqüencialidade dos testes. O motor de inferência PSC [GHALLAB 84] procura remediar estas ineficiências pela utilização de uma árvore de decisão quasi-ótima [GHALLAB 81] que discrimina os padrões não mais de forma linear, mas segundo um algoritmo que procura otimizar a propagação. Em seguida os nós da rede são compostos através de junções. Como a árvore de decisão é definida a partir da MR inicial, eventuais modificações na mesma podem comprometer sua otimalidade.

4.2 O ALGORITMO DE FILTRAGEM DO SP1:

O algoritmo de filtragem desenvolvido e implementado no sistema SP1 pertence aos métodos de rede [GARNOUSSET 88], [GARNOUSSET 89] . Seu objetivo é reduzir substancialmente o número de testes efetuados para instanciação das regras, através da construção de uma rede por compilação dos antecedentes das regras. Diversos tipos de redundâncias estruturais são evitadas. A rede é utilizada para propagar as modificações produzidas na MT e atualizar as instâncias das regras. A principal diferença deste algoritmo com os já citados é que a instanciação de um padrão efetua-se por propagação dos objetos segundo vários caminhos. Este mecanismo de propagação permite aumentar a eficiência do algoritmo especialmente no caso de modificações

parciais. Devido ao aspecto incremental da compilação, novas regras podem ser criadas dinamicamente.

4.2.1 A COMPILAÇÃO DAS REGRAS:

O compilador de regras do SP1 constrói: (a) um "grafo de unificação", que descreve as condições necessárias e suficientes para instanciação das variáveis (ver a noção de variável no ítem 3.6.1) e no qual são propagadas individualmente as modificações produzidas na memória; (b) um "grafo de junções", do tipo E / OU, que permite a avaliação de predicados sobre objetos diferentes e que descreve as condições necessárias e suficientes para aplicação das regras. É a partir do grafo de junções que o interpretador efetua a combinação das instâncias de variáveis a fim de determinar as instâncias de regras que constituirão o CC.

A seguir é apresentada a construção dos grafos de unificação e de junções a partir das três regras seguintes:

```
(RULE r1                          (RULE r2
      IF   C1(X)                        IF   C1(X)
           C2(Y)                             C2(Y)
           a1(X) = 1                         a1(X) >= 1
           a2(X) = a4(Y)                     a1(Y) = 2
           a3(X) > a1(Y)                     a4(Y) Or (= a3(X))
           a2(Y) > a1(Y)                              (> a2(Y))
      THEN                              THEN
           ...)                              ...)
```

```
(RULE r3
      IF   C2(X)
           a2(X) > a1(X)
      THEN
           ... = a4(X)
           ...)
```

4.2.2 INSTANCIAÇÃO DE UMA VARIÁVEL:

Devido à representação multi-atributo utilizada pelo SP1 para os objetos, é importante definir claramente a noção de identidade estrutural entre um objeto e um padrão (conjunto de condições sobre a mesma variável). Nesta unificação nenhuma relação de ordem entre atributos é utilizada, sendo suficiente que os atributos mencionados no LER estejam presentes no objeto em questão.

CONDIÇÃO NECESSÁRIA:

Para que um objeto possa instanciar uma variável, as seguintes condições necessárias devem ser sucessivamente satisfeitas:

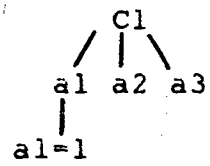
(a) O objeto deve ser da mesma classe que a variável correspondente;

(b) Os atributos presentes no antecedente da regra ou que são mencionados no conseqüente à direita do símbolo de atribuição "=" devem estar todos definidos no objeto testado. Nas três regras

que compõem o exemplo, é assumida a hipótese que não há outro atributo aparecendo a direita de um "=" além dos mencionados.
(c) Quando o termo que aparece numa condição é uma constante, o atributo mencionado deve verificar o predicado correspondente.

Por exemplo, o objeto (C1 ^a1 1 ^a2 3 ^a3 5 ^a4 2) pode instanciar a variável X da regra r1, já que: (a) é um objeto da classe C1; (b) os atributos a1, a2 e a3 estão nele definidos; e (c) seu atributo a3 é igual a 1. O mesmo não ocorre com o objeto (C1 ^a1 1 ^a2 2), pois o atributo a3 não está definido neste objeto e portanto a terceira condição de r1 não poderá ser avaliada.

Cada uma das condições necessárias a instanciação de uma variável pode ser representada sob a forma de uma árvore, como a indicada abaixo para a variável X da regra r1.



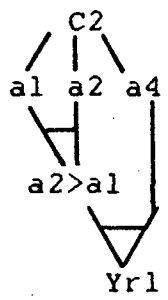
CONDIÇÃO NECESSÁRIA E SUFICIENTE:

Um objeto instancia uma variável se e somente se todas as condições necessárias descritas acima são verificadas e todos os testes entre atributos desta mesma variável são também verificados.

Por exemplo, o objeto (C2 ^a1 1 ^a2 1 ^a4 2) verifica as condições necessárias sobre a variável Y da regra r1. Entretanto,

este objeto não verifica a quarta condição desta regra (a2(Y) > a1(Y)) entre dois atributos da mesma variável.

Esta condição pode ser representada, por exemplo para a variável Y da regra r1, sob a forma do grafo apresentado a seguir. Utilizaremos a notação Xri para designar a variável X da regra ri.

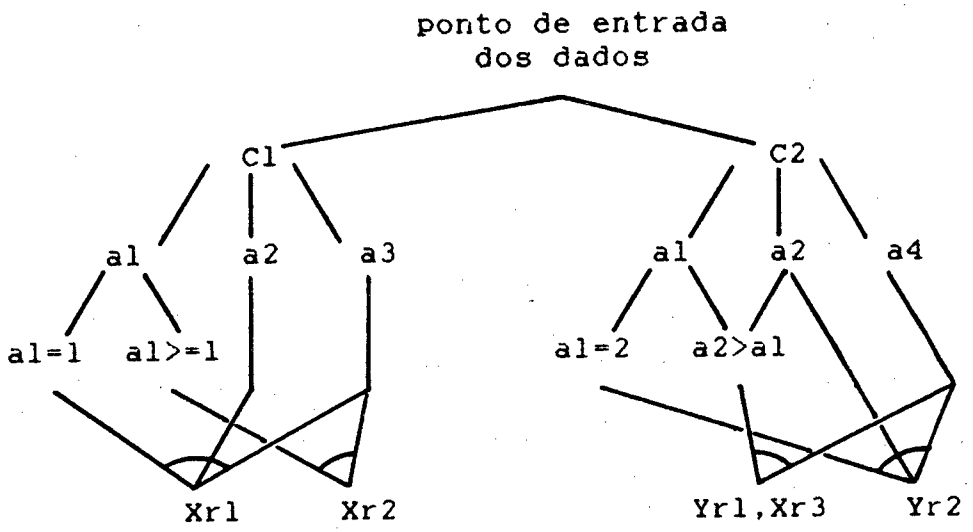


GRAFO DE UNIFICAÇÃO:

O algoritmo de compilação implementado procura otimizar os testes a efetuar para a determinação das instâncias das variáveis.

Inicialmente, observa-se que numerosas condições aparecem sob a mesma forma em regras distintas, de forma que sua avaliação sobre um mesmo objeto pode ser efetuada apenas uma vez. Por exemplo, a instanciação da variável Y de r1 e da variável X de r3 são totalmente idênticas. Estas redundâncias são denominadas estruturais, e são detetadas pelo SP1 independentemente da ordem de aparecimento das condições que compõem a regra.

O conjunto de todas os padrões vai ser compilado sob a forma de grafo de unificação, apresentado abaixo.



4.2.3 INSTANCIAÇÃO DE UMA REGRA:

CONDIÇÃO NECESSÁRIA:

Esta condição corresponde a existência de uma instância associada a cada uma das variáveis que constituem o antecedente da regra.

Por exemplo, a regra r1 é instanciável no momento que as variáveis X e Y são respectivamente instanciadas pelos objetos:

$2 : (C1 ^a1 1 ^a2 3 ^a3 5 ^a4 2) e

$3 : (C2 ^a1 1 ^a2 2 ^a4 3)

Esta condição pode ser representada pelo grafo:

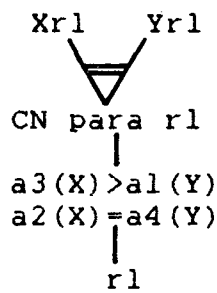
CONDIÇÃO NECESSÁRIA E SUFICIENTE:

Toda instância de uma regra que contém variáveis no seu antecedente é uma n-upla onde cada componente instancia o padrão correspondente.

Quando uma n-upla verifica a condição precedente basta avaliar as condições referentes a variáveis distintas, que são as únicas restantes, para determinar se esta n-upla constitui uma instância da regra. Estas condições constituem as denominadas "junções".

Por exemplo, o par constituído pelos objetos $2 e $3 anteriormente descritos e que instanciam respectivamente as variáveis X e Y da regra r1 verificam as duas restrições de junção entre X e Y: a2(X)=a4(Y) e a3(X)>a1(Y). Portanto este par de objetos instancia a regra r1.

Esta condição pode ser visualizada da seguinte forma:



Os testes entre padrões diferentes são compilados na forma de um grafo de junções, como será apresentado a seguir.

O GRAFO DE JUNÇÕES:

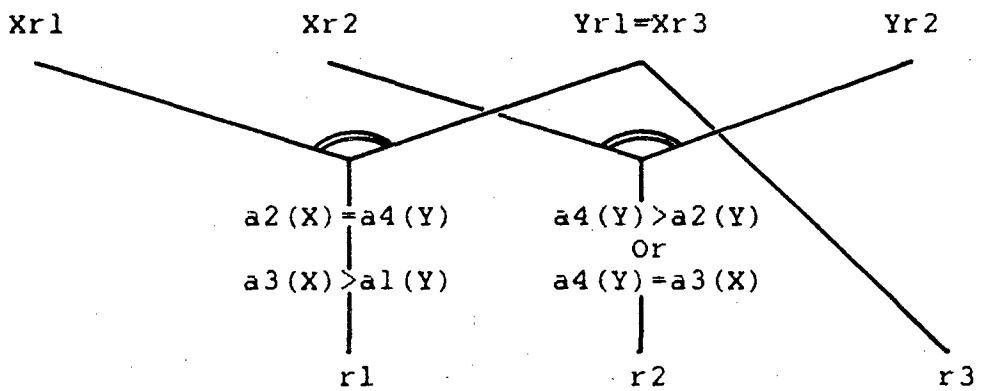
De forma análoga a instanciação das variáveis, o algoritmo de compilação também busca otimizar os testes a efetuar para a instanciação das regras. Quando vários testes devem ser efetuados para uma junção, eles são reordenados de modo a que a avaliação seja a mais rápida possível. Para tanto os testes mais restritivos são realizados em primeiro lugar. Esta estratégia de avaliação permite concluir mais rapidamente sobre a invalidade de uma junção. Por exemplo os testes para junção entre as instancias de X e Y na regra r1 serão reordenados sob a forma :

a2(X) = a4(Y)

a3(Y) > a1(Y)

Com efeito, o primeiro teste tem maior chance de falhar que o segundo (um teste de igualdade é mais restritivo que o teste de pertinência a um conjunto) e portanto a junção será, em média, avaliada mais rapidamente.

Considerando as descrições anteriores o compilador do SP1 vai construir o grafo de junções indicada abaixo:



4.2.4 O CASO DOS PADRÕES NEGADOS:

Quando uma regra possui um padrão negado, dois casos podem ser analisados:

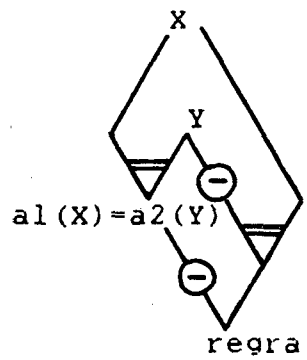
(1) A variável associada a este padrão não está ligada a nenhuma outra variável, através de seus atributos. Neste caso é suficiente que nenhum objeto instancie a variável para que a regra possa ser instanciada. Esta condição é representada por uma marca de negação sobre o arco que sai desta variável. Esta marca tem por efeito a transmissão de "falso" ou "verdadeiro" a seus sucessores, correspondendo a existência ou não de uma instância.

```
IF      C1(X)
     - C2(Y)
THEN ...
```

(2) A variável associada a este padrão está ligada a pelo menos uma outra variável no antecedente. A regra correspondente poderá ser instanciada num dos dois casos seguintes: (a) se nenhum objeto instancia a variável; ou (b) o conjunto de restrições de junção com os outros padrões da regra não são verificadas.

Isto é representado na rede por um nó do tipo OU entre as duas condições como ilustrado abaixo:

```
IF      C1(X)
     - C2(Y)
       a1(X) = a2(Y)
THEN ...
```



4.2.5 O CASO DOS ATRIBUTOS COMPOSTOS:

No caso de atributos compostos, o algoritmo de filtragem do SPl considera apenas o primeiro atributo referido. Por exemplo, seja o atributo composto cor(garra(X)): o algoritmo apenas testará a existência do atributo "garra" sobre o objeto candidato a instanciação da variável X. A operação de unificação sobre o segundo atributo (e outros de ordem superior) será feita por avaliação direta (sem testar antecipadamente a existência do atributo). Esta simplificação foi adotada para reduzir a complexidade da implementação do algoritmo.

4.3 UTILIZAÇÃO DA REDE:

A rede construída pelo compilador vai ser utilizada pelo MI para a determinação do conjunto de regras candidatas ao disparo, i.e., o conjunto de conflitos. A cada ciclo do sistema o CC não é inteiramente reavaliado, mas apenas atualizado a partir das modificações produzidas na MT e das informações armazenadas na rede. Isto evita as "redundâncias temporais". A cada nó da rede está associada uma lista dos objetos que o instanciam. Esta lista é alterada apenas quando necessário, de modo que a cada ciclo poucos são os nós que têm sua lista alterada.

A propagação de um objeto é feita em duas etapas: (a) sua propagação até as instâncias das variáveis no grafo de unificação; e (b) a atualização das instâncias das regras por propagação no grafo de junções.

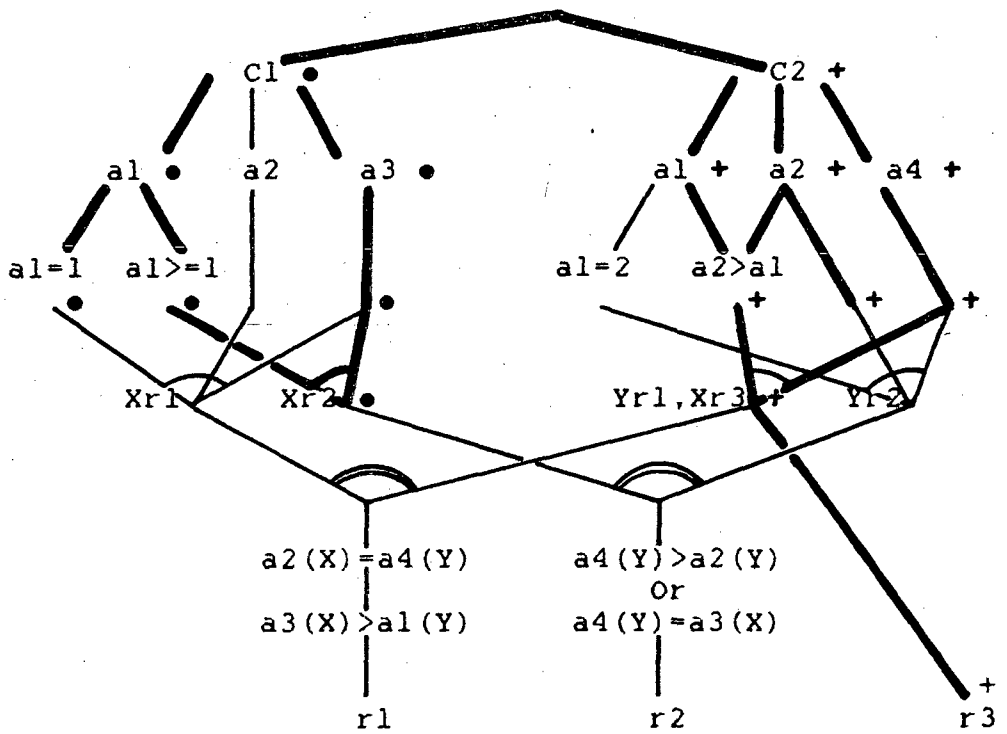
Todo objeto adicionado a MT que atinge um nó da rede é memorizado na lista associada ao nó. A deleção de um objeto da MT provoca sua retirada também das listas associadas em que apareça. Toda modificação de uma instância de variável é propagada através das junções no grafo correspondente até que as instâncias das regras sejam obtidas.

Para ilustrar o mecanismo de propagação, consideremos os objetos:

$1 : (C1 ^a1 1 ^a3 1), indicado por o, e

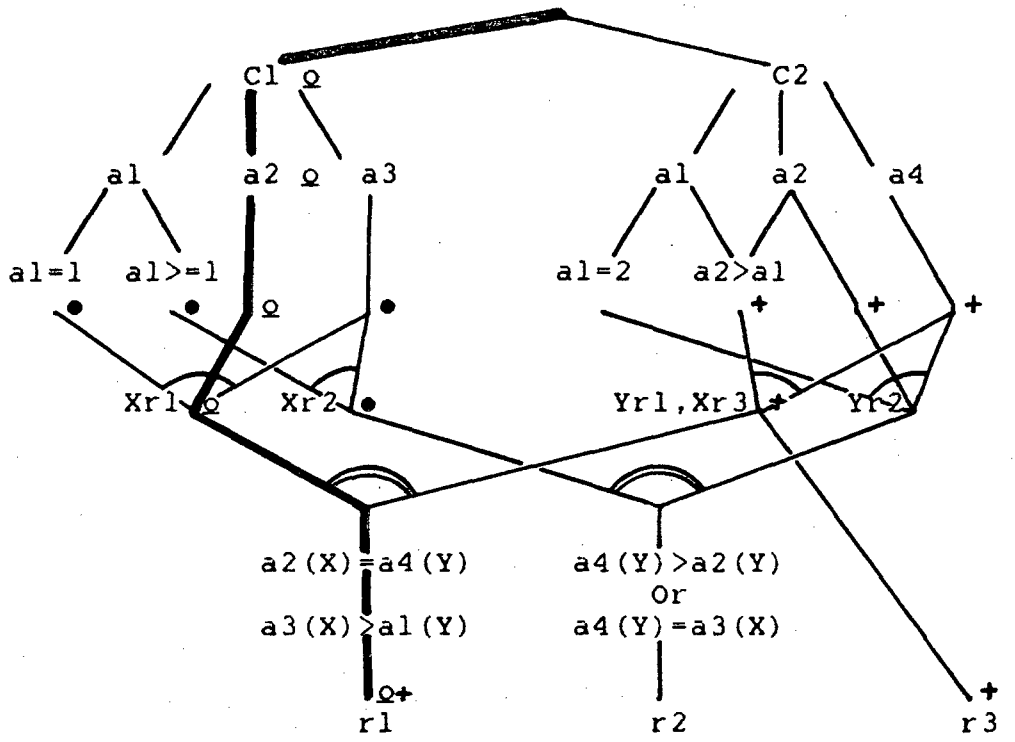
$2 : (C2 ^a1 1 ^a2 2 ^a4 1), indicado por +.

A propagação destes objetos na rede é mostrada a seguir:



Uma modificação sobre o objeto $1 que inclui o atributo a2 = 1, i.e., transforma $1 em $3 : (C1 ^a1 1 ^a2 1 ^a3 1), será

propagada como indicado abaixo pelo sinal o. Os nós da rede indicados por o e + já se encontravam instanciados.



O conjunto de conflitos obtido tem no SP1 a forma

((<nome> { (o1 o2 ... oN) }+)*)

onde:

<nome> : é o nome da regra;

(o1 o2 ... oN) é a n-upla que instancia a regra, formada pelas marcas de tempo do objeto oi que instancia a i-ésima variável da regra, ou pelo símbolo "-" no caso dos padrões negados.

No exemplo citado o CC obtido será ((r1 ($3 $2)) (r3 ($2)).

4.4 CONCLUSÃO:

O algoritmo de unificação desenvolvido no SP1 mantém o CC por propagação incremental das mudanças ocorridas na MT quando do disparo das regras. A eficiência deste algoritmo provém da compilação dos antecedentes das regras num conjunto de estruturas equivalentes que evita a redundância dos testes a nível das condições e dos padrões, e diminui o custo da instanciação pela ordenação dos mesmos. O caráter incremental da geração destas estruturas permite o acréscimo dinâmico de novas regras durante a inferência, sem prejuízo da eficiência do algoritmo.

V. O MOTOR DE INFERÊNCIA

Este capítulo apresenta o motor de inferência do SP1. Ele é caracterizado por sua grande flexibilidade de configuração, proporcionando diversas opções que devem ser selecionadas em função da aplicação que se pretende resolver.

Esta configuração é realizada pelo usuário no ambiente de execução do SP1, através de uma seqüência de comandos interativos (ver próximo capítulo).

Devido ao aspecto incremental de determinação das regras candidatas ao disparo, o ciclo de base do SP1 é distinto do descrito no capítulo I. Suas etapas são:

(a) Resolução de conflitos: escolha de uma regra candidata ao disparo dentro do conjunto de regras em conflito;

(b) Execução: disparo da regra escolhida. Ocorre parada do MI, caso a ação correspondente tenha sido executada;

(c) Filtragem: avaliação do antecedente das regras para determinar quais destas são satisfeitas pelo estado de MT; em seguida o sistema volta a executar o passo (a).

Esta alteração na seqüência das etapas acima é feita porque a operação de filtragem ocorre como efeito colateral da etapa de execução, e faz com que o conjunto de conflitos se torne consistente com o conteúdo corrente da MT.

A estrutura de controle do SP1 segue a "escolha por avaliação", e portanto está centrada na etapa de resolução de conflitos.

A seguir analisaremos estas diversas etapas do MI e as opções disponíveis.

5.1 A RESOLUÇÃO DE CONFLITOS:

A etapa de resolução de conflitos (que constitui a estrutura de controle do SP1) tem por objetivo determinar dentro do conjunto de regras em conflito, resultante da etapa de filtragem, uma regra para disparo. Cada instância de uma regra apresenta-se na forma

(<nome> (o1 o2 ... oN)),

onde:

<nome> é o nome da regra; e

(o1 o2 ... oN) é a n-upla que instancia a regra, formada pelas marcas de tempo dos objetos correspondentes, ou eventualmente pelo símbolo "-" para as premissas negativas.

Os critérios utilizados pelo sistema para esta etapa baseiam-se na prevenção de laços durante o processo de busca da solução, na relação de ordem entre os objetos e na prioridade e especificidade das regras.

5.1.1 APLICAÇÃO DOS CRITÉRIOS:

Para a aplicação dos critérios de resolução de conflitos, o SP1 efetua três operações: (a) restrição; (b) ordenação por prioridade; e (c) seleção da regra a disparar.

RESTRIÇÃO:

No SP1 a restrição consiste em retirar do CC certas instâncias, de forma a prevenir laços no grafo de estado, i.e., a execução da mesma seqüência de disparo repetidas vêzes.

Uma forma utilizada no SP1 limita-se a retirar do conjunto de conflitos qualquer instância anteriormente disparada. Esta restrição leva o nome de "refração", e é extensivamente utilizada nos SP's, como por exemplo no sistema OPS5 [FORGY 81].

No entanto nestes sistemas o acréscimo de um objeto à MT depois de sua deleção é equivalente à criação de uma nova instância, pois a marca de tempo deste objeto é modificada. Tal instância é no entanto sintaticamente idêntica a anterior. Este problema de identidade estrutural foi tratado no SP1, através da denominada "refração forte", que elimina do CC todas as instâncias sintaticamente equivalentes às disparadas em ciclos anteriores. Ela garante a terminação da procura de uma solução no grafo de estados, evitando laços infinitos devido a identidades estruturais não detetadas entre estados. Por exemplo, consideremos os objetos:

```
 $9  : (Robo ^nome r2d2 ^garra vazia  ^capacidade 4.0)
$10  : (Peca ^nome alfa ^acessivel ok ^peso 3.5 ^cor azul)
$11  : (Peca ^nome alfa ^acessivel ok ^peso 3.5)
$12  : (Peca ^nome alfa ^acessivel ok ^peso 3.5 ^cor azul)
```

Supondo que a instância (robo_pega_peca ($9 $10)) já tenha sido disparada, teríamos: (a) no caso da refração: as instâncias (robo_pega_peca ($9 $11)) e (robo_pega_peca ($9 $12)) poderiam

ser disparadas; (b) no caso da restrição forte: apenas a instância (robo_pega_peca ($9 $11)) poderia ser disparada, já que os objetos $10 e $12 são sintaticamente idênticos.

O SP1 oferece as opções de restrição por refração ou por refração forte. Nenhuma destas opções pode ser considerada como ideal: o mecanismo de restrição adequado é função do problema a resolver. No entanto a opção refração forte pode ser uma alternativa interessante para reduzir o tempo de resolução (ver exemplo 1 apêndice C).

ORDENAÇÃO POR PRIORIDADE:

O SP1 associa a cada regra um número inteiro que determina a prioridade da regra. A prioridade máxima é zero. As regras com prioridade negativa são consideradas desativadas.

A etapa de ordenação por prioridade corresponde a redução do CC apenas às instâncias formadas pelas regras de maior prioridade. Os demons tem prioridade zero, de forma que quando o sistema encontra um demon instanciado seu disparo ocorre imediatamente.

A operação de ordenação por prioridade é opcional no SP1.

SELEÇÃO DA REGRA POR AVALIAÇÃO:

Uma vez que o CC está gerado para a situação corrente, uma regra deve ser selecionada para disparo. O SP1 prove três estratégias para escolha desta regra: (a) FST, proporcionando a

escolha da primeira regra instanciada; (b) LEX, previligiando os objetos mais recentes e as regras mais específicas; e (c) MEA, uma modificação da estratégia anterior, que atribui importância especial à primeira variável presente na regra.

(a) FST:

Corresponde a escolha da primeira regra instanciada, de acordo com sua ordem de entrada na MR. O usuário pode determinar o controle através da ordem de entrada das regras.

(b) LEX:

É escolhida a instância que contém os fatos mais recentes, ou a regra mais específica [FORGY 81].

O CC é ordenado lexicograficamente de modo decrescente segundo as marcas de tempo dos objetos que compõem as instâncias, e previlegiando as instâncias com maior número de objetos. Por exemplo a instância (r1 ($3 $4 $8 $9)) é previlegiada em relação a (r2 ($3 $4 $8)), e esta em relação a (r2 ($3 $5 $7)). Se diversas regras forem previlegiadas de acordo com este critério, (i.e., se tiverem a mesma n-upla de objetos compondo a instância) a regra com maior número de condições será escolhida. Finalmente se este critério levar a um empate, uma escolha arbitrária é feita.

(c) MEA:

A estratégia MEA é uma variante da estratégia anterior que privilegia a primeira variável de cada regra [FORGY 81].

Seu objetivo é tornar o sistema sensível a problemas mais recentes. Se a primeira variável de cada regra for utilizada para representar metas ou subproblemas, a estratégia MEA criará uma pilha de sub-problemas provenientes desta regra, e fará com que o problema original só seja abandonado após sua resolução, pela obtenção da solução de todos os sub-problemas dele decorrentes. A utilização desta estratégia permite a construção de sistemas que simulam encadeamento misto.

A comparação entre instâncias na estratégia MEA é feita da seguinte forma: (a) Ordenação do CC de modo decrescente segundo a marca de tempo do primeiro objeto das n-uplas que compõem as instâncias; (b) Se várias instâncias forem previlegiadas (i.e., se tiverem o mesmo objeto como primeiro elemento de sua n-upla) aplica-se a estratégia LEX aos elementos restantes destas instâncias. Por exemplo a instância (r1 ($7 $2)) é previlegiada em relação a (r1 ($5 $9)).

5.1.2 LIMITAÇÃO DA PROFUNDIDADE DA BUSCA:

Uma interessante opção de controle é a indicação da profundidade máxima permissível para a inferência [NILSSON 80]. Esta limitação permite a redução do espaço de busca, transformando-o num conjunto finito. A utilização sucessiva de profundidades crescentes permite que se encontre a solução mais eficiente para o problema.

No SP1 a profundidade máxima de busca é definida pelo usuário. A ação tomada pelo MI do sistema quando este limite é ultrapassado depende do regime corrente do motor (ver a seguir).

5.1.3 REGIME IRREVOGÁVEL VERSUS REGIME POR TENTATIVAS:

É possível que na etapa de resolução de conflitos nenhuma regra esteja disponível para disparo. Por outro lado a profundidade corrente pode atingir o limite definido pelo usuário. A ação tomada nestes casos pelo MI depende de seu regime de funcionamento.

O SP1 pode funcionar tanto em regime irrevogável como por tentativas.

Em regime irrevogável, quando não há nenhuma instancia para disparo, o MI simplesmente pára.

Já em regime por tentativas ocorre retrocesso ao estado do sistema imediatamente anterior (backtrack sistemático). Uma nova etapa de resolução de conflitos é então realizada, proporcionando o disparo de outra instância. A instância já disparada é eliminada durante operação de restrição. Se não houver outra instância disponível, um novo retrocesso sistemático é realizado. Esta seqüência é repetida até o eventual esgotamento de todas as instâncias disponíveis, o que ocasionará a parada do MI.

Para realização de retrocesso , o SP1 armazena os objetos e as modificações ocorridas na MT a cada ciclo. No entanto, a restituição se limita a MT, ações como BUILD ou de controle não

são anuladas. Esta limitação pode ser importante no caso das ações de controle (em especial no manuseio de prioridades), e novas versões do SP1 deverão incorporar tais restituições.

O SP1 também permite operações de retrocesso dirigidas pelos dados, através da ação de controle "RETURN-OVER" (ver ítem 3.6.5).

5.2 EXECUÇÃO:

Na etapa de execução ocorre o disparo da regra selecionada na etapa anterior, de forma que as ações que constituem seu LDR são executadas seqüencialmente.

Como a seqüencialidade é obtida na versão atual durante a etapa de compilação das regras, o uso de funções LISP físicas não causarão o efeito desejado. O uso da ação BIND pode ser uma alternativa para esta situação, pois permite o armazenamento de valores temporários em tempo de execução (ver exemplo 9, apêndice C).

5.3 FILTRAGEM:

Esta etapa consiste na determinação, pelo interpretador do sistema, das instâncias de todas as regras e demons a partir do novo estado do sistema descrito pela MT resultante da etapa anterior. O algoritmo utilizado foi descrito em detalhes no capítulo anterior.

Uma mesma regra ou demon pode ter várias instâncias. Todas as instâncias válidas formam o conjunto de conflitos que é a saída desta etapa.

5.4 CONCLUSÃO:

O MI do sistema SP1 é um motor a três tempos, no qual o ciclo de base inicia pelo disparo de uma regra. Este disparo tem por efeito gerar de forma incremental o conjuntos de regras aplicáveis, segundo a metodologia de propagação descrita no capítulo anterior.

Sua principal característica é a diversidade de opções de controle. Estão previstas alternativas para uso de prioridades associadas às regras, deteção de laços no espaço de busca, estratégias de resolução de conflitos e escolha do regime de funcionamento do MI. Estas opções permitem ao usuário a escolha do mecanismo de controle mais adequado à resolução do problema em questão. No apêndice C são apresentados diversos exemplos de aplicação do SP1, na qual o controle adotado é salientado para cada caso.

VI. O AMBIENTE DE EXECUÇÃO

O sistema SP1 é acessível ao usuário por um ambiente de execução através do qual estão disponíveis as funções necessárias a execução, monitoração e obtenção dos resultados de um problema. Também estão acessíveis mecanismos para depuração e armazenamento de resultados.

A chamada ao sistema é feita acionando-se a função "sp1" (i.e., digitando-se " (sp1) ") a partir do interpretador LISP sobre o qual o SP1 foi desenvolvido. Surge então o prompt:

sp1>

A interpretação dos comandos no ambiente ocorre ciclicamente de acordo com a seqüência:

(a) leitura de um comando do usuário;

(b) execução do comando; e

(c) volta ao passo (a).

A seguir são descritos os comandos suportados. Os comandos têm a sintaxe básica:

(<comando> <argumento>*)

Analogamente a construção das regras, os comandos apresentam formato completamente livre. A tecla retorno_de_carro, quando utilizada no meio de um comando, é considerada como um espaço em branco. Para maior praticidade no manuseio os comandos são todos formados por caracteres minúsculos.

Para esta apresentação os comandos foram reunidos em sete grupos de acordo com sua função: sistema, execução, consulta, seleção, depuração, objeto e regra.

6.1 COMANDOS SISTEMA:

(a) quit:

Este comando permite encerrar uma seção de trabalho no SP1, retornando o controle ao interpretador LISP. Após a execução deste comando aparece o prompt do interpretador LISP:

?

(b) lisp:

O comando "lisp" permite avaliar qualquer s-expressão LISP, a partir do SP1. A s-expressão é avaliada pelo interpretador LISP e seu resultado é apresentado ao usuário. Por exemplo o comando (lisp (+ 3 4 5)) retorna:

sp1> 12

(c) system:

Este comando permite o uso temporário do sistema operacional. Após sua execução aparece o prompt do PC-DOS, e todos seus comandos ficam disponíveis ao usuário. Digitando-se "exit" o controle retorna ao SP1.

(d) help:

O comando "help" proporciona ao usuário auxilio a utilização do ambiente de execução. Se nenhum argumento é fornecido a lista

dos comandos válidos é apresentada. Caso seja fornecido um argumento o comando help fornecerá a sintaxe do mesmo, ou a mensagem "comando inexistente".

6.2 COMANDOS EXECUÇÃO:

(a) load:

O comando "load" permite carregar a BC relativa a um certo problema. Esta BC pode conter, além das regras de produção e demons que constituirão a MR, a definição do estado inicial do problema (MT). Eventualmente funções e predicados definidos pelo usuário podem também estar presentes. O comando carrega o compilador, e este gera as estruturas de apoio utilizadas pelo interpretador para a operação de fltragem.

(b) start:

O comando "start" prepara o ambiente SPl para a resolução de um novo problema, a partir do conhecimento operatório disponível no ambiente.

(c) run:

O comando "run" permite executar n ciclos de inferência. Se nenhum argumento é fornecido, a inferência prossegue até que uma ação STOP ou HALT seja executada, um ponto de quebra (breakpoint) seja atingido, ou até que não haja mais possibilidade de novas inferências.

Quando um argumento está presente, deve ser um número inteiro, que determina o número de ciclos a serem executados pelo MI. Por exemplo, o comando (run 50) provocará a execução de 50 ciclos pelo MI, a menos que algum critério de parada seja verificado. No caso de um número negativo o sistema restabelecerá o estado do sistema correspondente a profundidade calculada por:

profundidade atual + argumento

Caso este cálculo resulte menor que zero a mensagem "retorno impossível" é apresentada.

(d) manual:

Em modo "manual" o usuário pode escolher interativamente a cada ciclo a instância de regra a disparar. Este comando é especialmente útil nas etapas de modelagem e depuração.

(e) backtrack:

O comando "backtrack" realiza uma operação de retrocesso (backtrack) sob comando do usuário. Um número natural deve ser fornecido como argumento e representa a profundidade relativa que deve ser anulada. A profundidade do ponto de backtrack é portanto encontrada pela diferença:

profundidade corrente - deslocamento

Caso este cálculo resulte negativo, a operação se torna impossível a mensagem "backtrack impossível" aparece. O "backtrack" difere de um comando "run" com argumento negativo pois neste caso os ramos anteriormente percorridos não mais serão explorados.

6.3 COMANDOS CONSULTA:

(a) wm:

O comando "wm" apresenta os objetos que compõem a MT, com a sua marca de tempo.

(b) rm:

O comando "rm" apresenta as regras e demons que constituem a MR.

(c) cs:

Apresenta o conjunto de conflitos para o estado corrente do sistema.

(d) facts:

O comando "facts" permite a visualização da estrutura de objetos pertencentes a MT ou já utilizados pelo sistema. Para tanto basta fornecer como argumento uma lista de marcas de tempo correspondentes a estes objetos.

(e) list:

Permite listar uma ou mais regras, cujos nomes são fornecidos como argumentos.

(f) depth:

Apresenta a profundidade corrente da inferência. Como a profundidade pode ser usada para fins de controle, é importante que o usuário tenha acesso a esta informação.

(g) path:

O comando "path" apresenta ao usuário as instâncias disparadas deste o início da resolução.

6.4 COMANDOS SELEÇÃO:

(a) restriction:

Permite selecionar qual o mecanismo utilizado na operação de restrição durante a resolução de conflitos. Se nenhum argumento é fornecido, o comando retorna a opção atual. Os argumentos válidos são: ref, para seleção de refração (default) e sre, para seleção de refração forte.

(b) prior:

Determina que a operação de ordenação por prioridade deve ser executada na etapa de resolução de conflitos do sistema. É a opção por default.

(c) unprior:

Cancela a operação de ordenação por prioridade. A operação é então realizada como se todas as regras tivessem a mesma prioridade. Os demons não estão sujeitos à ação deste comando.

(d) strategy:

Permite a seleção da estratégia de resolução de conflitos. Se nenhum argumento é fornecido, a estratégia corrente é mostrada ao usuário. Os argumentos válidos são: (a) fst, para escolha da primeira regra; (b) lex, para ordem lexicográfica e (c) mea, para análise meio-fim. Por default o SP1 usa estratégia lex.

(e) regime:

Define o regime de funcionamento do MI (por tentativas ou irrevogável). Se nenhum argumento é fornecido o regime corrente é apresentado. Os argumentos válidos são: REV, para o regime por tentativas e IRR para o regime irrevogável. Por default o sistema atua em regime por tentativas.

(f) max-depth:

O comando "max-depth" estabelece a profundidade máxima permitida para a busca. Se nenhum argumento é fornecido, a profundidade máxima corrente é apresentada. Caso haja um argumento, este deve ser um número natural que estabelecerá a nova profundidade máxima.

6.5 COMANDOS DEPURAÇÃO:

(a) trace:

O comando "trace" determina o aparecimento de informações sobre cada disparo. São apresentados, a cada ciclo, o ciclo e

profundidade atuais, o conjunto de conflitos, a instância escolhida, e os objetos acrescentados, removidos e modificados.

(b) untrace:

Permite cancelar a opção anterior.

(c) step:

Proporciona a execução do MI passo a passo, de forma a permitir a monitoração do processo de inferência. A cada ciclo são apresentados: o conjunto de conflitos, a instância disparada, os objetos acrescentados, removidos e alterados, a MT e a MR.

(d) log:

O comando "log" redireciona a saída do sistema para um arquivo cujo nome é fornecido como argumento e de extensão "LOG"; caso nenhum argumento seja fornecido, o nome SP1.LOG é assumido por default. Todas as informações de saída são armazenadas no arquivo, permitindo posterior análise do problema.

(e) unlog:

O comando unlog determina o redirecionamento da saída para a tela, e fecha o arquivo de log (ver item anterior).

(f) pbreak:

O comando "pbreak" acrescenta um ponto de quebra (breakpoint) na regra cujo nome é fornecido como argumento. O MI pára quando do disparo desta regra.

(g) del-pbreak:

Retira o ponto de quebra da regra cujo nome é fornecido como argumento.

6.6 COMANDOS OBJETO:

(a) create:

O comando "create" permite criar um objeto na MT a partir do ambiente de execução. Por exemplo o comando:

(create Peca ^tipo alfa ^peso 40)

acrescenta a MT um novo objeto da classe "Peca" com os atributos "tipo" e "peso" de valores "alfa" e "40".

(b) modify:

Permite a alteração ou inclusão de atributos num objeto presente na MT, a partir do ambiente de execução. O comando:

(modify $3 ^cor azul)

modifica o atributo "cor" do objeto $3 para "azul".

(c) delete:

O comando "delete" possibilita a retirada de um objeto da MT. Por exemplo a execução do comando:

(delete $2)

causa a retirada do objeto $2 de MT.

6.7 COMANDOS REGRA:

(a) priority:

O comando "priority" permite atribuir uma nova prioridade a uma regra. O primeiro argumento é o nome da regra e o segundo um número inteiro correspondente a nova prioridade. Se o número for negativo, a regra é desativada.

(b) excise:

O comando "excise" permite desativar uma regra. O argumento deve ser o nome da regra a desativar. Este comando é equivalente a atribuição de uma prioridade -1 à regra.

(c) revive:

Permite a reativação de uma regra. O argumento fornecido é o nome da regra, cuja prioridade passa a ter o valor default (1). O comando revive equivalente a atribuição de prioridade 1.

(d) build:

O comando build equivale no ambiente de execução à ação BUILD da linguagem de representação do conhecimento. Permite a criação de uma nova regra, sua compilação e incorporação as estruturas de apoio do interpretador.

6.8 CONCLUSÃO:

O ambiente de execução do sistema fornece ao usuário facilidades para execução, monitoração e depuração de um programa SP1. Também as diversas opções de funcionamento do MI estão acessíveis, permitindo que o usuário configure de forma apropriada o sistema.

Esta interação se dá num sub-ambiente criado sobre o interpretador LISP utilizado para construção do SP1, através de mecanismos simples de edição de linhas.

VII. CONCLUSÃO E PERSPECTIVAS

Nesta dissertação é apresentado o SPl, um sistema de regras de produção cuja especificação e implementação de um protótipo foram objeto deste trabalho.

Durante o desenvolvimento deste projeto, as seguintes etapas foram cumpridas:

(a) especificação da linguagem de representação do conhecimento utilizada pelo sistema;

(b) desenvolvimento de um algoritmo para a operação de filtragem;

(c) desenvolvimento do motor de inferência do sistema e mecanismos de controle;

(d) implementação de uma versão protótipo do sistema em LISP; e

(e) especificação e implementação de um ambiente de execução para o sistema.

A seguir lembram-se as principais características do sistema:

(a) Linguagem:

A linguagem de representação do conhecimento do SPl utiliza uma representação multi-atributo para os objetos e uma notação predicativa para as regras, com ênfase à legibilidade e facilidade de redação das mesmas.

A expressividade da linguagem é aumentada pela possibilidade de utilização de atributos compostos. Além disto é possível a criação dinâmica de novas regras, o que dota o sistema capacidade de aprendizagem.

A linguagem possui grande extensibilidade, permitindo que funções e predicados sejam definidos pelo usuário.

(b) Algoritmo de filtragem:

Como a etapa de filtragem é crítica em termos de performance para um SP, um novo algoritmo foi desenvolvido para o SPl. Este algoritmo se baseia na compilação dos antecedentes das regras, e permite a eliminação de diversas redundâncias.

O CC é gerado a cada ciclo de forma incremental, por propagação das alterações produzidas sobre a MT nas estruturas (grafo de unificação e grafo de junções) geradas pelo compilador. O algoritmo é particularmente eficaz no caso de modificações parciais sobre os objetos.

(c) Motor de inferência:

O SPl atua de forma não-monotônica, em encadeamento para frente, com sensibilização das regras à esquerda. Uma instância de regra é escolhida a cada ciclo para disparo.

Várias opções de controle estão disponíveis. A repetição de seqüências de disparos é evitada pela deteção de laços no espaço de busca, segundo critérios de refração ou refração forte. A resolução de conflitos é feita por avaliação, com base na prioridade e especificidade das regras e na anterioridade dos

fatos que compõem a instância. São providas as estratégias FST (escolha da primeira regra), LEX (ordem lexicográfica) e MEA (análise meio-fim).

O SP1 pode funcionar tanto em regime irrevogável quanto por tentativas. Em regime por tentativas o sistema realiza retrocesso sistemático. Uma ação especial permite a execução de um retrocesso dirigido pelos dados.

(d) Ambiente de execução:

O ambiente de execução do SP1 está centrado na idéia básica que a interação é indispensável a um SP. Desta forma o SP1 proporciona ao usuário, através de mecanismos de edição em linhas, as funções necessárias para execução, monitoração e depuração de um programa SP1.

A versão protótipo do SP1 foi implementada num microcomputador IBM-PC compatível, sobre um interpretador LISP, e se encontra atualmente em fase de testes e validação. Esta versão é constituída por cerca e 2.200 linhas-fonte LISP, e utiliza em torno de 170 funções.

O desempenho do sistema (e em especial, do algoritmo de filtragem) foi avaliado utilizando exemplos clássicos de IA. Utilizando-se um microcomputador PC-AT o sistema realiza de 0,6 a 11,1 inferências por segundo (média de 3,84). O tempo gasto na etapa de filtragem corresponde de 35% a 96% do total (média 74%). Cerca de 2% a 35% (média 13%) do tempo total da etapa é consumido

para instanciação das condições, de 3% a 43% (média 22%) para instanciação das variáveis e de 36% a 96% (média 64%) para instanciação das regras. Os exemplos com pior performance são justamente aqueles que apresentam maior porcentagem de tempo consumido na instanciação das regras. Isto permite deduzir que as junções constituem a etapa crítica do algoritmo.

Com base nos resultados obtidos a partir da versão piloto do sistema, uma série de estudos e pesquisas podem ser desenvolvidos:

- Desenvolvimento de um editor "inteligente" para a linguagem SPl, com verificação automática de sintaxe, tipagem e consistência das informações à medida da edição.

- Extensão da linguagem, pela incorporação de novos tipos de dados (e.g. intervalos, que são importantes para uso em aplicações de Controle e Automação).

- Especificação e implementação de mecanismos de explanação para o sistema SPl.

- Estudo de aprendizagem automática (learning) com base na capacidade provida pelo SPl.

- Estudo para incorporação no SPl de mecanismos de raciocínio inexato.

- Aperfeiçoamento do algoritmo de filtragem, pelo estabelecimento de um maior número de relacionamentos entre as condições. Certas conclusões serão também compiladas.

- Estudo da complexidade do algoritmo de filtragem proposto.

- Implementação de uma nova versão do algoritmo, com acesso direto aos atributos e onde as estruturas de apoio ao interpretador serão transformadas em funções diretamente avaliáveis pelo interpretador LISP. Também nesta versão as modificações sobre objetos serão propagadas individualmente.

- Desenvolvimento de uma versão do sistema em linguagem "C", o que permitirá uma real avaliação da performance e aplicabilidade do sistema.

APÊNDICE A

DESCRIÇÃO SINTÁTICA DA LINGUAGEM SP1

A seguir é apresentada uma descrição sintática da linguagem SP1 numa forma BNF [DONOVAN 84]. Nesta BNF os símbolos não-terminais serão prefixados por "<" e sufixados por ">". Também são utilizados os meta-símbolos "::=" (é definido por), "*" (repetição zero ou mais vezes da expressão quantificada), "+" (repetição uma ou mais vezes da expressão quantificada), "|" (ou), "{" e "}" (que cercam elementos opcionais).

```
<marca_de_tempo>    ::= $ <número_natural>

<átomo>             ::= <número> | <símbolo>

<símbolo>           ::= <símbolo_M> | <símbolo_m>

<conjunto>          ::= { <constante>* }

<constante>         ::= <átomo> | <conjunto> | <vetor>

<vetor>             ::= #[ <átomo>+ ]

<classe>            ::= <símbolo_M>

<atributo>          ::= <símbolo_m>

<regra>             ::= (RULE <nome> {<prioridade>}
                            IF    <antecedente>
                            THEN  <consequente>)

<demon>             ::= (DEMON <nome>
                            IF    <antecedente>
                            THEN <consequente>)

<nome>              ::= <símbolo_m>

<prioridade>        ::= <número_inteiro>

<variável>          ::= <símbolo_M>
```

```
<antecedente>       ::= <d_classe>+ <condição>*

<d_classe>          ::= | - | <classe> ( <variável> )

<condição>          ::= <d_atributo> <predicado> <termo>+ |
                        <d_atributo> <operador_lóg>
                                     <seq_de_predicados>
<d_atributo>        ::= <atributo>(<variável>) |
                        <atributo>(<d_atributo>)

<predicado>         ::= = | <> | > | < | >= | <= | Member |
                          <operador_lóg> |
                          <símbolo_M>
<operador_lóg>      ::= Not | And | Or

<termo>             ::= <constante>  |
                        <d_atributo> |
                        <variável>   |
                        <função_esc> |
                        <função_vet>
<função_esc>        ::= (<símbolo_m> <termo>*) | <constante>

<função_vet>        ::= | <função_esc>+ |

<seq_predicados>    ::= (<predicado> <termo>+)+ |
                        (<operador_lóg> <seq_predicados>)+

<conseqüente>       ::= <ação_sobre_MT>*
                        <ação_de_controle>*
                        <ação_de_e/s>*
                        <ação_auxiliar>*
<ação_sobre_MT>     ::= <modificação> |
                        <deleção>   |
                        <acréscimo>
<modificação>       ::= <d_atributo> = <termo>

<deleção>           ::= - <variável>

<acréscimo>         ::= <d_classe> <modificação>* |
                        + <variável> = <d_atributo>

<ação_de_controle>  ::= (STOP) |
                        (HALT) |
                        (PRIORITY <nome> <prioridade>) |
                        (EXCISE <nome>) |
                        (REVIVE <nome>) |
                        (HARAKIRI) |
                        (RETURN-OVER <variável>) |

<ação_de_e/s>       ::= <d_atributo> = (accept <string>) |
                        (WRITE <string> <termo>*)

<ação_auxiliar>     ::= (BIND <variável> <termo>) |
                        (BUILD <regra'>)
```

Os elementos <número> e <número_natural> (número inteiro sem sinal) são definidos como em LISP. Utilizamos <símbolo_M> para designar símbolos LISP que iniciam por letra maiúscula e <símbolo_m> para os demais. <string> representa uma cadeia de caracteres como em LISP. <regra'> deve ter a sintaxe <regra> após as substituições decorrentes da instanciação da ação BUILD.

Além das regras e dos demons que compõem a BC para um problema, podem estar presentes comandos CREATE e SET, cuja sintaxe é indicada abaixo.

```
<set>       ::= (SET @<símbolo> <valor>)
<create>    ::= (CREATE <classe>{^<atributo><valor>}*)
<valor>     ::= <número> |
                <símbolo_m> |
                <conjunto> |
                <vetor> |
                <marca_de_tempo>
```

APÊNDICE B

COMANDOS DO AMBIENTE DE EXECUÇÃO

A seguir são apresentados os diversos comandos do ambiente de execução do sistema SP1.

```
<comando>     ::= <c_sistema> | <c_execução> | <c_consulta> |
                  <c_seleção> | <c_depuração> | <c_objeto> |
                  <c_regra>
<c_sistema>   ::= (quit) |
                  (lisp <s-expr>) |
                  (system) |
                  (help <nome_comando>*)
<c_execução>  ::= (load <arquivo>) |
                  (start) |
                  (run <numero_inteiro>*) |
                  (manual)
                  (backtrack <numero_natural>)
<c_consulta>  ::= (wm) |
                  (rm) |
                  (cs) |
                  (facts <marca_de_tempo>+) |
                  (list <nome>+) |
                  (depth) |
                  (path)
<c_seleção>   ::= (restriction {ref | sre}) |
                  (prior) |
                  (unprior) |
                  (strategy {fst | lex | mea}) |
                  (regime {rev | irr}) |
                  (max-depth { <numero_natural> })
<c_depuração>::= (trace) |
                  (untrace) |
                  (step) |
                  (log { <arquivo> }) |
                  (unlog) |
                  (pbreak <nome>) |
                  (del-pbreak <nome>)
<c_objeto>    ::= (create <classe> {^<atributo> <valor>}*) |
                  (modify <marca_de_tempo>{^<atributo><valor>}+) |
                  (delete <marca_de_tempo>)
<c_regra>     ::= (priority <nome> <prioridade>) |
                  (excise <nome>) |
                  (revive <nome>) |
                  (build <regra'>)
```

O elemento <s-expr> designa qualquer s-expressão LISP. <nome_comando> deve ser o símbolo que nomeia um comando do ambiente de execução. <arquivo> é um símbolo que identifica um nome de arquivo válido para o sistema operacional.

APÊNDICE C

EXEMPLOS DE UTILIZAÇÃO DO SISTEMA

A seguir são apresentado diversos exemplos de aplicação do SP1, procurando ilustrar suas opções de funcionamento. Alguns destes problemas são adequados a solução algorítmicas. No entanto são apresentados com intuito de salientar características do SP1. Será dada ênfase a estratégia de controle utilizada para a resolução.

Cada problema pode ser definido por tripla

$$P = < I, O, C >$$

onde:

I : conjunto de expressões que representam o estado ou a condição inicial do problema;

O : conjunto de operadores ou transformações que se podem efetuar sobre a situação inicial ou sobre expressões que dela derivam mediante alguma seqüência de operações anteriores; e

C : condição de parada, i.e., a condição que deve ser satisfeita pela expressão terminal do sistema.

A resolução de um problema consiste na determinação de operadores O1, O2, ..., On tal que sua aplicação ao estado inicial satisfaz a condição de parada. Dependendo do problema a

solução desejada pode corresponder a seqüência O1, O2, ..., On, ou a situação descrita pela MT final.

Para cada problema a seguir serão apresentados:

(a) Uma descrição do problema e das técnicas empregadas para sua resolução;

(b) o conjunto de operadores (regras e demons) e o estado inicial que o caracterizam; e

(c) um exemplo de uma sessão de trabalho do sistema, com a sequência de comandos e as informações obtidas.

1. PROBLEMA DO QUEBRA CABECA DE OITO (Jeu de taquin):

Este problema clássico de IA [NILSSON 71] ilustra a importância do controle por limitação de profundidade de busca e da restrição por "refração forte". Nas várias execuções com diferentes parâmetros esta diferença se torna evidente. O tabuleiro é representado por um vetor de nove coordenadas. A movimentação das peças é feita através de funções definidas pelo usuário.

Base de conhecimentos para o problema

```
; regra para criar estado inicial do problema
(RULE inicio
       IF   Comeco(Ok)
       THEN
          - Ok
            Taquin(X)
            tab(X) = (@ vector 2 8 3 1 6 4 7 '* 5)
            *(X) = 8)
; regras que indicam os movimentos possíveis
(RULE cima
       IF   Taquin(X)
            *(X) Not (Member {1 2 3})
       THEN
            *(X) = (- *(X) 3)
            tab(X) = (up tab(X)))
```

```
(RULE baixo
        IF    Taquin(X)
              *(X) Not (Member {7 8 9})
        THEN
              *(X) = (+ *(X) 3)
              tab(X) = (down tab(X)))
(RULE esquerda
        IF    Taquin(X)
              *(X) Not (Member {1 4 7})
        THEN
              *(X) = (- *(X) 1)
              tab(X) = (left tab(X)))
(RULE direita
        IF    Taquin(X)
              *(X) Not (Member {3 6 9})
        THEN
              *(X) = (+ *(X) 1)
              tab(X) = (right tab(X)))
; condição de parada
(DEMON fim
        IF    Taquin(X)
              tab(X) = (@ vector 1 2 3 8 '* 4 7 6 5)
        THEN
              (STOP))
(CREATE Comeco)
```

Funções definidas pelo usuário

```
; realizam modificação no vetor que representa o tabuleiro
(de find (a v)
    (let ((n 0))
         (until (equal (vref v n) a)
                (incr n))
         n))
(de up (v)
    (let ((p* (find '* v))
          (v2 (copyvector v)))
         (vset v2 p* (vref v2 (- p* 3)))
         (vset v2 (- p* 3) '*)
         v2))
(de down (v)
    (let ((p* (find '* v))
          (v2 (copyvector v)))
         (vset v2 p* (vref v2 (+ p* 3)))
         (vset v2 (+ p* 3) '*)
         v2))
(de right (v)
    (let ((p* (find '* v))
          (v2 (copyvector v)))
         (vset v2 p* (vref v2 (1+ p*)))
         (vset v2 (1+ p*) '*)
         v2))
```

```
(de left (v)
    (let ((p* (find '* v))
          (v2 (copyvector v)))
         (vset v2 p* (vref v2 (1- p*)))
         (vset v2 (1- p*) '*)
         v2))
(defmacro copyvector (vect)
    '(let ((n (vlength ,vect))
           (vect2 (makevector (vlength ,vect) ())))
         (until (= (decr n) -1)
                (vset vect2 n (vref ,vect n)))
         vect2))
```

Exemplo de uma sessão de trabalho

```
spl> (load taquin3)
Pre-Compilador      : .\spllib\precomvl.3
Compilador          : .\spllib\compilvl.3
Pre e Compilacao    : .\splbc\taquin3
spl> (run 1)
1
spl> (wm)
  $2 : (Taquin ^tab #[2 8 3 1 6 4 7 * 5] ^* 8)
spl> (cs)
((cima ($2)) (esquerda ($2)) (direita ($2)))
spl> (restriction)
ref
```

```
spl> (max-depth 7)
7
spl> (run)
117
   *** Solucao ***

- ciclos ..... 117
- backtrackings ... 55

   * 1 ..... ((inicio ($1)))
   * 2 ..... ((cima ($2)))
   * 3 ..... ((cima ($3)))
   * 4 ..... ((esquerda ($4)) (baixo ($4)))
   * 5 ..... ((baixo ($5)))
   * 6 ..... ((direita ($6)) (baixo ($6)) (cima ($6)))
   * 7 ..... ((fim ($7)))
spl> (start)
t
spl> (create Comeco)
()
spl> (restriction sre)
sre
** de : fonction redefinie : #:spl:verifica-inst
spl> (run)
67
```

```
   *** Solucao ***

- ciclos ..... 67
- backtrackings ... 30

   * 1 ..... ((inicio ($1)))
   * 2 ..... ((cima ($2)))
   * 3 ..... ((cima ($3)))
   * 4 ..... ((esquerda ($4)) (baixo ($4)))
   * 5 ..... ((baixo ($5)))
   * 6 ..... ((direita ($6)) (baixo ($6)) (cima ($6)))
   * 7 ..... ((fim ($7)))
spl> (wm)
  $7 : (Taquin ^tab #[1 2 3 8 * 4 7 6 5] ^* 5)
spl> (quit)
?              Retorno ao interpretador LISP
```

2. TORRES DE HANOI:

O problema clássico da torres de Hanói é aqui resolvido por decomposição de problemas. O uso da estratégia MEA faz com que cada sub-problema seja totalmente resolvido antes que o sistema passe a resolver outro sub-problema. Seu uso neste exemplo é obrigatório, pois determina a ordem de movimentação dos discos, uma restrição imposta pelo problema.

Base de conhecimentos para o problema

```
; regra para criar o estado inicial do problema
(RULE inicio
      IF
          Comeco(Go)
      THEN
        - Go
          Torre(Tor1)
          Torre(Tor2)
          Torre(Tor3)
          Meta(Meta)
          discos(Tor1) = 3
          nome(Tor1) = a
          discos(Tor2) = 0
          nome(Tor2) = b
          discos(Tor3) = 0
          nome(Tor3) = c
```

```
            discos(Meta) = 3
            de(Meta) = a
            para(Meta) = b
            outra(Meta) = c)
; determina a decomposição do problema em metas
(RULE cria_metas
      IF
            Meta(Meta_entrada)
            discos(Meta_entrada) > 1
      THEN
            Meta(M1)
            Meta(M2)
            Meta(M3)
            de(M1) = outra(Meta_entrada)
            para(M1) = para(Meta_entrada)
            outra(M1) = de(Meta_entrada)
            discos(M1) = (- discos(Meta_entrada) 1)
            de(M2) = de(Meta_entrada)
            para(M2) = para(Meta_entrada)
            outra(M2) = outra(Meta_entrada)
            discos(M2) = 1
            de(M3) = de(Meta_entrada)
            para(M3) = outra(Meta_entrada)
            outra(M3) = para(Meta_entrada)
            discos(M3) = (- discos(Meta_entrada) 1)
          - Meta_entrada)
```

```
; movimentação de um disco entre as torres
(RULE move_um_disco
      IF
          Meta(Meta_e)
          Torre(Torre_para)
          Torre(Torre_de)
          discos(Meta_e)    = 1
          nome(Torre_para)  = para(Meta_e)
          nome(Torre_de)    = de(Meta_e)
          discos(Torre_de)  > 0
      THEN
        - Meta_e
          discos(Torre_para) = (+ discos(Torre_para) 1)
          discos(Torre_de)   = (- discos(Torre_de)   1))
; condição de parada
(DEMON fim
       IF
           Torre(X)
           Torre(Y)
           Torre(Z)
           nome(X) = a
           discos(X) = 0
           nome(Y) = b
           discos(Y) = 3
           nome(Z) = c
           discos(Z) = 0
       THEN
```

```
            (STOP))
; inicialização da MT
(CREATE Comeco)
```

Exemplo de uma sessão de trabalho

```
spl> (load hanoi)
Pre-Compilador       : .\spllib\precomvl.3
Compilador           : .\spllib\compilvl.3
Pre e Compilacao     : .\splbc\hanoi.ll
spl> (strategy mea)
mea
**de : fonction redefinie : #:spl:escolhe-regra
spl> (regime irr)
irr
spl> (trace)
t
spl> (run)
   * 1 :
       - profund.   = 1
       - cs antes   = ((inicio ($1)))
       - instancia = (inicio ($1))
       - fatos (+) = ($5 $4 $3 $2)
       - fatos (-) = ($1)
   * 2 :
       - profund.   = 2
       - cs antes   = ((cria_metas ($5)))
```

- instancia = (cria_metas ($5))
- fatos (+) = ($8 $7 $6)
- fatos (-) = ($5)

* 3 :

- profund. = 3
- cs antes = ((cria_metas ($8) ($6)) (move_um_disco ($7 $3 $2)))
- instancia = (cria_metas ($8))
- fatos (+) = ($11 $10 $9)
- fatos (-) = ($8)

* 4 :

- profund. = 4
- cs antes = ((cria_metas ($6)) (move_um_disco ($11 $3 $2) ($10 $4 $2) ($7 $3 $2)))
- instancia = (move_um_disco ($11 $3 $2))
- fatos (+) = ($13 $12)
- fatos (-) = ($2 $3 $11)

* 5 :

- profund. = 5
- cs antes = ((cria_metas ($6)) (move_um_disco ($7 $12 $13) ($10 $4 $13) ($9 $4 $12)))
- instancia = (move_um_disco ($10 $4 $13))
- fatos (+) = ($15 $14)
- fatos (-) = ($13 $4 $10)

* 6 :

- profund. = 6

- cs antes = ((cria_metas ($6)) (move_um_disco ($7 $12 $15) ($9 $14 $12)))
- instancia = (move_um_disco ($9 $14 $12))
- fatos (+) = ($17 $16)
- fatos (-) = ($12 $14 $9)

* 7 :

- profund. = 7
- cs antes = ((cria_metas ($6)) (move_um_disco ($7 $17 $15)))
- instancia = (move_um_disco ($7 $17 $15))
- fatos (+) = ($19 $18)
- fatos (-) = ($15 $17 $7)

* 8 :

- profund. = 8
- cs antes = ((cria_metas ($6)))
- instancia = (cria_metas ($6))
- fatos (+) = ($22 $21 $20)
- fatos (-) = ($6)

* 9 :

- profund. = 9
- cs antes = ((move_um_disco ($22 $19 $16) ($21 $18 $16)))
- instancia = (move_um_disco ($22 $19 $16))
- fatos (+) = ($24 $23)
- fatos (-) = ($16 $19 $22)

* 10 :

- profund. = 10

```
        - cs antes  = ((move_um_disco ($21 $18 $24) ($20 $18
$23)))
        - instancia = (move_um_disco ($21 $18 $24))
        - fatos (+) = ($26 $25)
        - fatos (-) = ($24 $18 $21)
    * 11 :
        - profund.  = 11
        - cs antes  = ((move_um_disco ($20 $25 $23)))
        - instancia = (move_um_disco ($20 $25 $23))
        - fatos (+) = ($28 $27)
        - fatos (-) = ($23 $25 $20)
    * 12 :
        - profund.  = 12
        - cs antes  = ((fim ($28 $27 $26)))
        - instancia = (fim ($28 $27 $26))
        - fatos (+) = ()
        - fatos (-) = ()

    *** Situacao Final ***

- ciclos ..... 12
- backtrackings ... 0

    * 1 ..... ((inicio ($1)))
    * 2 ..... ((cria_metas ($5)))
    * 3 ..... ((cria_metas ($8)))
    * 4 ..... ((move_um_disco ($11 $3 $2)))
```

```
   * 5 ..... ((move_um_disco ($10 $4 $13)))
   * 6 ..... ((move_um_disco ($9 $14 $12)))
   * 7 ..... ((move_um_disco ($7 $17 $15)))
   * 8 ..... ((cria_metas ($6)))
   * 9 ..... ((move_um_disco ($22 $19 $16)))
   * 10 ..... ((move_um_disco ($21 $18 $24)))
   * 11 ..... ((move_um_disco ($20 $25 $23)))
   * 12 ..... ((fim ($28 $27 $26)))

spl> (wm)
 $28 : (Torre ^discos 0 ^nome a)
 $27 : (Torre ^discos 3 ^nome b)
 $26 : (Torre ^discos 0 ^nome c)
spl>
```

3. MANUSEIO DE CUBOS:

O problema clássico do manuseio de três cubos ([FARRENY 87], [NILSSON 80], [RICH 83]) é aqui resolvido não através de decomposição de problemas (como é usual e adequado) mas por busca exaustiva. O espaço de busca é limitado a uma profundidade 9. Esta abordagem é ineficiente, mas está aqui colocada para ilustrar o mecanismo de retrocesso (backtrack). Este exemplo também ilustra a criação de uma nova regra pela ação BUILD, e utiliza a possibilidade do sistema de armazenar um objeto como atributo de outro objeto.

Base de conhecimentos para o problema

```
; exemplo de criação dinâmica de uma regra
(DEMON inicio
    IF
       Comeco(X)
       Cubo(ZZ)
    THEN
     - X
     - ZZ
       Comeco2(W)
       Garra(Y)
       Cubo(Z)
       Cubo(T)
       segura(Y)       = nada
```

```
        nome(Z)         = a
        acessivel(Z)    = sim
        esta_sobre(Z)   = bancada
        nome(T)         = b
        acessivel(T)    = sim
        esta_sobre(T)   = bancada
        (build
                (DEMON inicio2
                        IF    Comeco2(W)
                              Cubo(T)
                              nome(T)        = nome(ZZ)
                              acessivel(T)   = sim
                        THEN
                           - W
                              Cubo(U)
                              nome(U)         = c
                              acessivel(U)    = sim
                              esta_sobre(U)   = T
                              acessivel(T)    = nao
                              (HARAKIRI)))
        (HARAKIRI))
; movimentos válidos
(RULE pega_do_plano
    IF
        Garra(X)
        Cubo(Y)
        segura(X)       = nada
```

```
        acessivel(Y)   = sim
        esta_sobre(Y) = bancada
    THEN
        segura(X)        = Y
        esta_sobre(Y)    = nada
        acessivel(Y)     = nao)
(RULE pega_de_outro_objeto
    IF
        Garra(X)
        Cubo(Z)
        Cubo(Y)
        segura(X)        = nada
        acessivel(Y)     = sim
        esta_sobre(Y)    = Z
        esta_sobre(Y)    <> bancada
    THEN
        segura(X)        = Y
        esta_sobre(Y)    = nada
        acessivel(Y)     = nao
        acessivel(Z)     = sim)
(RULE coloca_no_plano
   IF
        Cubo(Y)
        Garra(X)
        segura(X)        = Y
   THEN
        segura(X)        = nada
```

```
        esta_sobre(Y)    = bancada
        acessivel(Y)     = sim)
(RULE empilha
   IF
        Cubo(Y)
        Garra(X)
        Cubo(Z)
        segura(X)        = Y
        acessivel(Z)     = sim
   THEN
        segura(X)        = nada
        esta_sobre(Y)    = Z
        acessivel(Y)     = sim
        acessivel(Z)     = nao)
; condição de parada
(DEMON fim
   IF
        Cubo(Z)
        Cubo(Y)
        Cubo(X)
        Garra(T)
        nome(X) = a
        esta_sobre(X) = Y
        nome(Y) = b
        esta_sobre(Y) = Z
        nome(Z) = c
        esta_sobre(Z) = bancada
```

```
        segura(T) = nada
    THEN
        (WRITE "Cheguei ao fim")
        (STOP))
; inicialização da MT
(CREATE Comeco)
(CREATE Cubo ^nome a)
```

Exemplo de uma sessão de trabalho

```
spl> (load cubos)
Pre-compilador      : .\spllib\precomvl.3
Compilador          : .\spllib\complivl.3
Pre e compilacao    : .\splbc\cubos.ll
spl> (wm)
  $2 : (Cubo ^nome a)
  $1 : (Comeco)
spl> (rm)
((D : inicio fim) (R : pega_do_plano pega_de_outro_objeto
coloca_no_plano
empilha))
spl> (run 1)
Pre-compilador      : .\spllib\precomvl.3
Compilador          : .\spllib\compilvl.3
Pre e Compilacao    : inicio2
1
```

```
spl> (wm)
  $3 : (Comeco2)
  $4 : (Garra ^segura nada)
  $5 : (Cubo ^nome a ^acessivel sim ^esta_sobre bancada)
  $6 : (Cubo ^nome b ^acessivel sim ^esta_sobre bancada)
spl> (rm)
((D :  fim inicio2) (R :  pega_do_plano pega_de_outro_objeto
coloca_no_plano empilha))
spl> (run 1)
2
spl> (wm)
  $8 : (Cubo ^nome a ^acessivel nao ^esta_sobre bancada)
  $7 : (Cubo ^nome c ^acessivel sim ^esta_sobre $8)
  $4 : (Garra ^segura nada)
  $6 : (Cubo ^nome b ^acessivel sim ^esta_sobre bancada)
spl> (rm)
((D :  fim) (R :  pega_do_plano pega_de_outro_objeto
coloca_no_plano empilha))
spl> (max-depth 9)
9
spl> (strategy)
lex
spl> (restriction)
ref
spl> (run)
706
```

```
Cheguei ao fim
707

   *** Solucao ***

- ciclos ..... 707
- backtrackings ... 349

   * 1 ..... ((inicio ($1 $2)))
   * 2 ..... ((inicio2 ($3 $5)))
   * 3 ..... ((pega_de_outro_objeto ($4 $8 $7)))
   * 4 ..... ((coloca_no_plano ($10 $9)) (empilha ($10 $9 $6))
(empilha ($10 $9 $11)))
   * 5 ..... ((pega_do_plano ($12 $6)) (pega_do_plano ($12 $11))
(pega_do_plano ($12 $13)))
   * 6 ..... ((empilha ($15 $14 $13)))
   * 7 ..... ((pega_do_plano ($16 $11)) (pega_de_outro_objeto
($16 $18 $17)))
   * 8 ..... ((empilha ($20 $19 $17)))
   * 9 ..... ((fim ($18 $23 $22 $21)))

spl>
```

4. EXEMPLO COM FATOR DE CERTEZA "MYCIN-LIKE":

Este exemplo ilustra a possibilidade do sistema funcionar em modo irrevogável e disparar todas as regras segundo uma ordem predeterminada. O cálculo dos fatores de certeza é feito de acordo com o modelo de raciocínio inexato do sistema Mycin [BUCHANAN 85], através de funções definidas pelo usuário. A seqüencialidade das regras é obtida pela sua ordem de entrada. Os fatores de certeza presentes na MT final determinam a solução do problema.

Base de conhecimentos para o problema

```
; regra para criar o estado inicial do problema
(RULE inicio
      IF    Comeco(A)
      THEN
            Tipo(B)
            Estado(C)
            Tom_cor(D)
            Estado_lat(E)
            Cor(F)
            Prop(G)
            Prop(H)
            Prop(I)
            n(B)   = fusca
            c(B)   = 1.0
```

i(B) = 0

fc(B) = 1.0

n(C) = velho

c(C) = 0.7

i(C) = 0

fc(C) = 0.7

n(D) = desbotado

c(D) = 0.8

i(D) = 0

fc(D) = 0.8

n(E) = com_buracos

c(E) = 0.9

i(E) = 0

fc(E) = 0.9

n(F) = azul

c(F) = 1.0

i(F) = 0

fc(F) = 1.0

n(G) = herve

c(G) = 0

i(G) = 0

fc(G) = 0

n(H) = joni

c(H) = 0

i(H) = 0

fc(H) = 0

n(I) = augusto

```
            c(I)  = 0
            i(I)  = 0
            fc(I) = 0
            (HARAKIRI))
; regras que mudam os fatores de certeza e incerteza
(RULE r1
      IF    Estado(X)
            Tom_cor(Y)
            Prop(Z)
            Prop(T)
            Prop(R)
            n(X) = velho
            n(Y) = desbotado
            n(Z) = herve
            n(T) = joni
            n(R) = augusto
      THEN
            c(Z) = (cfc 0.5 c(Z) fc(X) fc(Y))
            i(Z) = (aci c(Z) i(Z))
            fc(Z) = (- c(Z) i(Z))
            c(T) = (cfc 0.8 c(T) fc(X) fc(Y))
            i(T) = (aci c(T) i(T))
            fc(T) = (- c(T) i(T))
            i(R) = (cfc 0.9 i(R) fc(X) fc(Y))
            c(R) = (aci i(R) c(R))
            fc(R) = (- c(R) i(R))
            (HARAKIRI))
```

```
(RULE r2
      IF    Estado_lat(X)
            Prop(Y)
            n(X) = com_buracos
            n(Y) = augusto
      THEN
            c(Y) = (cfc 0.4 c(Y) fc(X))
            i(Y) = (aci c(Y) i(Y))
            fc(Y) = (- c(Y) i(Y))
            (HARAKIRI))
(RULE r3
      IF    Prop(X)
            Prop(Y)
            n(X) = joni
            n(Y) = herve
      THEN
            i(Y) = (cfc 1.0 i(Y) fc(X))
            c(Y) = (aci i(Y) c(Y))
            fc(Y) = (- c(Y) i(Y))
            (HARAKIRI))
(RULE r4
      IF    Cor(X)
            Prop(Y)
            Prop(Z)
            n(X) = azul
            n(Y) = herve
            n(Z) = joni
```

```
      THEN
            c(Y) = (cfc 0.9 c(Y) fc(X))
            i(Y) = (aci c(Y) i(Y))
            fc(Y) = (- c(Y) i(Y))
            i(Z) = (cfc 0.9 i(Z) fc(X))
            c(Z) = (aci i(Z) c(Z))
            fc(Z) = (- c(Z) i(Z))
            (HARAKIRI))
(RULE r5
      IF    Tipo(X)
            Prop(Y)
            n(X) = fusca
            n(Y) = augusto
      THEN
            i(Y) = (cfc 1.0 i(Y) fc(X))
            c(Y) = (aci i(Y) c(Y))
            fc(Y) = (- c(Y) i(Y))
            (HARAKIRI))
; inicialização da MT
(CREATE Comeco)
```

Funções definidas pelo usuário

```
; executam raciocínio inexato de modo análogo ao sistema Mycin
(de cfc l
    (if (equal (abs (car l)) 1)
        (apply 'minr (cddr l))
        (+ (cadr l) (* (car l)
```

```
                        (- l (cadr l))
                        (apply 'minr (cddr l))
        ))))
(de minr l
    (divide  (truncate (apply 'min (mapcar '(lambda (e)
                                        (fix (* 1000 e)))
                                l)))
                1000))
(de aci  (a b)
    (if (equal a 1) 0 b))
```

Exemplo de uma sessão de trabalho

```
spl> (load fusca)
Pre_Compilador      : .\spllib\precomvl.3
Compilador          : .\spllib\compilvl.3
Pre e Compilacao    : .\splbc\fusca.ll
spl> (wm)
  $1 : (Comeco)
spl> (regime)
rev
spl> (regime irr)
irr
spl> (strategy fst)
fst
spl> (run 2)
2
```

```
spl> (wm)
 $12 : (Prop ^n augusto ^c 0 ^i .6291 ^fc -.6291)
 $11 : (Prop ^n joni ^c .5592 ^i 0 ^fc .5592)
 $10 : (Prop ^n herve ^c .3495 ^i 0 ^fc .3495)
  $6 : (Cor ^n azul ^c 1. ^i 0 ^fc 1.)
  $5 : (Estado_lat ^n com_buracos ^c .9 ^i 0 ^fc .9)
  $4 : (Tom_cor ^n desbotado ^c .8 ^i 0 ^fc .8)
  $3 : (Estado ^n velho ^c .7 ^i 0 ^fc .7)
  $2 : (Tipo ^n fusca ^c 1. ^i 0 ^fc 1.)
  $1 : (Comeco)
spl> (run)
6
   *** Situacao Final ***

- ciclos ..... 6
- backtrackings ... 0

   * 1 ..... ((inicio ($1)))
   * 2 ..... ((r1 ($3 $4 $7 $8 $9)))
   * 3 ..... ((r2 ($5 $12)))
   * 4 ..... ((r3 ($11 $10)))
   * 5 ..... ((r4 ($6 $14 $11)))
   * 6 ..... ((r5 ($2 $13)))
spl> (wm)
 $17 : (Prop ^n augusto ^c 0 ^i 1 ^fc -1)
 $16 : (Prop ^n joni ^c .5592 ^i .9 ^fc -.3408)
 $15 : (Prop ^n herve ^c .93495 ^i .559 ^fc .37595)
```

```
   $6 : (Cor ^n azul ^c 1. ^i 0 ^fc 1.)
   $5 : (Estado_lat ^n com_buracos ^c .9 ^i 0 ^fc .9)
   $4 : (Tom_cor ^n desbotado ^c .8 ^i 0 ^fc .8)
   $3 : (Estado ^n velho ^c .7 ^i 0 ^fc .7)
   $2 : (Tipo ^n fusca ^c 1. ^i 0 ^fc 1.)
   $1 : (Comeco)
spl>
```

5. EXEMPLO RACIOCÍNIO PARA TRÁS:

Neste problema o encadeamento regressivo é emulado pelo sistema. O conjunto de regras de decomposição abaixo é utilizado [FARRENY 87]:

r1 : a :=> b , c

r2 : d :=> a, e , f

r3 : d :=> a , k

r4 : f :=> i

r5 : f :=> c , j

r6 : d :=> g , h

r7 : k :=> e , l

Durante o processo de decomposição estes problemas geram uma árvore E/OU de sub-problemas. Para cada sub-problema não resolvido uma questão é feita ao usuário. Caso a resposta seja que o sub-problema não está resolvido, as regras são utilizam para sua resolução. A seqüencialidade necessária no caso das perguntas é obtida através do uso de prioridades.

Base de conhecimentos para o problema

```
; retira um problema se está resolvido
(DEMON deleta-problema
       IF   Problema(P)
            Fato(F)
            nome(F) = nome(P)
```

```
        THEN
            - P)
; Regras de decomposição de problemas
(RULE r1
      IF    Problema(A)
            nome(A) = a
      THEN
          - A
            Problema(B)
            nome(B) = b
            Problema(C)
            nome(C) = c)
(RULE r2
      IF    Problema(D)
            nome(D) = d
      THEN
          - D
            Problema(A)
            nome(A) = a
            Problema(E)
            nome(E) = e
            Problema(F)
            nome(F) = f)
(RULE r3
      IF    Problema(D)
            nome(D) = d
      THEN
```

```
        - D
          Problema(A)
          nome(A) = a
          Problema(K)
          nome(K) = k)
(RULE r4
     IF   Problema(F)
          nome(F) = f
     THEN
        - F
          Problema(I)
          nome(I) = i)
(RULE r5
     IF   Problema(F)
          nome(F) = f
     THEN
        - F
          Problema(C)
          nome(C) = c
          Problema(J)
          nome(J) = j)
(RULE r6
     IF   Problema(D)
          nome(D) = d
     THEN
        - D
          Problema(G)
```

```
            nome(G) = g
            Problema(H)
            nome(H) = h)
(RULE r7
      IF    Problema(K)
            nome(K) = k
      THEN
         - K
            Problema(E)
            nome(E) = e
            Problema(L)
            nome(L) = l)
; pergunta ao usuário se o problema está resolvido
(DEMON pergunta
       IF   Problema(P)
          - Fato(F)
            nome(F) = nome(P)
       THEN
            Resposta(R)
            objeto(R) = nome(P)
            valor(R) =
                (accept "Problema" nome(P)"resolvido (yes/no)")
            (PRIORITY resposta 0)
            (PRIORITY resposta-nao 0)
            (HARAKIRI))
; tratamento da resposta
(RULE resposta -1
```

```
      IF    Resposta(R)
            valor(R) = yes
      THEN
          - R
            Fato(F)
            nome(F) = objeto(R)
            (PRIORITY pergunta 0)
            (HARAKIRI))
(RULE resposta-nao -1
      IF    Resposta(R)
          - valor(R) = no
      THEN
          - R
            (PRIORITY pergunta 0)
            (HARAKIRI))
; condição de parada
(DEMON fim
       IF   - Problema(P)
       THEN
            (WRITE "Ate que enfim cheguei ao fim ! ")
            (STOP))
; inicialização da MT
(CREATE Fato ^nome g)
(CREATE Fato ^nome c)
(CREATE Fato ^nome b)
(CREATE Problema ^nome d)
```

Exemplo de uma sessão de trabalho

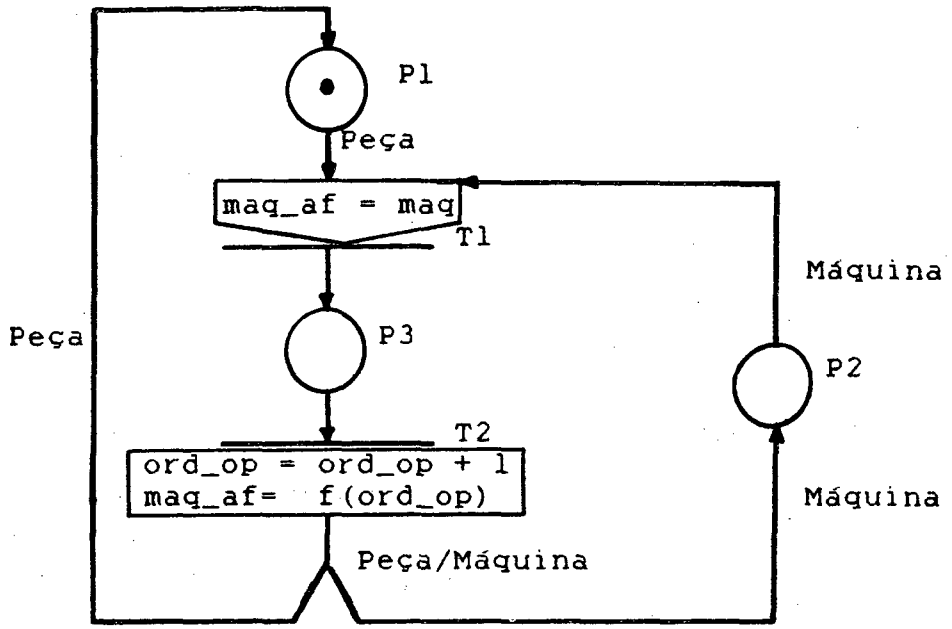
```
spl> (load eou)
Pre_Compilador       : .\spllib\precomvl.3
Compilador           : .\spllib\compilvl.3
Pre e Compilacao     : .\splbc\eou.ll
spl> (wm)
  $4 : (Problema ^nome d)
  $3 : (Fato ^nome b)
  $2 : (Fato ^nome c)
  $1 : (Fato ^nome g)
spl> (run 5)
Problema  d  resolvido (yes/no) ? no
3
Problema  f  resolvido (yes/no) ? no
5
spl> (wm)
  $8 : (Problema ^nome f)
  $7 : (Problema ^nome e)
  $6 : (Problema ^nome a)
  $3 : (Fato ^nome b)
  $2 : (Fato ^nome c)
  $1 : (Fato ^nome g)
spl> (cs)
((r1 ($6)) (r4 ($8)) (r5 ($8)) (pergunta ($7 -) ($6 -)))
```

```
spl> (run)
Problema  e  resolvido (yes/no) ? yes
7
Problema  a  resolvido (yes/no) ? yes
12
Problema  i  resolvido (yes/no) ? no
18
Problema  j  resolvido (yes/no) ? yes
22
Ate que enfim cheguei ao fim !
23

   *** Solucao ***

- ciclos ..... 23
- backtrackings ... 3

   * 1 ..... ((pergunta ($4 -)))
   * 2 ..... ((resposta-nao ($5)))
   * 3 ..... ((r2 ($4)))
   * 4 ..... ((pergunta ($8 -)))
   * 5 ..... ((resposta-nao ($9)))
   * 6 ..... ((pergunta ($7 -)))
   * 7 ..... ((resposta ($10)))
   * 8 ..... ((pergunta ($6 -)))
   * 9 ..... ((deleta-problema ($7 $11)))
   * 10 ..... ((resposta ($12)))
```

```
    * 11 ..... ((deleta-problema ($6 $13)))
    * 12 ..... ((r5 ($8)) (r4 ($8)))
    * 13 ..... ((pergunta ($15 -)))
    * 14 ..... ((deleta-problema ($14 $2)))
    * 15 ..... ((resposta ($16)))
    * 16 ..... ((deleta-problema ($15 $17)))
    * 17 ..... ((fim (-)))
spl> (wm)
 $17 : (Fato ^nome j)
  $1 : (Fato ^nome g)
  $2 : (Fato ^nome c)
  $3 : (Fato ^nome b)
 $11 : (Fato ^nome e)
 $13 : (Fato ^nome a)
```

6. REDE DE PETRI PEÇA-MÁQUINA:

Neste exemplo o SP1 descreve a rede de Petri a objetos abaixo que representa a especificação do módulo de controle da execução de um plano de fabricação [BAKO 89].



Duas regras de produção modelam os processos de usinagem e fim de usinagem. Os objetos caracterizam as peças e as máquinas utilizadas. As peças são sucessivamente usinadas pelas máquinas numa ordem determinada.

Base de conhecimentos para o problema

```
; modelagem do processo
(RULE inicia-usinagem
      IF
          Peca (X)
```

```
            Maq  (Y)
            maq-af (X) = nome (Y)
        THEN
            Usinagem (Z)
            peca-usi (Z) = X
            maq-usi  (Z) = Y
            - X
            - Y)
(RULE fim-usinagem
        IF
            Usinagem (Z)
        THEN
            ord-op (peca-usi (Z)) = (1+ ord-op (peca-usi (Z)))
            maq-af (peca-usi (Z)) = (f  ord-op (peca-usi (Z)))
            + Y = peca-usi (Z)
            + X = maq-usi  (Z)
            - Z)
; inicialização da MT
(CREATE Peca ^nome p1 ^ord-op 1 ^maq-af m1)
(CREATE Peca ^nome p2 ^ord-op 1 ^maq-af m1)
(CREATE Peca ^nome p3 ^ord-op 1 ^maq-af m1)
(CREATE Peca ^nome p4 ^ord-op 1 ^maq-af m1)
(CREATE Maq  ^nome m1)
(CREATE Maq  ^nome m2)
(CREATE Maq  ^nome m3)
```

Função definida pelo usuário

```
; altera a máquina a ser usada pela peça
(de f (n)
    (cond ((equal n 1) 'm1)
          ((equal n 2) 'm2)
          ((equal n 3) 'm3)
          (t ()) ))
```

Exemplo de uma sessão de trabalho

```
spl> (load pec-maq2)
Pre-compilador        : .\spllib\precomvl.3
Compilador            : .\spllib\compilvl.3
Pre e compilacao      : .\splbc\pec-maq2.11
spl> (wm)
  $7 : (Maq ^nome m3)
  $6 : (Maq ^nome m2)
  $5 : (Maq ^nome m1)
  $4 : (Peca ^nome p4 ^ord-op 1 ^maq-af m1)
  $3 : (Peca ^nome p3 ^ord-op 1 ^maq-af m1)
  $2 : (Peca ^nome p2 ^ord-op 1 ^maq-af m1)
  $1 : (Peca ^nome p1 ^ord-op 1 ^maq-af m1)
spl> (rm)
((D : ) (R :  inicia-usinagem fim-usinagem)
spl> (run 1)
1
```

```
spl> (wm)
  $8 : (Usinagem ^peca-usi $4 ^maq-usi $5)
  $7 : (Maq ^nome m3)
  $6 : (Maq ^nome m2)
  $3 : (Peca ^nome p3 ^ord-op 1 ^maq-af m1)
  $2 : (Peca ^nome p2 ^ord-op 1 ^maq-af m1)
  $1 : (Peca ^nome p1 ^ord-op 1 ^maq-af m1)
spl> (run 1)
2
spl> (wm)
 $12 : (Maq ^nome m1)
 $11 : (Peca ^nome p4 ^ord-op 2 ^maq-af m2)
  $7 : (Maq ^nome m3)
  $6 : (maq ^nome m2)
  $3 : (Peca ^nome p3 ^ord-op 1 ^maq-af m1)
  $2 : (Peca ^nome p2 ^ord-op 1 ^maq-af m1)
  $1 : (Peca ^nome p1 ^ord-op 1 ^maq-af m1)
spl> (cs)
((inicia-usinagem ($3 $12) ($2 $12) ($1 $12) ($11 $6)))
spl> (regime irr)
irr
spl> (run)
24

   *** Situacao Final ***

- ciclos ..... 24
```

- backtrackings ... 0

* 1 ..... ((inicia-usinagem ($4 $5)))
* 2 ..... ((fim-usinagem ($8)))
* 3 ..... ((inicia-usinagem ($3 $12)))
* 4 ..... ((fim-usinagem ($13)))
* 5 ..... ((inicia-usinagem ($2 $17)))
* 6 ..... ((fim-usinagem ($18)))
* 7 ..... ((inicia-usinagem ($1 $22)))
* 8 ..... ((fim-usinagem ($23)))
* 9 ..... ((inicia-usinagem ($26 $6)))
* 10 ..... ((fim-usinagem ($28)))
* 11 ..... ((inicia-usinagem ($21 $32)))
* 12 ..... ((fim-usinagem ($33)))
* 13 ..... ((inicia-usinagem ($16 $37)))
* 14 ..... ((fim-usinagem ($38)))
* 15 ..... ((inicia-usinagem ($11 $42)))
* 16 ..... ((fim-usinagem ($43)))
* 17 ..... ((inicia-usinagem ($46 $7)))
* 18 ..... ((fim-usinagem ($48)))
* 19 ..... ((inicia-usinagem ($41 $52)))
* 20 ..... ((fim-usinagem ($53)))
* 21 ..... ((inicia-usinagem ($36 $57)))
* 22 ..... ((fim-usinagem ($58)))
* 23 ..... ((inicia-usinagem ($31 $62)))
* 24 ..... ((fim-usinagem ($63)))

```
spl> (wm)
 $67 : (Maq ^nome m3)
 $66 : (Peca ^nome p1 ^ord-op 4 ^maq-af ())
 $61 : (Peca ^nome p2 ^ord-op 4 ^maq-af ())
 $56 : (Peca ^nome p3 ^ord-op 4 ^maq-af ())
 $51 : (Peca ^nome p4 ^ord-op 4 ^maq-af ())
 $47 : (Maq ^nome m2)
 $27 : (Maq ^nome m1)
spl>
```

7. SIMULAÇÃO DE MÁQUINA DE TURING:

Neste exemplo uma máquina de Turing é simulada pelo SP1. O exemplo retirado de [HOPCROFT 79], utiliza uma máquina de Turing modificada, que realiza em todo passo uma operação de leitura e escrita na fita. Esta máquina aceita cadeias de caracteres constituidos por um número idêntico de zeros e uns.

O exemplo mostra que o SP1 é computacionalmente equivalente a uma máquina de Turing.

Base de conhecimentos para o problema

```
; movimentos da máquina
(RULE left
     IF
          Estado(X)
          Cabeca(Y)
          Fita(Z)
          Quadrupla(T)
          cell(Y) = cell(Z)
          s(T) = s(X)
          c(T) = c(Z)
          a(T) = left
     THEN
          s(X) = n(T)
          cell(Y) = (1- cell(Y))
          c(Z) = e(T))
```

```
(RULE right
      IF
            Estado(X)
            Cabeca(Y)
            Fita(Z)
            Quadrupla(T)
            cell(Y) = cell(Z)
            s(T) = s(X)
            c(T) = c(Z)
            a(T) = right
      THEN
            s(X) = n(T)
            cell(Y) = (1+ cell(Y))
            c(Z) = e(T))
; condição de parada
(DEMON fim
       IF
            Estado(T)
            s(T) = q4
       THEN
            (STOP))
; inicialização da MT
(CREATE Cabeca      ^cell 0)
(CREATE Fita        ^cell 0 ^c 0)
(CREATE Fita        ^cell 1 ^c 0)
(CREATE Fita        ^cell 2 ^c 1)
(CREATE Fita        ^cell 3 ^c 1)
```

```
(CREATE Fita        ^cell 4 ^c b)
(CREATE Quadrupla   ^s q0 ^c 0 ^n q1 ^e x ^a right)
(CREATE Quadrupla   ^s q1 ^c 0 ^n q1 ^e 0 ^a right)
(CREATE Quadrupla   ^s q2 ^c 0 ^n q2 ^e 0 ^a left)
(CREATE Quadrupla   ^s q1 ^c 1 ^n q2 ^e y ^a left)
(CREATE Quadrupla   ^s q2 ^c x ^n q0 ^e x ^a right)
(CREATE Quadrupla   ^s q0 ^c y ^n q3 ^e y ^a right)
(CREATE Quadrupla   ^s q1 ^c y ^n q1 ^e y ^a right)
(CREATE Quadrupla   ^s q2 ^c y ^n q2 ^e y ^a left)
(CREATE Quadrupla   ^s q3 ^c y ^n q3 ^e y ^a right)
(CREATE Quadrupla   ^s q3 ^c b ^n q4 ^e b ^a right)
(CREATE Estado      ^s q0)
```

Exemplo de uma sessão de trabalho

```
spl> (load turing)
Pre-compilador          : .\spllib\precomv1.3
Compilador              : .\spllib\compilv1.3
Pre e compilacao        : .\splbc\turing.l1
spl> (wm)
 $17 : (Estado ^s q0)
 $16 : (Quadrupla ^s q3 ^c b ^n q4 ^e b ^a right)
 $15 : (Quadrupla ^s q3 ^c y ^n q3 ^e y ^a right)
 $14 : (Quadrupla ^s q2 ^c y ^n q2 ^e y ^a left)
 $13 : (Quadrupla ^s q1 ^c y ^n q1 ^e y ^a right)
 $12 : (Quadrupla ^s q0 ^c y ^n q3 ^e y ^a right)
 $11 : (Quadrupla ^s q2 ^c x ^n q0 ^e x ^a right)
```

```
 $10 : (Quadrupla ^s q1 ^c 1 ^n q2 ^e y ^a left)
  $9 : (Quadrupla ^s q2 ^c 0 ^n q2 ^e 0 ^a left)
  $8 : (Quadrupla ^s q1 ^c 0 ^n q1 ^e 0 ^a right)
  $7 : (Quadrupla ^s q0 ^c 0 ^n q1 ^e x ^a right)
  $6 : (Fita ^cell 4 ^c b)
  $5 : (Fita ^cell 3 ^c 1)
  $4 : (Fita ^cell 2 ^c 1)
  $3 : (Fita ^cell 1 ^c 0)
  $2 : (Fita ^cell 0 ^c 0)
  $1 : (Cabeca ^cell 0)
spl> (rm)
((D :  fim) (R :  left right))
spl> (run 1)
1
spl> (wm)
 $20 : (Fita ^cell 0 ^c x)
 $19 : (Cabeca ^cell 1)
 $18 : (Estado ^s q1)
 $16 : (Quadrupla ^s q3 ^c b ^n q4 ^e b ^a right)
 $15 : (Quadrupla ^s q3 ^c y ^n q3 ^e y ^a right)
 $14 : (Quadrupla ^s q2 ^c y ^n q2 ^e y ^a left)
 $13 : (Quadrupla ^s q1 ^c y ^n q1 ^e y ^a right)
 $12 : (Quadrupla ^s q0 ^c y ^n q3 ^e y ^a right)
 $11 : (Quadrupla ^s q2 ^c x ^n q0 ^e x ^a right)
 $10 : (Quadrupla ^s q1 ^c 1 ^n q2 ^e y ^a left)
  $9 : (Quadrupla ^s q2 ^c 0 ^n q2 ^e 0 ^a left)
  $8 : (Quadrupla ^s q1 ^c 0 ^n q1 ^e 0 ^a right)
```

```
   $7 : (Quadrupla ^s q0 ^c 0 ^n q1 ^e x ^a right)
   $6 : (Fita ^cell 4 ^c b)
   $5 : (Fita ^cell 3 ^c 1)
   $4 : (Fita ^cell 2 ^c 1)
   $3 : (Fita ^cell 1 ^c 0)
spl> (run 6)
7
spl> (wm)
  $38 : (Fita ^cell 2 ^c y)
  $37 : (Cabeca ^cell 3)
  $36 : (Estado ^s q1)
  $35 : (Fita ^cell 1 ^c x)
  $32 : (Fita ^cell 0 ^c x)
  $16 : (Quadrupla ^s q3 ^c b ^n q4 ^e b ^a right)
  $15 : (Quadrupla ^s q3 ^c y ^n q3 ^e y ^a right)
  $14 : (Quadrupla ^s q2 ^c y ^n q2 ^e y ^a left)
  $13 : (Quadrupla ^s q1 ^c y ^n q1 ^e y ^a right)
  $12 : (Quadrupla ^s q0 ^c y ^n q3 ^e y ^a right)
  $11 : (Quadrupla ^s q2 ^c x ^n q0 ^e x ^a right)
  $10 : (Quadrupla ^s q1 ^c 1 ^n q2 ^e y ^a left)
   $9 : (Quadrupla ^s q2 ^c 0 ^n q2 ^e 0 ^a left)
   $8 : (Quadrupla ^s q1 ^c 0 ^n q1 ^e 0 ^a right)
   $7 : (Quadrupla ^s q0 ^c 0 ^n q1 ^e x ^a right)
   $6 : (Fita ^cell 4 ^c b)
   $5 : (Fita ^cell 3 ^c 1)
spl> (run)
14
```

```
   *** Solucao ***

- ciclos .......... 14
- backtrackings ... 0

   * 1 ..... ((right ($17 $1 $2 $7)))
   * 2 ..... ((right ($18 $19 $3 $8)))
   * 3 ..... ((left ($21 $22 $4 $10)))
   * 4 ..... ((left ($24 $25 $23 $9)))
   * 5 ..... ((right ($27 $28 $20 $11)))
   * 6 ..... ((right ($30 $31 $29 $7)))
   * 7 ..... ((right ($33 $34 $26 $13)))
   * 8 ..... ((left ($36 $37 $5 $10)))
   * 9 ..... ((left ($39 $40 $38 $14)))
   * 10 ..... ((right ($42 $43 $35 $11)))
   * 11 ..... ((right ($45 $46 $44 $12)))
   * 12 ..... ((right ($48 $49 $41 $15)))
   * 13 ..... ((right ($51 $52 $6 $16)))
   * 14 ..... ((fim ($54)))

spl>
```

8. CÁLCULO DE RAÍZES DO SEGUNDO GRAU:

O problema corresponde a aplicação da fórmula de Baskara sobre uma equação de parâmetros fornecidos pelo usuário. A seqüencialidade é obtida através de objetos presentes na MT (como e.g. Continua) e através das prioridades das regras (ação HARAKIRI na regra "disc" e REVIVE na regra "continua2"). Desta forma é possível a obtenção de uma seqüência de disparo que permite emular o comportamento algorítmico desejado.

Base de conhecimentos para o problema

```
; entrada dos dados
(RULE inicio
      IF   Comeco(X)
      THEN
         - X
           Equacao(Y)
           a(Y) = (accept "Termo do 2. grau  ")
           b(Y) = (accept "Termo do 1. grau  ")
           c(Y) = (accept "Termo independente"))
; verificação se a equação é do 1 o. grau
(RULE grau_um
      IF   Equacao(X)
           a(X) = 0
      THEN
           r1(X) = (/ (- c(X)) b(X))
```

```
            (WRITE "Equacao do primeiro grau raiz : " r1(X))
            Continua(Y))
; regras que descrevem a fórmula de Baskara
(RULE disc
      IF    Equacao(X)
            a(X) <> 0
      THEN
            d(X) = (- (* b(X) b(X)) (* 4 a(X) c(X)))
            (HARAKIRI))
(RULE raiz_unica
      IF    Equacao(X)
            d(X) = 0
      THEN
            r1(X) = (/ (- b(X)) (* 2 a(X)))
            (WRITE "Raiz unica : " r1(X) )
            Continua(Y))
(RULE raiz_normal
      IF    Equacao(X)
            d(X) > 0
      THEN
            r1(X) = (/ (+ (- b(X)) (sqrt d(X))) (* 2 a(X)))
            r2(X) = (/ (- (- b(X)) (sqrt d(X))) (* 2 a(X)))
            (WRITE "Raiz 1 : " r1(X))
            (WRITE "Raiz 2 : " r2(X))
            Continua(Y))
(RULE raiz_imag
      IF    Equacao(X)
```

```
            d(X) < 0
      THEN
            (WRITE "Nao ha raiz real")
            Continua(Y))
; verificação se deve haver continuação
(RULE continua
      IF    Continua(X)
      THEN
            sim(X) = (accept "Deseja continuar (sim/nao)"))
(DEMON continua2
      IF    Continua(X)
            sim(X) = sim
      THEN
         - X
            (REVIVE disc)
            Comeco(Y))
; final de sessão
(DEMON para
      IF    Continua(X)
            sim(X) = nao
      THEN
         - X
            (STOP))
; inicialização da MT
(CREATE Comeco)
```

Exemplo de uma sessão de trabalho

```
spl> (start)
t
spl> (load baskara)
Pre-Compilador      : .\spllib\precomvl.3
Compilador          : .\spllib\compilvl.3
Pre e Compilacao    : .\splbc\baskara.ll
spl> (wm)
  $1 : (Comeco)
spl> (rm)
((D :   continua2  para) (R  :   inicio grau_um  disc  raiz_unica
raiz_normal
raiz_imag continua))
spl> (cs)
((inicio ($1)))
spl> (run)
Termo do 2. grau   ? 1
Termo do 1. grau   ? 5
Termo independente ? 6
2
Raiz 1 : -2.

Raiz 2 : -3.
3
Deseja continuar (sim/nao) ? sim
5
```

```
Termo do 2. grau   ? 0
Termo do 1. grau   ? 1
Termo independente ? 1
6
Equacao do primeiro grau raiz -1
7
Deseja continuar (sim/nao) ? sim
9
Termo do 2. grau   ? 1
Termo do 1. grau   ? 6
Termo independente ? 6
11
Raiz 1 : -1.267949

Raiz 2 : -4.732051
12
Deseja continuar (sim/nao) ? sim
14
Termo do 2. grau   ? 2
Termo do 1. grau   ? 2
Termo independente ? 2
16
Nao ha raiz real
17
Deseja continuar (sim/nao) ? nao
19
```

```
   *** Solucao ***


- ciclos ..... 19
- backtrackings ... 0


   * 1 ..... ((inicio ($1)))
   * 2 ..... ((disc ($2)))
   * 3 ..... ((raiz_normal ($3)))
   * 4 ..... ((continua ($5)))
   * 5 ..... ((continua2 ($6)))
   * 6 ..... ((inicio ($7)))
   * 7 ..... ((grau_um ($8)))
   * 8 ..... ((continua ($10)))
   * 9 ..... ((continua2 ($11)))
   * 10 ..... ((inicio ($12)))
   * 11 ..... ((disc ($13)))
   * 12 ..... ((raiz_normal ($14)))
   * 13 ..... ((continua ($16)))
   * 14 ..... ((continua2 ($17)))
   * 15 ..... ((inicio ($18)))
   * 16 ..... ((disc ($19)))
   * 17 ..... ((raiz_imag ($20)))
   * 18 ..... ((continua ($21)))
   * 19 ..... ((para ($22)))
```

```
spl> (wm)
 $20 : (Equacao ^d -12 ^a 2 ^b 2 ^c 2)
 $15 : (Equacao ^r2 -4.732051 ^r1 -1.267949 ^d 12 ^a 1 ^b 6 ^c 6)
  $9 : (Equacao ^r1 -1 ^a 0 ^b 1 ^c 1)
  $4 : (Equacao ^r2 -3. ^r1 -2. ^d 1 ^a 1 ^b 5 ^c 6)
 spl>
```

9. CONSTRUÇÃO DE UMA LISTA LIGADA:

Este exemplo a proporciona a criação de uma lista ligada através de dois mecanismos: (a) pela utilização da composição de atributos presente no SP1; e (b) através da utilização de um rótulo comum entre dois objetos. Em cada caso o acréscimo e a retirada de novos elementos é representado por uma regra. O exemplo também ilustra a utilização da ação BIND, utilizada em conjunto com a função LISP gensym (função física) e permitindo a obtenção da seqüencialidade desejada.

Base de conhecimentos para o problema

```
; atribuição de valores ao símbolo @no
(SET @no no)
(SET no 0)
; criação do primeiro elemento da lista
(RULE inicio
      IF    Comeco(X)
      THEN
         - X
           No(Y)
           nome(Y) = (incr @no)
           prox(Y) = nil)
; inclusão no comeco da lista
(RULE coloca_comeco
      IF    No(X)
```

```
          - No(Y)
            prox(Y) = X
      THEN
            No(Z)
            nome(Z) = (incr @no)
            prox(Z) = X)
; inclusão no fim da lista
(RULE coloca_fim
      IF    No(X)
            prox(X) = nil
      THEN
            No(Y)
            nome(Y) = (incr @no)
            prox(Y) = espera
            (PRIORITY coloca_fim_2 0))
(RULE coloca_fim_2 -1
      IF    No(Y)
            No(X)
            prox(X) = nil
            prox(Y) = espera
      THEN
            prox(X) = Y
            prox(Y) = nil
            (HARAKIRI))
; outro modo de incluir na lista
(RULE coloca2
      IF    No(X)
```

```
            prox(X) = nil
      THEN
            (BIND Z (gensym))
            No(Y)
            prox(Y) = nil
            label(Y) = Z
            nome(Y) = (incr @no)
            prox(X) = label(Y))
; retirada da lista primeiro tipo
(RULE retira_lista
      IF    No(X)
            No(Y)
            prox(Y) = X
      THEN
            prox(Y) = prox(X)
         - X)
; retirada da lista segundo tipo
(RULE retira_lista2
      IF    No(X)
            No(Y)
            prox(X) = label(Y)
      THEN
            prox(X) = nil
         - Y)
```

Exemplo de uma sessão de trabalho

```
spl> (load lista)
Pre-compilador      : .\spllib\precomv1.3
Compilador          : .\spllib\compilv1.3
Pre e compilacao    : .\splbc\lista.ll
spl> (create Comeco)
()
spl> (run 1)
1
spl> (wm)
  $2 : (No ^nome 1 ^prox nil)
spl> (manual)
Conjunto de Conflitos
1  :  coloca_comeco  :  1  :  ($2 -)
2  :  coloca_fim  :  1  :  ($2)
3  :  coloca2  :  1  :  ($2)
Dispara qual regra/instancia ?
1
Disparada  :  coloca_comeco  :  ($2 -)
spl> (wm)
  $3 : (No ^nome 2 ^prox $2)
  $2 : (No ^nome 1 ^prox nil)
spl> (manual)
Conjunto de Conflitos
1  :  coloca_comeco  :  1  :  ($3 -)
2  :  coloca_fim  :  1  :  ($2)
```

```
3  :  coloca2  :  1  :  ($2)
4  :  retira_lista  :  1  :  ($2 $3)
Dispara qual regra/instancia ?
2
Disparada  :  coloca_fim  :  ($2)
spl> (wm)
  $4 : (No ^nome 3 ^prox espera)
  $3 : (No ^nome 2 ^prox $2)
  $2 : (No ^nome 1 ^prox nil)
spl> (cs)
((coloca_comeco ($4 -) ($3 -)) (coloca_fim_2 ($4 $2)) (coloca2
($2)) (retira_lista ($2 $3)))
spl> (run 1)
4
spl> (wm)
  $6 : (No ^nome 3 ^prox nil)
  $5 : (No ^nome 1 ^prox $6)
  $3 : (No ^nome 2 ^prox $5)
spl> (manual)
Conjunto de Conflitos
1  :  coloca_comeco  :  1  :  ($3 -)
2  :  coloca_fim  :  1  :  ($6)
3  :  coloca2  :  1  :  ($6)
4  :  retira_lista  :  1  :  ($6 $5)
5  :  retira_lista  :  1  :  ($5 $3)
Dispara qual regra/instancia ?
5
```

```
Disparada  :  retira_lista  :  ($5 $3)
spl> (wm)
  $7 : (No ^nome 2 ^prox $6)
  $6 : (No ^nome 3 ^prox nil)
spl> (manual)
Conjunto de Conflitos
1  :  coloca_comeco  :  1  :  ($7 -)
2  :  coloca_fim  :  1  :  ($6)
3  :  coloca2  :  1  :  ($6)
4  :  retira_lista  :  1  :  ($6 $7)
Dispara qual regra/instancia ?
3
Disparada  :  coloca2  :  ($6)
spl> (wm)
  $9 : (No ^nome 3 ^prox g104)
  $8 : (No ^prox nil ^label g104 ^nome 4)
  $7 : (No ^nome 2 ^prox $9)
spl> (manual)
Conjunto de Conflitos
1  :  coloca_comeco  :  1  :  ($8 -)
2  :  coloca_comeco  :  1  :  ($7 -)
3  :  coloca_fim  :  1  :  ($8)
4  :  coloca2  :  1  :  ($8)
5  :  retira_lista  :  1  :  ($9 $7)
6  :  retira_lista2  :  1  :  ($9 $8)
Dispara qual regra/instancia ?
6
```

```
Disparada   :   retira_lista2   :   ($9 $8)
spl> (wm)
 $10 : (No ^nome 3 ^prox nil)
  $7 : (No ^nome 2 ^prox $10)
spl>
```

10. PROBLEMA DAS n RAINHAS:

Este exemplo mostra como é possível resolver através do SP1 o problema clássico da colocação de n rainhas em um tabuleiro n X n, sem que nenhuma possa ameaçar qualquer outra.

A estratégia de resolução adotada é do tipo geração de hipótese e teste. As rainhas são colocadas sucessivamente linha após linha, tomando em conta o fato que duas rainhas jamais podem ficar numa mesma linha. Caso durante a resolução a configuração anterior das linhas não permita a colocação de uma rainha na linha corrente, uma ação de retrocesso seletivo (RETURN-OVER) faz com que a posição das rainhas já colocadas seja reconsiderada.

Base de conhecimentos para o problema

```
; inicialização: entrada do número de rainhas
(DEMON inicio
       IF
           Comeco(X)
       THEN
           t(X) = (accept "numero de rainhas")
           Hipo(Y)
           x(Y) = 1
           y(Y) = 1
           (HARAKIRI))
```

```
; colocação de uma rainha se possível
(RULE poe_rainha
      IF
          Hipo(X)
        - Queen(Y)
          y(Y) Or (= y(X))
                   (= (+ y(X) (- x(Y) x(X))))
                   (= (+ y(X) (- x(X) x(Y))))
      THEN
          Queen(T)
          x(T) = x(X)
          y(T) = y(X)
          x(X) = (1+ x(X))
          y(X) = 1)
; alteração a hipótese
(RULE muda_hip  2
      IF
          Hipo(X)
      THEN
          y(X) = (1+ y(X)))
; não é possível rainha nesta coluna: retrocesso
(DEMON fim_coluna
       IF
           Hipo(X)
           Comeco(Y)
           Queen(W)
           y(X) > t(Y)
```

```
        THEN
             (RETURN-OVER W))
; todas as rainhas colocadas
(DEMON fim
        IF
             Hipo(X)
             Comeco(Y)
             x(X)    t(Y)
        THEN
          - X
             (STOP))
; inicialização da MT
(CREATE Comeco)
```

Exemplo de uma sessão de trabalho

```
spl> (run 1)
numero de rainhas ? 4
1
spl> (wm)
  $3 : (Hipo ^x 1 ^y 1)
  $2 : (Comeco ^t 4)
spl> (cs)
((poe_rainha ($3 -)) (muda_hip ($3)))
spl> (run 1)
2
```

```
spl> (wm)
  $5 : (Hipo ^x 2 ^y 1)
  $4 : (Queen ^x 1 ^y 1)
  $2 : (Comeco ^t 4)
spl> (cs)
((muda_hip ($5)))
spl> (run 2)
4
spl> (wm)
  $7 : (Hipo ^x 2 ^y 3)
  $4 : (Queen ^x 1 ^y 1)
  $2 : (Comeco ^t 4)
spl> (cs)
((poe_rainha ($7 -)) (muda_hip ($7)))
spl> (run 1)
5
spl> (wm)
  $9 : (Hipo ^x 3 ^y 1)
  $8 : (Queen ^x 2 ^y 3)
  $4 : (Queen ^x 1 ^y 1)
  $2 : (Comeco ^t 4)
spl> (cs)
((muda_hip ($9)))
spl> (run 4)
9
```

```
spl> (wm)
 $13 : (Hipo ^x 3 ^y 5)
  $8 : (Queen ^x 2 ^y 3)
  $4 : (Queen ^x 1 ^y 1)
  $2 : (Comeco ^t 4)
spl> (cs)
((poe_rainha ($13 -)) (muda_hip ($13))
 (fim_coluna ($13 $2 $8) ($13 $2 $4)))
spl> (run 1)
10
spl> (wm)
  $2 : (Comeco ^t 4)
  $4 : (Queen ^x 1 ^y 1)
  $7 : (Hipo ^x 2 ^y 3)
spl> (cs)
((muda_hip ($7)) (poe_rainha ($7 -)))
spl> (run)
34

   *** Solucao ***

- ciclos ..... 34
- backtrackings ... 3

   * 1 ..... ((inicio ($1)))
   * 2 ..... ((muda_hip ($3)) (poe_rainha ($3 -)))
   * 3 ..... ((poe_rainha ($4 -)))
```

```
   * 4 ..... ((muda_hip ($6)))
   * 5 ..... ((muda_hip ($7)))
   * 6 ..... ((muda_hip ($8)))
   * 7 ..... ((poe_rainha ($9 -)))
   * 8 ..... ((poe_rainha ($11 -)))
   * 9 ..... ((muda_hip ($13)))
   * 10 ..... ((muda_hip ($14)))
   * 11 ..... ((poe_rainha ($15 -)))
   * 12 ..... ((fim ($17 $2)))

spl> (wm)
 $16 : (Queen ^x 4 ^y 3)
 $12 : (Queen ^x 3 ^y 1)
 $10 : (Queen ^x 2 ^y 4)
  $5 : (Queen ^x 1 ^y 2)
  $2 : (Comeco ^t 4)
spl>
```

REFERÊNCIAS BIBLIOGRÁFICAS

[AHO 72]

AHO A., ULLMAN J., "The theory of parsing, translation and compiling - Vol. 1 parsing", Prentice-Hall, 1972.

[AHO 73]

AHO A., ULLMAN J., "The theory of parsing, translation and compiling - Vol. 2 compiling", Prentice-Hall, 1973.

[AIKINS 83]

AIKINS J., "Prototypical knowledge for expert systems", Artificial Intelligence 20, pp. 163-210, 1983.

[BAKO 89]

BAKO B., VALETTE R., "Systémes de compilation de régles et reseaux de Petri a objets", Congres International des systemes experts, Avignon, 1989

[BARR 86]

BARR A., FEIGENBAUM E. (eds.), "The handbook of artificial intelligence", vols. 1 e 2, Addison-Wesley, 1986.

[BLAIR 88]

BLAIR H., SUBRAHMANIAN V., "Paraconsistent logic programming", submetido para publicação no 7 o. Int. Conf. on Foundations of Software Tech. & Theoret. Comput. Sci., 1988.

[BROWNSTON 86]

BROWNSTON L., FARRELL R., KANT E., MARTIN N., "Programming expert systems in OPS5", Addison-Wesley, 1986.

[BUCHANAN 78]

BUCHANAN B., FEIGENBAUM E., "Dendral and Meta-dendral", Artificial Intelligence 11, pp. 5-24, 1978.

[BUCHANAN 85]

BUCHANAN B.G., SHORTLIFFE E.H., "Rulebased expert systems : the MYCIN experiments of the Stanford heuristic programming project", Addison-Wesley, 1985.

[CASANOVA 87]

CASANOVA M., GIORGIO F., FURTADO A., "Programação em lógica e a linguagem PROLOG", Edgard Blücher, 1987.

[CHARNIAK 86]

CHARNIAK E., McDERMOTT D.,"Introduction to Artificial Intelligence", Addison-Wesley, 1986.

[CUNHA 87]

CUNHA H., RIBEIRO S., "Introdução aos sistemas especialistas", Livros Técnicos e Científicos, 1987.

[DATE 82]

DATE C.J., "An introduction to database systems", Addison-Wesley, 3 ed., 1982.

[DAVIS 80]

DAVIS R., "Meta-rules : reasoning about control", Artificial Intelligence 15, pp. 179-222, 1980.

[DONOVAN 84]

DONOVAN J., "Systems programming", McGraw-Hill, 1984.

[ERMAN 81]

ERMAN L., LONDON P., FICKAS S., "The design and an example use of the Hearsay-III", Proc. 7 o. IJCAI, pp. 409-415, Vancouver, 1981.

[FARRENY 87]

FARRENY H., GHALLAB M., "Eléménts d'intelligence artificielle", Hermes P. Co., 1987.

[FORGY 81]

FORGY C.L., "OPS5 user's manual", Dept. of Computer Science, Carnegie-Mellon University, 1981.

[FORGY 82]

Forgy C.L., "Rete : a fast algorithm for the many pattern/many object pattern match problem", Artificial Intelligence 19, 17-37, 1982.

[GARNOUSSET 88]

GARNOUSSET H.E., KAESTNER C.A., "SPl : motor de inferência para sistemas de regras de produção", 5 o. SBIA, Natal, 1988.

[GARNOUSSET 89]

GARNOUSSET H.E., KAESTNER C.A., "Une implementation du système de production SPl avec compilation des régles", submetido ao 9 th. International Workshop on Expert Systems ans their applications", Avignon, France, 1989.

[GEORGEFF 82]

M.P. GEORGEFF, "Procedural control in production systems", Artificial Intelligence 18, 175-201, 1982.

[GHALLAB 81]

GHALLAB M.,"Decision trees for optimizing pattern-matching algorithms in production systems", 7o. IJCAI, 310-312, Vancouver, 1981.

[GHALLAB 84]

GHALLAB M., DUFRESNE P., "Moteurs d'inference pour systèmes de règles de production: techniques de compilation et d'interpretation", Colloque Intern. D'Intelligence Artificielle, 89-103, Marseille, 1984.

[GHALLAB 88]

GHALLAB M., "Compilation de bases de connaissances", PRC - Greco, Actes des Journees Nationales, pp. 231-253, Toulouse, 1988.

[GUIDA 84]

GUIDA G., TASSO C., "A new approach to the design of expert system architectures", Artificial Intelligence and Information-Control Systems of Robots, I. Plander (editor), Elsevier Science Publishers B.V. (North Holland), 1984.

[HARMON 85]

HARMON P., KING D., "Expert systems: artificial intelligence in business", John Wiley & Sons , 1985.

[HAYES-ROTH 75]

HAYES-ROTH F., MOSTOW D., "An automatically compilable recognition network for structured patterns", Proc. 4 o. IJCAI, pp. 246-251, Tbilissi, 1975.

[HAYES-ROTH 83]

HAYES-ROTH F., WATERMAN D., LENAT D.(eds.), "Building expert systems", Addison-Wesley, 1983.

[HOPCROFT 79]

HOPCROFT J., ULLMAN J., "Introduction to automata theory, languages and computation", Addison-Wesley, 1979.

[KONOLIGE 79]

KONOLIGE K., "An inference net compiler for the Prospector rule-based consultation system", Proc. 6 o. IJCAI, pp. 487-489, Tokyo, 1979.

[LAURENT 84]

LAURENT J.P., "La structure de contrôle dans les systemes experts", Technique et Science Informatiques, vol.3, no.3, 1984.

[LUCCHESI 77]

LUCCHESI C., SIMON I., SIMON I., SIMON J., KOWALTOWSKI T., "Aspectos teóricos da computação", 11 o. Colóquio Brasileiro de Matemática, Poços de Caldas, 1977.

[LUCENA 87]

LUCENA C., "Inteligência artificial e engenharia de software", Jorge Zahar ed., 1987.

[McCRACKEN 78]

McCRACKEN D.A., "A production system version of the Hearsay-II speech understanding system", Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1978.

[McDERMOTT 82]

McDERMOTT D., "R1: a rule-based configurer of computer systems", Artificial Intelligence 19, pp. 39-88, 1982.

[MENDELSON 79]

MENDELSON E., "Introduction to mathematical logic", 2 ed., Litton E.P., 1979.

[NII 79]

NII H., AIELLO N., "AGE: a knowledge-based program for building knowledge-based programs", Proc. 6 o. IJCAI, Tokyo, 1979.

[NILSSON 71]

NILSSON N., "Problem solving methods in artificial inteligence", McGraw-Hill, 1971.

[NILSSON 80]

NILSSON N., "Principles of artificial intelligence", Tioga P., 1980.

[RICH 83]

RICH E., "Artificial intelligence", McGraw-Hill, 1983.

[SHORTLIFFE 76]

SHORTLIFFE E.H., "Computer-based medical consultations Mycin", American Elsevier, 1976.

[SOBEK 87]

SOBEK R.P., "AND/OR matchs nets: an efficient production system representation", Rapport LAAS 87120, 1987.

[TURNER 86]

TURNER R., "Logiques pour l'intelligence artificielle", Masson, 1986.

[VELASCO 88a]

VELASCO F., "Manual da linguagem OPS5 para computadores compatíveis com IBM-PC (Versão 0.3)", INPE, 1988.

[VELASCO 88b]

VELASCO F., "Uma introdução à linguagem OPS5", INPE, 1988.

[VIALATTE 84]

VIALATTE M., "Introduction de metaconnaissance, de gestion d'hypothèses, de logiques d'ordre 0 et 2 dans SNARK", Rapport de l'Institut de Programmation, 1984.

[WATERMAN 86]

WATERMAN D.A., "A guide to expert systems", Addison-Wesley, 1986.

[WERTZ 85]

WERTZ H., "LISP: une introduction a la programmation", Masson, 1985.

[WINSTON 84a]

WINSTON P., "Artificial intelligence", Addison-Wesley, 2ed., 1984.

[WINSTON 84b]

WINSTON P., HORN B., "LISP", Addison-Wesley, 2 ed., 1984.